Параллельное программирование на C++ в действии

Энтони Уильямс

Практика разработки многопоточных программ

ДМК Пресс

MANNING

Annotation

В наши дни компьютеры с несколькими многоядерными процессорами стали нормой. Стандарт C++11 языка C++ предоставляет развитую поддержку многопоточности в приложениях. Поэтому, чтобы сохранять конкурентоспособность, вы должны овладеть принципами и приемами их разработки, а также новыми средствами языка, относящимися к параллелизму.

Книга «Параллельное программирование на C++ в действии» не предполагает предварительных знаний в этой области. Вдумчиво читая ее, вы научитесь писать надежные и элегантные многопоточные программы на C++11. Вы узнаете о том, что такое потоковая модель памяти, и о том, какие средства поддержки многопоточности, в том числе запуска и синхронизации потоков, имеются в стандартной библиотеке. Попутно вы познакомитесь с различными нетривиальными проблемами программирования в условиях параллелизма.

Энтони Уильямс
Параллельное программирование на C++ в действии
Практика разработки многопоточных программ

Ким, Хью и Ирен.

Предисловие

С идеей многопоточного программирования я столкнулся на своей первой работе после окончания колледжа. Мы занимались приложением, которое должно было помещать входные записи в базу данных. Данных было много, но все они были независимы и требовали значительной предварительной обработки. Чтобы задействовать всю мощь нашего десятипроцессорного компьютера UltraSPARC, мы организовали несколько потоков, каждый из которых обрабатывал свою порцию входных данных. Код был написан на языке C++, с использованием потоков POSIX. Ошибок мы наделали кучу — многопоточность для всех была внове — но до конца все-таки добрались. Именно во время работы над этим проектом я впервые услыхал о комитете по стандартизации C++ и о недавно опубликованном стандарте языка C++.

С тех мой интерес к многопоточному программированию и параллелизму не затухает. Там, где другим видятся трудности и источник разнообразных проблем, я нахожу мощный инструмент, который позволяет программе использовать всё наличное оборудование и в результате работать быстрее. Позднее я научился применять эти идеи и при наличии всего одного процессора или ядра, чтобы улучшить быстроту реакции и повысить производительность, — благодаря тому, что одновременная работа нескольких потоков дает программе возможность не простаивать во время таких длительных операций, как ввод/вывод. Я также узнал, как это устроено на уровне ОС и как в процессорах Intel реализовано контекстное переключение задач.

Тем временем интерес к C++ свел меня с членами Ассоциации пользователей C и C++ (ACCU), а затем с членами комиссии по стандартизации C++ при Институте стандартов Великобритании (BSI) и разработчиками библиотек Boost. Я с интересом наблюдал за началом разработки библиотеки многопоточности Boost, а когда автор забросил проект, я воспользовался шансом перехватить инициативу. С тех пор разработка и сопровождение библиотеки Boost Thread Library лежит в основном на мне.

По мере того как в работе комитета по стандартизации C++ наметился сдвиг от исправления дефектов в существующем стандарте в сторону выработки предложений для нового стандарта (получившего условное название C++0x в надежде, что его удастся завершить до 2009 года, и официально названного C++11, так как он наконец был опубликован в 2011 году), я стал принимать более активное участие в деятельности BSI и даже вносить собственные предложения. Когда стало ясно, что многопоточность стоит на повестке дня, я по-настоящему встрепенулся — многие вошедшие в стандарт предложения по многопоточности и параллелизму написаны как мной самим, так и в соавторстве с коллегами. Я считаю большой удачей, что таким образом удалось совместить две основных сферы моих интересов в области программирования — язык C++ и многопоточность.

В этой книге, опирающейся на весь мой опыт работы с C++ и многопоточностью, я ставил целью научить других программистов, как безопасно и эффективно пользоваться библиотекой C++11 Thread Library. Надеюсь, что мне удастся заразить читателей своим энтузиазмом.

Благодарности

Прежде всего, хочу сказать огромное спасибо своей супруге, Ким, за любовь и поддержку, которую она выказывала на протяжении работы над книгой, отнимавшей изрядную долю моего свободного времени за последние четыре года. Без ее терпения, ободрения и понимания я бы не справился.

Далее я хочу поблагодарить коллектив издательства Manning, благодаря которому эта книга появилась на свет: Марджана Баджи (Marjan Bace), главного редактора; Майкла Стивенса (Michael Stephens), его заместителя; Синтию Кейн (Cynthia Kane), моего редактора-консультанта; Карен Тегтмейер (Karen Tegtmeyer), выпускающего редактора; Линду Ректенвальд (Linda Recktenwald), редактора; Кати Теннант (корректора) и Мэри Пирджис, начальника производства. Без их стараний вы не читали бы сейчас эту книгу. Я хочу также поблагодарить других членов комитета по стандартизации C++, которые подавали на рассмотрение материалы, относящиеся к многопоточности: Андрея Александреску (Andrei Alexandrescu), Пита Беккера (Pete Becker), Боба Блэйнера (Bob Blainer), Ханса Бема (Hans Boehm), Бимана Доуса (Beman Dawes), Лоуренса Кроула (Lawrence Crowl), Петера Димова (Peter Dimov), Джеффа Гарланда (Jeff Garland), Кевлина Хэнни (Kevlin Henney), Ховарда Хиннанта (Howard Hinnant), Бена Хатчингса (Ben Hutchings), Йана Кристоферсона (Jan Kristofferson), Дуга Ли (Doug Lea), Пола Маккинни (Paul McKenney), Ника Макларена (Nick McLaren), Кларка Нельсона (Clark Nelson), Билла Пью (Bill Pugh), Рауля Силвера (Raul Silvera), Герба Саттера (Herb Sutter), Детлефа Вольмана (Detlef Vollmann) и Майкла Вонга (Michael Wong), а также всех тех, кто рецензировал материалы, принимал участие в их обсуждении на заседаниях комитета и иными способами содействовал оформлению поддержки многопоточности и параллелизма в C++11.

Наконец, хочу выразить благодарность людям, чьи предложения позволили заметно улучшить книгу: д-ру Джейми Оллсопу (Jamie Allsop), Петеру Димову, Ховарду Хиннанту, Рику Моллою (Rick Molloy), Джонатану Уэйкли (Jonathan Wakely) и д-ру Расселу Уиндеру (Russel Winder). Отдельное спасибо Расселу за подробные рецензии и Джонатану, который в качестве технического редактора, тщательно проверил окончательный текст на предмет наличия вопиющих ошибок. (Все оставшиеся ошибки — целиком моя вина.) И напоследок выражаю признательность группе рецензентов: Райану Стивенсу (Ryan Stephens), Нилу Хорлоку (Neil Horlock), Джону Тейлору младшему (John Taylor Jr.), Эзре Дживану (Ezra Jivan), Джошуа Хейеру (Joshua Heyer), Киту С. Киму (Keith S. Kim), Мишель Галли (Michele Galli) Майку Тянь-Чжань Чжану (Mike Tian-Jian Jiang), Дэвиду Стронгу (David Strong), Роджеру Орру (Roger Orr), Вагнеру Рику (Wagner Rick), Майку Буксасу (Mike Buksas) и Бас Воде (Bas Vodde). Также спасибо всем читателям предварительного издания, которые нашли время указать на ошибки и отметить места, нуждающиеся в уточнении.

Об этой книге

Эта книга представляет собой углубленное руководство по средствам поддержки многопоточности и параллелизма в новом стандарте C++, от базового использования классов и функций из пространств имен `std::thread`, `std::mutex` и `std::async` до сложных вопросов, связанных с атомарными операциями и моделью памяти.

Структура книги

В первых четырех главах описываются различные библиотечные средства и порядок работы с ними.

Глава 5 посвящена низкоуровневым техническим деталям модели памяти и атомарных операций. В частности, рассматривается вопрос об использовании атомарных операций для задания ограничений на порядок выполнения других частей программы. Вводные главы на этом заканчиваются.

В главах 6 и 7 начинается изучение программирования на более высоком уровне, с примерами использования базовых средств для построения сложных структур данных — с блокировками (глава 6) и без блокировок (глава 7).

В главе 8 эта линия продолжается: даются рекомендации по проектированию многопоточных программ, рассматриваются аспекты, влияющие на производительность, и приводятся примеры реализации различных параллельных алгоритмов.

Глава 9 посвящена средствам управления потоками, рассматриваются пулы потоков, очереди работ и прерывание операций.

Тема главы 10 — тестирование и отладка: типы ошибок, методы их отыскания, способы тестирования и так далее.

В приложениях вы найдете краткое описание некоторых языковых средств, добавленных в новый стандарт и имеющих отношение к многопоточности; детали реализации библиотеки передачи сообщениями, упомянутой в главе 4, и полный справочник по библиотеке C++11 Thread Library.

На кого рассчитана эта книга

Если вы пишете многопоточный код на C++, то эта книга для вас. Если вы пользуетесь средствами многопоточности из стандартной библиотеки C++, то здесь вы найдете руководство по основным вопросам. Если вы работаете с другими библиотеками многопоточности, то описанные рекомендации и приемы все равно могут оказаться полезным подспорьем.

Предполагается владение языком C++ на рабочем уровне, по предварительное знакомство с новыми языковыми средствами необязательно — они описаны в приложении A. Также не требуются знания или опыт работы в области многопоточного программирования, хотя их наличие было бы плюсом.

Как пользоваться этой книгой

Если раньше вы не писали многопоточных программ, то я рекомендую читать книгу последовательно от начала до конца, опустив, быть может, кое-какие детали из главы 5. Глава 7 опирается на материал главы 5, поэтому если вы пропустите главу 5, то отложите также чтение седьмой главы.

Если вам не доводилось использовать новые языковые средства, вошедшие в стандарт C ++11, то имеет смысл с самого начала бегло просмотреть приложение A, чтобы понимать приведенные в тексте примеры. Впрочем, в основном тексте упоминания о новых средствах графически выделены, так что, встретив что-то незнакомое, вы всегда можете обратиться к приложению.

Если вы располагаете обширным опытом написания многопоточного кода в других средах, то все-таки стоит просмотреть печальные главы, чтобы попять, как знакомые вам понятия соответствуют средствам из нового стандарта C++. Если вы планируете работать с атомарными переменными на низком уровне, то главу 5 следует изучить обязательно. Полезно также ознакомиться с главой 8, где рассказывается о безопасности исключений в многопоточных программах на C++. Если перед вами стоит конкретная задача, то указатель и оглавление помогут быстро найти соответствующий раздел.

Даже после того как вы освоите библиотеку C++ Thread Library, приложение D все равно останется полезным, потому что в нем легко найти детали использования каждого класса и функции. Время от времени вы, наверное, будет заглядывать и в основные главы, чтобы освежить в памяти порядок работы с той или иной конструкцией или взглянуть на пример кода.

Графические выделения и загрузка исходного кода

Исходный код в листингах и основном тексте набран `моноширинным шрифтом`. Многие листинги сопровождаются аннотациями, в которых излагаются важные концепции. В некоторых случаях в листингах присутствуют нумерованные маркеры, с которыми соотносятся последующие пояснения.

Исходный код всех примеров можно скачать с сайта издательства по адресу www.manning.com/CPlusPlusConcurrencyinAction.

Требования к программному обеспечению

Чтобы приведенный в этой книге код работал без модификаций, понадобится версия компилятора C++ с поддержкой тех вошедших в стандарт C++11 средств, которые перечислены в приложении A. Кроме того, нужна стандартная библиотека многопоточности C++ (Standard Thread Library).

На момент написания этой книги единственный известный мне компилятор, поставляемый с библиотекой Standard Thread Library, — это g++, хотя в предварительную версию Microsoft Visual Studio 2011 она также входит. Что касается g++, то первая реализация

основных возможностей библиотеки многопоточности была включена в версию g++ 4.3, а впоследствии добавлялись улучшения и расширения. Кроме того, в g++ 4.3 впервые появилась поддержка некоторых новых языковых средств C++11, и в каждой новой версии она расширяется. Дополнительные сведения см. на странице текущего состояния реализации C++11 в g++[1].

В составе Microsoft Visual Studio 2010 также имеются некоторые новые средства из стандарта C++11, например лямбда-функции и ссылки на r-значения, по реализация библиотеки Thread Library отсутствует.

Моя компания, Just Software Solutions Ltd, продает полную реализацию стандартной библиотеки C++11 Standard Thread Library для Microsoft Visual Studio 2005, Microsoft Visual Studio 2008, Microsoft Visual Studio 2010 различных версий g++[2]. Именно эта реализация применялась для тестирования примеров из этой книги.

В библиотеке Boost Thread Library[3], протестированной на многих платформах, реализовал API, основанный на предложениях, поданных в комитет по стандартизации C++. Большинство приведенных в книге примеров будут работать с Boost Thread Library, если заменить `std::` на `boost::` и включить подходящие директивы `#include`. Но некоторые возможности в библиотеке Boost Thread Library либо не поддерживаются вовсе (например, `std::async`), либо называются по-другому (например, `boost::unique_future`).

Автор в Сети

Приобретение книги «Параллелизм на C++ в действии» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/CPlusPlusConcurrencyinAction. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестает печататься.

Об иллюстрации на обложке

Рисунок на обложке книги «Параллельное программирование на C++ в действии» называется «Традиционный костюм японской девушки». Репродукция взята из четырехтомного «Собрания костюмов разных пародов», напечатанного в Лондоне между 1757 и 1772 годом. Это издание, включающее изумительные раскрашенные вручную гравюры на меди с изображениями одежды пародов мира, оказало большое влияние на дизайн театральных костюмов. Разнообразие рисунков позволяет составить наглядное представление о великолепии костюма на Лондонской сцене свыше 200 лет назад. Костюмы, исторические и того времени, позволяли познакомиться с традиционной одеждой людей, живших в разное время в разных странах, и тем самым сделать их ближе и понятнее лондонской театральной публике.

Манера одеваться за последние 100 лет сильно изменилась, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов. Но можно взглянуть на это и с оптимизмом — мы обменяли культурное и визуальное разнообразие на иное устройство личной жизни — основанное на многостороннем и стремительном технологическом и интеллектуальном развитии.

Издательство Manning откликается на новации, инициативы и курьезы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов и театральной жизни в позапрошлом веке. Мы возвращаем его в виде иллюстраций из этой коллекции.

Глава 1.
Здравствуй, параллельный мир!

В этой главе:

- Что понимается под параллелизмом и многопоточностью.
- Зачем использовать параллелизм и многопоточность в своих приложениях.
- Замечания об истории поддержки параллелизма в C++.
- Структура простой многопоточной программы на C++.

Для программистов на языке C++ настали радостные дни. Спустя тринадцать лет после публикации первой версии стандарта C++ в 1998 году комитет по стандартизации C++ решил основательно пересмотреть как сам язык, так и поставляемую вместе с ним библиотеку. Новый стандарт C++ (обозначаемый C++11 или C++0x), опубликованный в 2010 году, несёт многочисленные изменения, призванные упростить программирование на C++ и сделать его более продуктивным.

К числу наиболее существенных новшеств в стандарте C++11 следует отнести поддержку многопоточных программ. Впервые комитет официально признал существование многопоточных приложений, написанных на C++, и включил в библиотеку компоненты для их разработки. Это позволит писать на C++ многопоточные программы с гарантированным поведением, не полагаясь на зависящие от платформы расширения. И как раз вовремя, потому что разработчики, стремясь повысить производительность приложений, все чаще посматривают в сторону параллелизма вообще и многопоточного программирования в особенности.

Эта книга о том, как писать на C++ параллельные программы с несколькими потоками и о тех средствах самого языка и библиотеки времени выполнения, благодаря которым это стало возможно. Я начну с объяснения того, что понимаю под параллелизмом и многопоточностью и для чего это может пригодиться в приложениях. После краткого отвлечения на тему о том, когда программу *не* следует делать многопоточной, я в общих чертах расскажу о поддержке параллелизма в C++ и закончу главу примером простой параллельной программы. Читатели, имеющие опыт разработки многопоточных приложений, могут пропустить начальные разделы. В последующих главах мы рассмотрим более сложные примеры и детально изучим библиотечные средства. В конце книги приведено подробное справочное руководство по всем включенным в стандартную библиотеку C++ средствам поддержки многопоточности и параллелизма.

Итак, что же я понимаю под *параллелизмом* и *многопоточностью*?

1.1. Что такое параллелизм?

Если упростить до предела, то параллелизм — это одновременное выполнение двух или более операций. В жизни он встречается на каждом шагу: мы можем одновременно идти и разговаривать или одной рукой делать одно, а второй — другое. Ну и, разумеется, каждый из нас живет своей жизнью независимо от других — вы смотрите футбол, я в это время плаваю и т.д.

1.1.1. Параллелизм в вычислительных системах

Говоря о параллелизме в контексте компьютеров, мы имеем в виду, что одна и та же система выполняет несколько независимых операций параллельно, а не последовательно. Идея не нова: многозадачные операционные системы, позволяющие одновременно запускать на одном компьютере несколько приложений с помощью переключения между задачами уже много лет как стали привычными, а дорогие серверы с несколькими процессорами, обеспечивающие истинный параллелизм, появились еще раньше. *Новым* же является широкое распространение компьютеров, которые не просто создают иллюзию одновременного выполнения задач, а действительно исполняют их параллельно.

Исторически компьютеры, как правило, оснащались одним процессором с одним блоком обработки, или ядром, и это остается справедливым для многих настольных машин и по сей день. Такая машина в действительности способна исполнять только одну задачу в каждый момент времени, по может переключаться между задачами много раз в секунду. Таким образом, сначала одна задача немножко поработает, потом другая, а в итоге складывается впечатление, будто все происходит одновременно. Это называется *переключением задач*. Тем не менее, и для таких систем мы можем говорить о *параллелизме:* задачи сменяются очень часто и заранее нельзя сказать, в какой момент процессор приостановит одну и переключится на другую. Переключение задач создает иллюзию параллелизма не только у пользователя, но и у самого приложения. Но так как это всего лишь *иллюзия*, то между поведением приложения в однопроцессорной и истинно параллельной среде могут существовать топкие различия. В частности, неверные допущения о модели памяти (см. главу 5) в однопроцессорной среде могут не проявляться. Подробнее эта тема рассматривается в главе 10.

Компьютеры с несколькими процессорами применяются для организации серверов и выполнения высокопроизводительных вычислений уже много лет, а теперь машины с несколькими ядрами на одном кристалле (многоядерные процессоры) все чаще используются в качестве настольных компьютеров. И неважно, оснащена машина несколькими процессорами или одним процессором с несколькими ядрами (или комбинацией того и другого), она все равно может исполнять более одной задачи в каждый момент времени. Это называется *аппаратным параллелизмом*.

На рис. 1.1 показан идеализированный случай: компьютер, исполняющий ровно две задачи, каждая из которых разбита на десять одинаковых этапов. На двухъядерной машине каждая задача может исполняться в своем ядре. На одноядерной машине с переключением задач этапы той и другой задачи чередуются. Однако между ними существует крохотный промежуток времени (на рисунке эти промежутки изображены в виде серых полосок,

разделяющих более широкие этапы выполнения) — чтобы обеспечить чередование, система должна произвести *контекстное переключение* при каждом переходе от одной задачи к другой, а на это требуется время. Чтобы переключить контекст, ОС должна сохранить состояние процессора и счетчик команд для текущей задачи, определить, какая задача будет выполняться следующей, и загрузить в процессор состояние новой задачи. Не исключено, что затем процессору потребуется загрузить команды и данные новой задачи в кэш-память; в течение этой операции никакие команды не выполняются, что вносит дополнительные задержки.

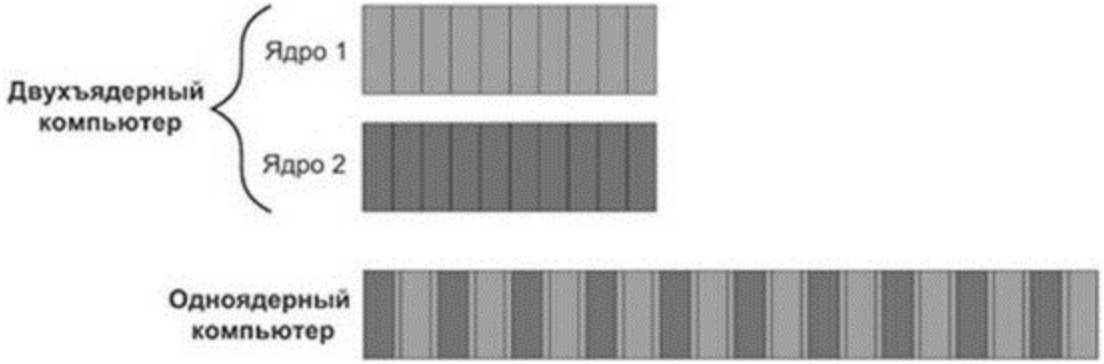

Рис. 1.1. Два подхода к параллелизму: параллельное выполнение на двухъядерном компьютере и переключение задач на одноядерном

Хотя аппаратная реализация параллелизма наиболее наглядно проявляется в многопроцессорных и многоядерных компьютерах, существуют процессоры, способные выполнять несколько потоков на одном ядре. В действительности существенным фактором является количество *аппаратных потоков* характеристика числа независимых задач, исполняемых оборудованием по-настоящему одновременно. И наоборот, в системе с истинным параллелизмом количество задач может превышать число ядер, тогда будет применяться механизм переключения задач. Например, в типичном настольном компьютере может быть запущено несколько сотен задач, исполняемых в фоновом режиме даже тогда, когда компьютер по видимости ничего не делает. Именно за счет переключения эти задачи могут работать параллельно, что и позволяет одновременно открывать текстовый процессор, компилятор, редактор и веб-браузер (да и вообще любую комбинацию приложений). На рис. 1.2 показано переключение четырех задач на двухъядерной машине, опять-таки в идеализированном случае, когда задачи разбиты на этапы одинаковой продолжительности. На практике существует много причин, из-за которых разбиение неравномерно и планировщик выделяет процессор каждой задаче не столь регулярно. Некоторые из них будут рассмотрены в главе 8 при обсуждении факторов, влияющих на производительность параллельных программ.

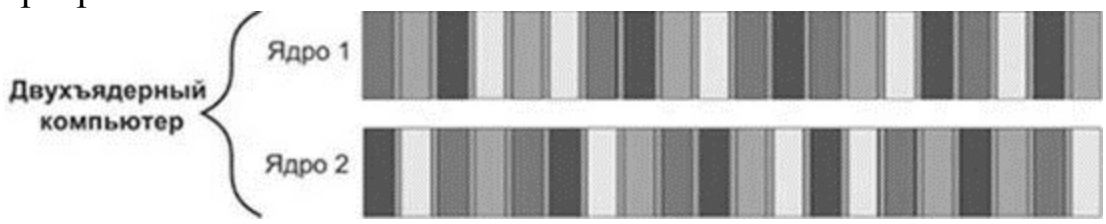

Рис. 1.2. Переключение задач на двухъядерном компьютере

Все рассматриваемые в этой книге приемы, функции и классы применимы вне зависимости оттого, исполняется приложение на машине с одноядерным процессором или с несколькими многоядерными процессорами. Не имеет значения, как реализован параллелизм: с помощью переключения задач или аппаратно. Однако же понятно, что

способ использования параллелизма в приложении вполне может зависеть от располагаемого оборудования. Эта тема обсуждается в главе 8 при рассмотрении вопросов проектирования параллельного кода на C++.

1.1.2. Подходы к организации параллелизма

Представьте себе пару программистов, работающих над одним проектом. Если они сидят в разных кабинетах, то могут мирно трудиться, не мешая друг другу, причем у каждого имеется свой комплект документации. Но общение при этом затруднено вместо того чтобы просто обернуться и обменяться парой слов, приходится звонить по телефону, писать письма или даже встать и дойти до коллеги. К тому же, содержание двух кабинетов сопряжено с издержками, да и на несколько комплектов документации надо будет потратиться.

А теперь представьте, что всех разработчиков собрали в одной комнате. У них появилась возможность обсуждать между собой проект приложения, рисовать на бумаге или на доске диаграммы, обмениваться мыслями. Содержать придется только один офис и одного комплекта документации вполне хватит. Но есть и минусы теперь им труднее сконцентрироваться и могут возникать проблемы с общим доступом к ресурсам («Ну куда опять запропастилось это справочное руководство?»).

Эти два способа организации труда разработчиков иллюстрируют два основных подхода к параллелизму. Разработчик это модель потока, а кабинет модель процесса В первом случае имеется несколько однопоточных процессов (у каждого разработчика свой кабинет), во втором несколько потоков в одном процессе (два разработчика в одном кабинете). Разумеется, возможны произвольные комбинации: может быть несколько процессов, многопоточных и однопоточных, но принцип остается неизменным. А теперь поговорим немного о том, как эти два подхода к параллелизму применяются в приложениях.

Параллелизм за счет нескольких процессов

Первый способ распараллелить приложение — разбить его на несколько однопоточных одновременно исполняемых процессов. Именно так вы и поступаете, запуская вместе браузер и текстовый процессор. Затем эти отдельные процессы могут обмениваться сообщениями, применяя стандартные каналы межпроцессной коммуникации (сигналы, сокеты, файлы, конвейеры и т.д.), как показано на рис. 1.3. Недостаток такой организации связи между процессами в его сложности, медленности, а иногда том и другом вместе. Дело в том, что операционная система должна обеспечить защиту процессов, так чтобы ни один не мог случайно изменить данные, принадлежащие другому. Есть и еще один недостаток — неустранимые накладные расходы на запуск нескольких процессов: для запуска процесса требуется время, ОС должна выделить внутренние ресурсы для управления процессом и т.д.

Рис. 1.3. Коммуникация между двумя параллельно работающими процессами

Конечно, есть и плюсы. Благодаря надежной защите процессов, обеспечиваемой операционной системой, и высокоуровневым механизмам коммуникации написать *безопасный* параллельный код проще, когда имеешь дело с процессами, а не с потоками. Например, в среде исполнения, создаваемой языком программирования Erlang, в качестве фундаментального механизма параллелизма используются процессы, и это дает отличный эффект.

У применения процессов для реализации параллелизма есть и еще одно достоинство — процессы можно запускать на разных машинах, объединенных сетью. Хотя затраты на коммуникацию при этом возрастают, по в хорошо спроектированной системе такой способ повышения степени параллелизма может оказаться очень эффективным, и общая производительность увеличится.

Параллелизм за счет нескольких потоков

Альтернативный подход к организации параллелизма — запуск нескольких потоков в одном процессе. Потоки можно считать облегченными процессами — каждый поток работает независимо от всех остальных, и все потоки могут выполнять разные последовательности команд. Однако все принадлежащие процессу потоки разделяют общее адресное пространство и имеют прямой доступ к большей части данных — глобальные переменные остаются глобальными, указатели и ссылки на объекты можно передавать из одного потока в другой. Для процессов тоже можно организовать доступ к разделяемой памяти, но это и сделать сложнее, и управлять не так просто, потому что адреса одного и того же элемента данных в разных процессах могут оказаться разными. На рис. 1.4 показано, как два потока в одном процессе обмениваются данными через разделяемую память.

Рис. 1.4. Коммуникация между двумя параллельно исполняемыми потоками в одном процессе

Благодаря общему адресному пространству и отсутствию защиты данных от доступа со стороны разных потоков накладные расходы, связанные с наличием нескольких потоков, существенно меньше, так как на долю операционной системы выпадает гораздо меньше учетной работы, чем в случае нескольких процессов. Однако же за гибкость разделяемой памяти приходится расплачиваться — если к некоторому элементу данных обращаются несколько потоков, то программист должен обеспечить согласованность представления этого элемента во всех потоках. Возникающие при этом проблемы, а также средства и рекомендации по их разрешению рассматриваются на протяжении всей книги, а особенно в главах 3, 4, 5 и 8. Эти проблемы не являются непреодолимыми, надо лишь соблюдать осторожность при написании кода. Но само их наличие означает, что коммуникацию между потоками необходимо тщательно продумывать.

Низкие накладные расходы на запуск потоков внутри процесса и коммуникацию между ними стали причиной популярности этого подхода во всех распространенных языках программирования, включая C++, даже несмотря на потенциальные проблемы, связанные с разделением памяти. Кроме того, в стандарте C++ не оговаривается встроенная поддержка межпроцессной коммуникации, а, значит, приложения, основанные на применении нескольких процессов, вынуждены полагаться на платформенно-зависимые API. Поэтому в этой книге мы будем заниматься исключительно параллелизмом на основе многопоточности, и в дальнейшем всякое упоминание о параллелизме предполагает использование нескольких потоков.

Определившись с тем, что понимать под параллелизмом, посмотрим, зачем он может понадобиться в приложениях.

1.2. Зачем нужен параллелизм?

Существует две основных причины для использования параллелизма в приложении: разделение обязанностей и производительность. Я бы даже рискнул сказать, что это *единственные* причины — если внимательно приглядеться, то окажется, что все остальное сводится к одной или к другой (или к обеим сразу). Ну, конечно, если не рассматривать в качестве аргумента «потому что я так хочу».

1.2.1. Применение параллелизма для разделения обязанностей

Разделение обязанностей почти всегда приветствуется при разработке программ: если сгруппировать взаимосвязанные и разделить несвязанные части кода, то программа станет проще для понимания и тестирования и, стало быть, будет содержать меньше ошибок. Использовать распараллеливание для разделения функционально не связанных между собой частей программы имеет смысл даже, если относящиеся к разным частям операции должны выполняться одновременно: без явного распараллеливания нам пришлось бы либо реализовать какую-то инфраструктуру переключения задач, либо то и дело обращаться к коду из посторонней части программы во время выполнения операции.

Рассмотрим приложение, имеющее графический интерфейс и выполняющее сложные вычисления, например DVD-проигрыватель на настольном компьютере. У такого приложения два принципиально разных набора обязанностей: во-первых, читать данные с диска, декодировать изображение и звук и своевременно посылать их графическому и звуковому оборудованию, чтобы при просмотре фильма не было заминок, а, во-вторых, реагировать на действия пользователя, например, на нажатие кнопок «Пауза», «Возврат в меню» и даже «Выход». Если бы приложение было однопоточным, то должно было бы периодически проверять, не было ли каких-то действий пользователя, поэтому код воспроизведения DVD перемежался бы кодом, относящимся к пользовательскому интерфейсу Если же для разделения этих обязанностей использовать несколько потоков, то код интерфейса и воспроизведения уже не будут так тесно переплетены: один поток может заниматься отслеживанием действий пользователя, а другой - воспроизведением. Конечно, как-то взаимодействовать они все равно должны, например, если пользователь нажимает кнопку «Пауза», но такого рода взаимодействия непосредственно связаны с решаемой задачей.

В результате мы получаем «отзывчивый» интерфейс, так как поток пользовательского интерфейса обычно способен немедленно отреагировать на запрос пользователя, даже если реакция заключается всего лишь в смене формы курсора на «занято» или выводе сообщения «Подождите, пожалуйста» на время, требуемое для передачи запроса другому потоку для обработки. Аналогично, несколько потоков часто создаются для выполнения постоянно работающих фоновых задач, например, мониторинга изменений файловой системы в приложении локального поиска. Такое использование потоков позволяет существенно упростить логику каждого потока, так как взаимодействие между ними ограничено немногими четко определенными точками, а не размазано по всей программе.

В данном случае количество потоков не зависит от количества имеющихся процессорных ядер, потому что программа разбивается на потоки ради чистоты дизайна, а

не в попытке увеличить производительность.

1.2.2. Применение параллелизма для повышения производительности

Многопроцессорные системы существуют уже десятки лет, но до недавнего времени они использовались исключительно в суперкомпьютерах, больших ЭВМ и крупных серверах. Однако ныне производители микропроцессоров предпочитают делать процессоры с 2, 4, 16 и более ядрами на одном кристалле, а не наращивать производительность одного ядра. Поэтому все большее распространение получают настольные компьютеры и даже встраиваемые устройства с многоядерными процессорами. Увеличение вычислительной мощи в этом случае связано не с тем, что каждая отдельная задача работает быстрее, а с тем, что несколько задач исполняются параллельно.

В прошлом программист мог откинуться на спинку стула и наблюдать, как его программа работает все быстрее с каждым новым поколением процессоров, без каких-либо усилий с его стороны. Но теперь, как говорит Герб Саттер, «время бесплатных завтраков закончилось» [Sutter 2005]. *Если требуется, чтобы программа выигрывала от увеличения вычислительной мощности, то ее необходимо проектировать как набор параллельных задач*. Поэтому программистам придется подтянуться, и те, кто до сих пор не обращал внимания на параллелизм, должны будут добавить его в свой арсенал.

Существует два способа применить распараллеливание для повышения производительности. Первый, самый очевидный, разбить задачу на части и запустить их параллельно, уменьшив тем самым общее время выполнения. Это *распараллеливание по задачам*. Хотя эта процедура и представляется простой, на деле все может сильно усложниться из-за наличия многочисленных зависимостей между разными частями. Разбиение можно формулировать как в терминах обработки: один поток выполняет одну часть обработки, другой — другую, так и в терминах данных: каждый поток выполняет одну и ту же операцию, но с разными данными. Последний вариант называется *распараллеливание по данным*.

Алгоритмы, легко поддающиеся такому распараллеливанию, часто называют *естественно параллельными* (embarrassingly parallel, naturally parallel, conveniently concurrent.). Они очень хорошо масштабируются — если число располагаемых аппаратных потоков увеличивается, то и степень параллелизма алгоритма возрастает. Такой алгоритм — идеальная иллюстрации пословицы «берись дружно, не будет грузно». Те части алгоритма, которые не являются естественно параллельными, можно разбить на фиксированное (и потому не масштабируемое) число параллельных задач. Техника распределения задач по потокам рассматривается в главе 8.

Второй способ применения распараллеливания для повышения производительности — воспользоваться имеющимся параллелизмом для решения более крупных задач, например, обрабатывать не один файл за раз, а сразу два, десять или двадцать. Это по сути дела пример распараллеливания но данным, так как одна и та же операция производится над несколькими наборами данных одновременно, но акцент немного иной. Для обработки одной порции данных требуется столько же времени, сколько и раньше, но за фиксированное время можно обработать больше данных. Очевидно, что и у этого подхода есть ограничения, и не во всех случаях он дает выигрыш, но достигаемое повышение производительности иногда открывает новые возможности. Например, если разные области изображения можно

обрабатывать параллельно, то можно будет обработать видео более высокого разрешения.

1.2.3. Когда параллелизм вреден?

Понимать, когда параллелизмом пользоваться *не* следует, не менее важно. Принцип простой: единственная причина не использовать параллелизм — ситуация, когда затраты перевешивают выигрыш. Часто параллельная программа сложнее для понимания, поэтому для написания и сопровождения многопоточного кода требуются дополнительные интеллектуальные усилия, а, стало быть, возрастает и количество ошибок. Если потенциальный прирост производительности недостаточно велик или достигаемое разделение обязанностей не настолько очевидно, чтобы оправдать дополнительные затраты времени на разработку, отладку и сопровождение многопоточной программы, то не используйте параллелизм.

Кроме того, прирост производительности может оказаться меньше ожидаемого: с запуском потоков связаны неустранимые накладные расходы, потому что ОС должна выделить ресурсы ядра и память для стека и сообщить о новом потоке планировщику, а на все это требуется время. Если задача, исполняемая в отдельном потоке, завершается быстро, то может оказаться, что в общем времени ее работы доминируют именно накладные расходы на запуск потока, поэтому производительность приложения в целом может оказаться хуже, чем если бы задача исполнялась в уже имеющемся потоке.

Далее, потоки — это ограниченный ресурс. Если одновременно работает слишком много потоков, то ресурсы ОС истощаются, что может привести к замедлению работы всей системы. Более того, при чрезмерно большом количестве потоков может исчерпаться память или адресное пространство, выделенное процессу, так как каждому потоку необходим собственный стек. Особенно часто эта проблема возникает в 32-разрядных процессах с «плоской» структурой памяти, где на размер адресного пространства налагается ограничение 4 ГБ: если у каждого потока есть стек размером 1 МБ (типичное соглашение во многих системах), то 4096 потоков займут все адресное пространство, не оставив места для кода, статических данных и кучи. В 64-разрядных системах (и системах с большей разрядностью слова) такого ограничения на размер адресного пространства нет, но ресурсы все равно конечны: если запустить слишком много потоков, то рано или поздно возникнут проблемы. Для ограничения количества потоков можно воспользоваться пулами потоков (см. главу 9), но и это не панацея — у пулов есть и свои проблемы.

Если на серверной стороне клиент-серверного приложения создается по одному потоку для каждого соединения, то при небольшом количестве соединений все будет работать прекрасно, но когда нагрузка на сервер возрастает и ему приходится обрабатывать очень много соединений, такая техника быстро приведет к истощению системных ресурсов. В такой ситуации оптимальную производительность может дать обдуманное применение пулов потоков (см. главу 9).

Наконец, чем больше работает потоков, тем чаще операционная система должна выполнять контекстное переключение. На каждое такое переключение уходит время, которое можно было бы потратить на полезную работу, поэтому в какой-то момент добавление нового потока не увеличивает, а *снижает* общую производительность приложения. Поэтому, пытаясь достичь максимально возможной производительности системы, вы должны выбирать число потоков с учетом располагаемого аппаратного параллелизма (или его

отсутствия).

Применение распараллеливания для повышения производительности ничем не отличается от любой другой стратегии оптимизации — оно может существенно увеличить скорость работы приложения, но при этом сделать код более сложным для понимания, что чревато ошибками. Поэтому распараллеливать имеет смысл только критически важные с точки зрения производительности участки программы, когда это может принести поддающийся измерению выигрыш. Но, конечно, если вопрос об увеличении производительности вторичен, а на первую роль выходит ясность дизайна или разделение обязанностей, то рассмотреть возможность многопоточной структуры все равно стоит.

Но предположим, что вы уже решили, что хотите распараллелить приложение, будь то для повышения производительности, ради разделения обязанностей или просто потому, что сегодня «День многопоточности». Что это означает для программиста на C++?

1.3. Параллелизм и многопоточность в C++

Стандартизованная поддержка параллелизма за счет многопоточности — вещь новая для C++. Только новый стандарт C++11 позволит писать многопоточный код, не прибегая к платформенно-зависимым расширениям. Чтобы разобраться в подоплёке многочисленных решений, принятых в новой стандартной библиотеке C++ Thread Library, необходимо вспомнить историю.

1.3.1. История многопоточности в C++

Стандарт C++ 1998 года не признавал существования потоков, поэтому результаты работы различных языковых конструкций описывались в терминах последовательной абстрактной машины. Более того, модель памяти не была формально определена, поэтому без поддержки со стороны расширений стандарта C++ 1998 года писать многопоточные приложения вообще было невозможно.

Разумеется, производители компиляторов вправе добавлять в язык любые расширения, а наличие различных API для поддержки многопоточности в языке C, например, в стандарте POSIX C Standard и в Microsoft Windows API, заставило многих производителей компиляторов C++ поддержать многопоточность с помощью платформенных расширений. Как правило, эта поддержка ограничивается разрешением использовать соответствующий платформе C API с гарантией, что библиотека времени исполнения C++ (в частности, механизм обработки исключений) будет корректно работать при наличии нескольких потоков. Хотя лишь очень немногие производители компиляторов предложили формальную модель памяти с поддержкой многопоточности, практическое поведение компиляторов и процессоров оказалось достаточно приемлемым для создания большого числа многопоточных программ на C++.

Не удовлетворившись использованием платформенно-зависимых C API для работы с многопоточностью, программисты на C++ пожелали, чтобы в используемых ими библиотеках классов были реализованы объектно-ориентированные средства для написания многопоточных программ. В различные программные каркасы типа MFC и в универсальные библиотеки на C++ типа Boost и ACE были включены наборы классов C++, которые обертывали платформенно-зависимые API и предоставляли высокоуровневые средства для работы с многопоточностью, призванные упростить программирование. Детали реализации в этих библиотеках существенно различаются, особенно в части запуска новых потоков, но общая структура классов очень похожа. В частности, во многих библиотеках классов C++ применяется крайне полезная идиома *захват ресурса есть инициализация (RAII)*, которая материализуется в виде блокировок, гарантирующих освобождение мьютекса при выходе из соответствующей области видимости.

Во многих случаях поддержка многопоточности в имеющихся компиляторах C++ вкупе с доступностью платформенно-зависимых API и платформенно-независимых библиотек классов типа Boost и ACE оказывается достаточно прочным основанием, на котором можно писать многопоточные программы. В результате уже написаны многопоточные приложения на C++, содержащие миллионы строк кода. Но коль скоро прямой поддержки в стандарте нет, бывают случаи, когда отсутствие модели памяти, учитывающей многопоточность, приводит к

проблемам. Особенно часто с этим сталкиваются разработчики, пытающиеся увеличить производительность за счет использования особенностей конкретного процессора, а также те, кто пишет кросс-платформенный код, который должен работать независимо от различий между компиляторами на разных платформах.

1.3.2. Поддержка параллелизма в новом стандарте

Все изменилось с выходом стандарта C++11. Мало того что в нем определена совершенно новая модель памяти с поддержкой многопоточности, так еще и в стандартную библиотеку C++ включены классы для управления потоками (глава 2), защиты разделяемых данных (глава 3), синхронизации операций между потоками (глава 4) и низкоуровневых атомарных операций (глава 5).

В основу новой библиотеки многопоточности для C++ положен опыт, накопленный за время использования вышеупомянутых библиотек классов. В частности, моделью новой библиотеки стала библиотека Boost Thread Library, из которой заимствованы имена и структура многих классов. Эволюция нового стандарта была двунаправленным процессом, и сама библиотека Boost Thread Library во многих отношениях изменилась, чтобы лучше соответствовать стандарту. Поэтому пользователи Boost, переходящие на новый стандарт, будут чувствовать себя очень комфортно.

Поддержка параллелизма — лишь одна из новаций в стандарте C++. Как уже отмечалось в начале главы, в сам язык тоже внесено много изменений, призванных упростить жизнь программистам. Хотя, вообще говоря, сами по себе они не являются предметом настоящей книги, некоторые оказывают прямое влияние на библиотеку многопоточности и способы ее использования. В приложении A содержится краткое введение в эти языковые средства.

Прямая языковая поддержка атомарных операций позволяет писать эффективный код с четко определенной семантикой, не прибегая к языку ассемблера для конкретной платформы. Это манна небесная для тех, кто пытается создавать эффективный и переносимый код, — мало того что компилятор берет на себя заботу об особенностях платформы, так еще и оптимизатор можно написать так, что он будет учитывать семантику операций и, стало быть, лучше оптимизировать программу в целом.

1.3.3. Эффективность библиотеки многопоточности для C++

Одна из проблем, с которыми сталкиваются разработчики высокопроизводительных приложений при использовании языка C++ вообще и классов, обертывающих низкоуровневые средства, типа тех, что включены в стандартную библиотеку C++ Thread Library, в частности, — это эффективность. Если вас интересует достижение максимальной производительности, то необходимо понимать, что использование любых высокоуровневых механизмов вместо обертываемых ими низкоуровневых средств влечет за собой некоторые издержки. Эти издержки называются *платой за абстрагирование*.

Комитет по стандартизации C++ прекрасно донимал это, когда проектировал стандартную библиотеку C++ вообще и стандартную библиотеку многопоточности для C++ в частности. Среди целей проектирования была и такая: выигрыш от использования

низкоуровневых средств по сравнению с высокоуровневой оберткой (если такая предоставляется) должен быть ничтожен или отсутствовать вовсе. Поэтому библиотека спроектирована так, чтобы ее можно было эффективно реализовать (с очень небольшой платой за абстрагирование) на большинстве популярных платформ.

Комитет по стандартизации C++ поставил и другую цель — обеспечить достаточное количество низкоуровневых средств для желающих работать на уровне «железа», чтобы выдавить из него все, что возможно. Поэтому наряду с новой моделью памяти включена полная библиотека атомарных операций для прямого управления на уровне битов и байтов, а также средства межпоточной синхронизации и обеспечения видимости любых изменений. Атомарные типы и соответствующие операции теперь можно использовать во многих местах, где раньше разработчики были вынуждены опускаться на уровень языка ассемблера для конкретной платформы. Таким образом, код с применением новых стандартных типов и операций получается более переносимым и удобным для сопровождения.

Стандартная библиотека C++ также предлагает высокоуровневые абстракции и средства, позволяющие писать многопоточный код проще и с меньшим количеством ошибок. Некоторые из них несколько снижают производительность из-за необходимости выполнять дополнительный код. Однако эти накладные расходы не обязательно означают высокую плату за абстрагирование: в общем случае цена не выше, чем пришлось бы заплатить при написании эквивалентной функциональности вручную, и к тому же компилятор волне может встроить значительную часть дополнительного кода.

В некоторых случаях высокоуровневые средства обеспечивают большую функциональность, чем необходимо для конкретной задачи. Как правило, это не страшно: вы не платите за то, чем не пользуетесь. Редко, но бывает, что избыточная функциональность негативно сказывается на производительности других частей программы. Если ее стоимость слишком высока, а производительность имеет первостепенное значение, то, быть может, имеет смысл вручную запрограммировать необходимую функциональность, пользуясь низкоуровневыми средствами. Но в подавляющем большинстве случаев дополнительная сложность и возможность внести ошибки намного перевешивают небольшой выигрыш в производительности. Даже если профилирование показывает, что средства стандартной библиотеки C++ *действительно* являются узким местом, не исключено, что проблема в неудачном дизайне приложения, а не в плохой реализации библиотеки. Например, когда слишком много потоков конкурируют за один мьютекс, производительность упадет — и сильно. Но лучше не пытаться чуть-чуть ускорить операции с мьютексами, а изменить структуру приложения, так чтобы снизить конкуренцию. Вопрос о том, как проектировать приложения, чтобы уменьшить конкуренцию, обсуждается в главе 8.

В тех крайне редких случаях, когда стандартная библиотека не обеспечивает необходимой производительности или поведения, может возникнуть необходимость в использовании платформенно-зависимых средств.

1.3.4. Платформенно-зависимые средства

Хотя библиотека многопоточности для C++ содержит достаточно полный набор средств для создания многопоточных программ, на любой платформе имеются специальные средства, помимо включенных в библиотеку. Чтобы можно было получить доступ к этим средствам, не отказываясь от использования стандартной библиотеки, типы, имеющиеся в

библиотеки многопоточности, иногда содержат функцию-член `native_handle()`, которая позволяет работать на уровне платформенного API. По природе своей любые операции, выполняемые с помощью функции `native_handle()`, зависят от платформы и потому в данной книге (как и в самой стандартной библиотеке C++) не рассматриваются.

Разумеется, перед тем задумываться о применении платформенно-зависимых средств, стоит как следует разобраться в том, что предлагает стандартная библиотека, поэтому начнем с примера.

1.4. В начале пути

Итак, вы получили новенький, с пылу с жару компилятор, совместимый со стандартом C++11. Что дальше? Как выглядит многопоточная программа на C++? Да примерно так же, как любая другая программа, — с переменными, классами и функциями. Единственное существенное отличие состоит в том, что некоторые функции могут работать параллельно, поэтому нужно следить за тем, чтобы доступ к разделяемым данным был безопасен (см. главу 3). Понятно, что для параллельного исполнения необходимо использовать специальные функции и объекты, предназначенные для управления потоками.

1.4.1. Здравствуй, параллельный мир

Начнем с классического примера — программы, которая печатает фразу «Здравствуй, мир». Ниже приведена тривиальная однопоточная программа такого сорта, от нее мы будем отталкиваться при переходе к нескольким потокам.

```
#include <iostream>
int main() {
 std::cout << "Здравствуй, мир\n";
}
```

Эта программа всего лишь выводит строку *Здравствуй мир* в стандартный поток вывода. Сравним ее с простой программой «Здравствуй, параллельный мир», показанной в листинге 1.1, — в ней для вывода сообщения запускается отдельный поток.

```
#include <iostream>
#include <thread> ← (1)


void hello()      ← (2)
{
 std::cout << "Здравствуй, параллельный мир\n";
}


int
main() {
 std::thread t(hello); ← (3)
 t.join();             ← (4)
}
```

Прежде всего, отметим наличие дополнительной директивы `#include <thread>` **(1)**. Все объявления, необходимые для поддержки многопоточности, помещены в новые заголовочные файлы; функции и классы для управления потоками объявлены в файле `<thread>`, а те, что нужны для защиты разделяемых данных, — в других заголовках.

Далее, код вывода сообщения перемещен в отдельную функцию **(2)**. Это объясняется тем, что в каждом потоке должна быть *начальная функция*, в которой начинается исполнение потока. Для первого потока в приложении таковой является `main()`, а для всех остальных задается в конструкторе объекта `std::thread`. В данном случае в качестве начальной функции объекта типа `std::thread`, названного `t` **(3)**, выступает функция `hello()`.

Есть и еще одно отличие вместо того, чтобы сразу писать на стандартный вывод или вызывать `hello()` из `main()`, эта программа запускает новый поток, так что теперь общее

число потоков равно двум: главный, с начальной функцией `main()`, и дополнительный, начинающий работу в функции `hello()`.

После запуска нового потока **(3)** начальный поток продолжает работать. Если бы он не ждал завершения нового потока, то просто дошел бы до конца `main()`, после чего исполнение программы закончилась бы быть может, еще до того, как у нового потока появился шанс начать работу. Чтобы предотвратить такое развитие событие, мы добавили обращение к функции `join()` **(4)**; в главе 2 объясняется, что это заставляет вызывающий поток (`main()`) ждать завершения потока, ассоциированного с объектом `std::thread`, — в данном случае `t`.

Если вам показалось, что для элементарного вывода сообщения на стандартный вывод работы слишком много, то так оно и есть, — в разделе 1.2.3 выше мы говорили, что обычно для решения такой простой задачи не имеет смысла создавать несколько потоков, особенно если главному потоку в это время нечего делать. Но далее мы встретимся с примерами, когда запуск нескольких потоков дает очевидный выигрыш.

1.5. Резюме

В этой главе мы говорили о том, что такое параллелизм и многопоточность и почему стоит (или не стоит) использовать их в программах. Мы также рассмотрели историю многопоточности в C++ — от полного отсутствия поддержки в стандарте 1998 года через различные платформенно-зависимые расширения к полноценной поддержке в новом стандарте C++11. Эта поддержка, появившаяся очень вовремя, дает программистам возможность воспользоваться преимуществами аппаратного параллелизма, которые стали доступны в современных процессорах, поскольку их производители пошли но пути наращивания мощности за счет реализации нескольких ядер, а не увеличения быстродействия одного ядра.

Мы также видели (пример в разделе 1.4), как просто использовать классы и функции из стандартной библиотеки C++. В C++ использование нескольких потоков само по себе несложно — сложно спроектировать программу так, чтобы она вела себя, как задумано.

Закусив примерами из раздела 1.4, пора приступить к чему-нибудь более питательному. В главе 1 мы рассмотрим классы и функции для управления потоками.

Глава 2.
Управление потоками

В этой главе:

- Запуск потоков и различные способы задания кода, исполняемого в новом потоке.
- Ждать завершения потока или позволить ему работать независимо?
- Уникальные идентификаторы потоков.

Итак, вы решили написать параллельную программу, а конкретно — использовать несколько потоков. И что теперь? Как запустить потоки, как узнать, что поток завершился, и как отслеживать их выполнение? Средства, имеющиеся в стандартной библиотеке, позволяют относительно просто решить большинство задач управления потоками. Как мы увидим, почти все делается с помощью объекта `std::thread`, ассоциированного с потоком. Для более сложных задач библиотека позволяет построить то, что нужно, из простейших кирпичиком.

Мы начнем эту главу с рассмотрения базовых операций: запуск потока, ожидание его завершения, исполнение в фоновом режиме. Затем мы поговорим о передаче дополнительных параметров функции потока в момент запуска и о том, как передать владение потока от одного объекта `std::thread` другому. Наконец, мы обсудим вопрос о том, сколько запускать потоков и как идентифицировать отдельный поток.

2.1. Базовые операции управления потоками

В каждой программе на C++ имеется по меньшей мере один поток, запускаемый средой исполнения C++: тот, в котором исполняется функция `main()`. Затем программа может запускать дополнительные потоки с другими функциями в качестве точки входа. Эти потоки работают параллельно друг с другом и с начальным потоком. Мы знаем, что программа завершает работу, когда `main()` возвращает управление; точно так же, при возврате из точки входа в поток этот поток завершается. Ниже мы увидим, что, имея объект `std::thread` для некоторого потока, мы можем дождаться завершения этого потока, но сначала посмотрим, как потоки запускаются.

2.1.1. Запуск потока

В главе 1 мы видели, что для запуска потока следует сконструировать объект `std::thread`, который определяет, какая задача будет исполняться в потоке. В простейшем случае задача представляет собой обычную функцию без параметров, возвращающую `void`. Эта функция работает в своем потоке, пока не вернет управление, и в этом момент поток завершается. С другой стороны, в роли задачи может выступать объект-функция, который принимает дополнительные параметры и выполняет ряд независимых операций, информацию о которых получает во время работы от той или иной системы передачи сообщений. И останавливается такой поток, когда получит соответствующий сигнал, опять же с помощью системы передачи сообщений. Вне зависимости от того, что поток будет делать и откуда он запускается, сам запуск потока в стандартном C++ всегда сводится к конструированию объекта `std::thread`:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

Как видите, все просто. Разумеется, как и во многих других случаях в стандартной библиотеке C++, класс `std::thread` работает с любым типом, допускающим вызов (*Callable*), поэтому конструктору `std::thread` можно передать экземпляр класса, в котором определен оператор вызова:

```
class background_task {
public:
 void operator()() const {
  do_something();
  do_something_else();
 }
};

background_task f;
std::thread my_thread(f);
```

В данном случае переданный объект-функция *копируется* в память, принадлежащую только что созданному потоку выполнения, и оттуда вызывается. Поэтому необходимо, чтобы с точки зрения поведения копия была эквивалентна оригиналу, иначе можно получить неожиданный результат.

При передаче объекта-функции конструктору потока нужно избегать феномена «самого досадного разбора в C++» (C++'s most vexing parse). Синтаксически передача конструктору временного объекта вместо именованной переменной выглядит так же, как объявление

функции, и именно так компилятор и интерпретирует эту конструкцию. Например, в предложении

```
std::thread my_thread(background_task());
```

объявлена функция `my_thread`, принимающая единственный параметр (типа указателя на функцию без параметров, которая возвращает объект `background_task`) и возвращающая объект `std::thread`. Никакой новый поток здесь не запускается. Решить эту проблему можно тремя способами: поименовать объект-функцию, как в примере выше; добавить лишнюю пару скобок или воспользоваться новым универсальным синтаксисом инициализации, например:

```
std::thread my_thread((background_task())); ← (1)
std::thread my_thread{background_task()};   ← (2)
```

В случае **(1)** наличие дополнительных скобок не дает компилятору интерпретировать конструкцию как объявление функции, так что действительно объявляется переменная `my_thread` типа `std::thread`. В случае **(2)** использован новый универсальный синтаксис инициализации с фигурными, а не круглыми скобками, он тоже приводит к объявлению переменной.

В стандарте C++11 имеется новый тип допускающего вызов объекта, в котором описанная проблема не возникает, — *лямбда-выражение*. Этот механизм позволяет написать локальную функцию, которая может захватывать некоторые локальные переменные, из-за чего передавать дополнительные аргументы просто не нужно (см. раздел 2.2). Подробная информация о лямбда-выражениях приведена в разделе A.5 приложения A. С помощью лямбда-выражений предыдущий пример можно записать в таком виде:

```
std::thread my_thread([](
 do_something();
 do_something_else();
});
```

После запуска потока необходимо явно решить, ждать его завершения (присоединившись к нему, см. раздел 2.1.2) или предоставить собственной судьбе (отсоединив его, см. раздел 2.1.3). Если это решение не будет принято к моменту уничтожения объекта `std::thread`, то программа завершится (деструктор `std::thread` вызовет функцию `std::terminate()`). Поэтому вы обязаны гарантировать, что поток корректно присоединен либо отсоединен, даже если возможны исключения. Соответствующая техника программирования описана в разделе 2.1.3. Отметим, что это решение следует принять именно до уничтожения объекта `std::thread`, к самому потоку оно не имеет отношения. Поток вполне может завершиться задолго до того, как программа присоединится к нему или отсоединит его. А отсоединенный поток может продолжать работу и после уничтожения объекта `std::thread`.

Если вы не хотите дожидаться завершения потока, то должны гарантировать, что данные, к которым поток обращается, остаются действительными до тех пор, пока они могут ему понадобиться. Эта проблема не нова даже в однопоточной программа доступ к уже уничтоженному объекту считается неопределенным поведением, но при использовании потоков есть больше шансов столкнуться с проблемами, обусловленными временем жизни.

Например, такая проблема возникает, если функция потока хранит указатели или ссылки на локальные переменные, и поток еще не завершился, когда произошел выход из области видимости, где эти переменные определены. Соответствующий пример приведен в листинге 2.1.

Листинг 2.1. Функция возвращает управление, когда поток имеет доступ к определенным в ней локальным переменным

```
struct func {
 int& i;
 func(int& i_) : i(i_){}
 void operator() () {
  for(unsigned j = 0; j < 1000000; ++j) {
   do_something(i); ←┐ Потенциальный доступ
  }                  (1) к висячей ссылке
 }
};

void oops() {
 int some_local_state = 0;          (2) Не ждем завершения
 func my_func(some_local_state); ←┘ потока
 std::thread my_thread(my_func); ←┐ Новый поток, возможно,
 my_thread.detach();               (3) еще работает
}
```

В данном случае вполне возможно, что новый поток, ассоциированный с объектом `my_thread`, будет еще работать, когда функция `oops` вернет управление **(2)**, поскольку мы явно решили не дожидаться его завершения, вызвав `detach()` **(3)**. А если поток *действительно* работает, то при следующем вызове `do_something(i)` **(1)** произойдет обращение к уже уничтоженной переменной. Точно так же происходит в обычном однопоточном коде — сохранять указатель или ссылку на локальную переменную после выхода из функции всегда плохо, — но в многопоточном коде такую ошибку сделать проще, потому что не сразу видно, что произошло.

Один из распространенных способов разрешить такую ситуацию — сделать функцию потока замкнутой, то есть *копировать* в поток данные, а не разделять их. Если функция потока реализовала в виде вызываемого объекта, то сам этот объект копируется в поток, поэтому исходный объект можно сразу же уничтожить. Однако по-прежнему необходимо следить за тем, чтобы объект не содержал ссылок или указателей, как в листинге 2.1. В частности, не стоит создавать внутри функции поток, имеющий доступ к локальным переменным этой функции, если нет гарантии, что поток завершится до выхода из функции.

Есть и другой способ — явно гарантировать, что поток завершит исполнение до выхода из функции, *присоединившись* к нему.

2.1.2. Ожидание завершения потока

Чтобы дождаться завершения потока, следует вызвать функцию `join()` ассоциированного объекта `std::thread`. В листинге 2.1 мы можем заменить вызов `my_thread.detach()` перед закрывающей скобкой тела функции вызовом `my_thread.join()`, и тем самым гарантировать, что поток завершится до выхода из функции, то есть раньше, чем будут уничтожены локальные переменные. В данном случае это означает, что запускать функцию в отдельном потоке не имело смысла, так как первый поток в это время ничего не делает, по в реальной программе исходный поток мог бы либо сам делать что-то полезное,

либо запустить несколько потоков параллельно, а потом дождаться их всех.

Функция `join()` дает очень простую и прямолинейную альтернативу — либо мы ждем завершения потока, либо нет. Если необходим более точный контроль над ожиданием потока, например если необходимо проверить, завершился ли поток, или ждать только ограниченное время, то следует прибегнуть к другим механизмам, таким, как условные переменные и будущие результаты, которые мы будем рассматривать в главе 4. Кроме тот, при вызове `join()` очищается вся ассоциированная с потоком память, так что объект `std::thread` более не связан с завершившимся потоком — он вообще не связан ни с каким потоком. Это значит, что для каждого потока вызвать функцию `join()` можно только один раз; после первого вызова объект `std::thread` уже не допускает присоединения, и функция `joinable()` возвращает `false`.

2.1.3. Ожидание в случае исключения

Выше уже отмечалось, что функцию `join()` или `detach()` необходимо вызвать до уничтожения объекта `std::thread`. Если вы хотите отсоединить поток, то обычно достаточно вызвать `detach()` сразу после его запуска, так что здесь проблемы не возникает. Но если вы собираетесь дождаться завершения потока, то надо тщательно выбирать место, куда поместить вызов `join()`. Важно, чтобы из-за исключения, произошедшего между запуском потока и вызовом `join()`, не оказалось, что обращение к `join()` вообще окажется пропущенным.

Чтобы приложение не завершилось аварийно при возникновении исключения, необходимо решить, что делать в этом случае. Вообще говоря, если вы намеревались вызвать функцию `join()` при нормальном выполнении программы, то следует вызывать ее и в случае исключения, чтобы избежать проблем, связанных с временем жизни. В листинге 2.2 приведен простой способ решения этой задачи.

Листинг 2.2. Ожидание завершения потока

```
struct func; ←┐ см. определение
              | в листинге 2.1
void f() {
 int some_local_state = 0;
 func my_func(some_local_state)
 std::thread t(my_func);
 try {
  do_something_in_current_thread()
 }
 catch(...) {
  t.join(); ← (1)
  throw;
 }
 t.join();  ← (2)
}
```

В листинге 2.2 блок `try/catch` используется для того, чтобы поток, имеющий доступ к локальному состоянию, гарантированно завершился до выхода из функции вне зависимости оттого, происходит выход нормально **(2)** или вследствие исключения **(1)**. Записывать блоки `try/catch` очень долго и при этом легко допустить ошибку, поэтому такой способ не идеален.

Если необходимо гарантировать, что поток завершается до выхода из функции потому ли, что он хранит ссылки на локальные переменные, или по какой-то иной причине то важно обеспечить это на всех возможных путях выхода, как нормальных, так и в результате исключения, и хотелось бы иметь для этого простой и лаконичный механизм.

Один из способов решить эту задачу воспользоваться стандартной идиомой *захват ресурса есть инициализация* (RAII) и написать класс, который вызывает `join()` в деструкторе, например, такой, как в листинге 2.3. Обратите внимание, насколько проще стала функция `f()`.

Листинг 2.3. Использование идиомы RAII для ожидания завершения потока

```
class thread_guard {
 std::threads t;
public:
 explicit thread_guard(std::thread& t_) : t(t_) {}
 ~thread_guard() {
  if (t.joinable()) ← (1)
  {
   t.join();        ← (2)
  }
 }
 thread_guard(thread_guard const&)=delete; ← (3)
 thread_guard& operator=(thread_guard const&)=delete;
};


struct func; ←┐ см.определение
              | в листинге 2.1
void f() {
 int some_local_state;
 std::thread t(func(some_local_state));
 thread_guard g(t);
 do_something_in_current_thread();
}             ← (4)
```

Когда текущий поток доходит до конца `f` **(4)**, локальные объекты уничтожаются в порядке, обратном тому, в котором были сконструированы. Следовательно, сначала уничтожается объект `g` типа `thread_guard`, и в его деструкторе **(2)** происходит присоединение к потоку Это справедливо даже в том случае, когда выход из функции `f` произошел в результате исключения внутри функции `do_something_in_current_thread`.

Деструктор класса `thread_guard` в листинге 2.3 сначала проверяет, что объект `std::thread` находится в состоянии `joinable()` **(1)** и, лишь если это так, вызывает `join()` **(2)**. Это существенно, потому что функцию `join()` можно вызывать только один раз для данного потока, так что если он уже присоединился, то делать это вторично было бы ошибкой.

Копирующий конструктор и копирующий оператор присваивания помечены признаком `=delete` **(3)**, чтобы компилятор не генерировал их автоматически: копирование или присваивание такого объекта таит в себе опасность, поскольку время жизни копии может оказаться дольше, чем время жизни присоединяемого потока. Но раз эти функции объявлены как «удаленные», то любая попытка скопировать объект типа `thread_guard` приведет к ошибке компиляции. Дополнительные сведения об удаленных функциях см. в приложении А,

раздел A.2.

Если ждать завершения потока не требуется, то от проблемы безопасности относительно исключений можно вообще уйти, отсоединив поток. Тем самым связь потока с объектом `std::thread` разрывается, и при уничтожении объекта `std::thread` функция `std::terminate()` не будет вызвана. Но отсоединенный поток по-прежнему работает — в фоновом режиме.

2.1.4. Запуск потоков в фоновом режиме

Вызов функции-члена `detach()` объекта `std::thread` оставляет поток работать в фоновом режиме, без прямых способов коммуникации с ним. Теперь ждать завершения потока не получится — после того как поток отсоединен, уже невозможно получить ссылающийся на него объект `std::thread`, для которого можно было бы вызвать `join()`. Отсоединенные потоки действительно работают в фоне: отныне ими владеет и управляет библиотека времени выполнения C++, которая обеспечит корректное освобождение связанных с потоком ресурсов при его завершении.

Отсоединенные потоки часто называют *потоками-демонами* по аналогии с *процессами-демонами* в UNIX, то есть с процессами, работающими в фоновом режиме и не имеющими явного интерфейса с пользователем. Обычно такие потоки работают в течение длительного времени, в том числе на протяжении всего времени жизни приложения. Они, например, могут следить за состоянием файловой системы, удалять неиспользуемые записи из кэша или оптимизировать структуры данных. С другой стороны, иногда отсоединенный поток применяется, когда существует какой-то другой способ узнать о его завершении или в случае, когда нужно запустить задачу и «забыть» о ней.

В разделе 2.1.2 мы уже видели, что для отсоединения потока следует вызвать функцию-член `detach()` объекта `std::thread`. После возврата из этой функции объект `std::thread` уже не связан ни с каким потоком, и потому присоединиться к нему невозможно.

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

Разумеется, чтобы отсоединить поток от объекта `std::thread`, поток должен существовать: нельзя вызвать `detach()` для объекта `std::thread`, с которым не связан никакой поток. Это то же самое требование, которое предъявляется к функции `join()`, поэтому и проверяется оно точно так же — вызывать `t.detach()` для объекта `t` типа `std::thread` можно только тогда, когда `t.joinable()` возвращает `true`.

Возьмем в качестве примера текстовый редактор, который умеет редактировать сразу несколько документов. Реализовать его можно разными способами — как на уровне пользовательского интерфейса, так и с точки зрения внутренней организации. В настоящее время все чаще для этой цели используют несколько окон верхнего уровня, по одному для каждого редактируемого документа. Хотя эти окна выглядят совершенно независимыми, в частности, у каждого есть свое меню и все прочее, на самом деле они существуют внутри единственного экземпляра приложения. Один из подходов к внутренней организации программы заключается в том, чтобы запускать каждое окно в отдельном потоке: каждый такой поток исполняет один и тот же код, но с разными данными, описывающими редактируемый документ и соответствующее ему окно. Таким образом, чтобы открыть еще

один документ, необходимо создать новый поток. Потоку, обрабатывающему запрос, нет дела до того, когда созданный им поток завершится, потому что он работает над другим, независимым документом. Вот типичная ситуация, когда имеет смысл запускать отсоединенный поток.

В листинге 2.4 приведен набросок кода, реализующего этот подход.

Листинг 2.4. Отсоединение потока для обработки другого документа

```
void edit_document(std::string const& filename) {
 open_document_and_display_gui(filename);
 while(!done_editing()) {
  user_command cmd = get_user_input();
  if (cmd.type == open_new_document) {
   std::string const new_name = get_filename_from_user();
   std::thread t(edit_document,new_name); ← (1)
   t.detach(); ← (2)
  }
  else {
   process_user_input(cmd);
  }
 }
}
```

Когда пользователь открывает новый документ, мы спрашиваем, какой документ открыть, затем запускаем поток, в котором этот документ открывается **(1)**, и отсоединяем его **(2)**. Поскольку новый поток делает то же самое, что текущий, только с другим файлом, то мы можем использовать ту же функцию (`edit_document`), передав ей в качестве аргумента имя только что выбранного файла.

Этот пример демонстрирует также, почему бывает полезно передавать аргументы функции потока: мы передаем конструктору объекта `std::thread` не только имя функции **(1)**, но и её параметр — имя файла. Существуют другие способы добиться той же цели, например, использовать не обычную функцию с параметрами, а объект-функцию с данными-членами, но библиотека предлагает и такой простой механизм.

2.2. Передача аргументов функции потока

Из листинга 2.4 видно, что по существу передача аргументов вызываемому объекту или функции сводится просто к передаче дополнительных аргументов конструктору `std::thread`. Однако важно иметь в виду, что по умолчанию эти аргументы *копируются* в память объекта, где они доступны вновь созданному потоку, причем так происходит даже в том случае, когда функция ожидает на месте соответствующего параметра ссылку. Вот простой пример:

```
void f(int i, std::string const& s);
std::thread t(f, 3, "hello");
```

Здесь создается новый ассоциированный с объектом `t` поток, в котором вызывается функция `f(3, "hello")`. Отметим, что функция `f` принимает в качестве второго параметра объект типа `std::string`, но мы передаем строковый литерал `char const*`, который преобразуется к типу `std::string` уже в контексте нового потока. Это особенно важно, когда переданный аргумент является указателем на автоматическую переменную, как в примере ниже:

```
void f(int i, std::string const& s);

void oops(int some_param) {
 char buffer[1024];            ← (1)
 sprintf(buffer, "%i", some_param);
 std::thread t(f, 3, buffer); ← (2)
 t.detach();
}
```

В данном случае в новый поток передается **(2)** указатель на локальную переменную `buffer` **(1)**, и есть все шансы, что выход из функции oops произойдет раньше, чем буфер будет преобразован к типу `std::string` в новом потоке. В таком случае мы получим неопределенное поведение. Решение заключается в том, чтобы выполнить преобразование в `std::string` *до* передачи `buffer` конструктору `std::thread`:

```
void f(int i,std::string const& s);

void not_oops(int some_param) {
 char buffer[1024];
 sprintf(buffer, "%i", some_param);
 std::thread t(f, 3, std::string(buffer)); ←┘
 t.detach();
}
```

| Использование std::string позволяет избежать висячего указателя

В данном случае проблема была в том, что мы положились на неявное преобразование указателя на `buffer` к ожидаемому типу первого параметра `std::string`, а конструктор `std::thread` копирует переданные значения «как есть», без преобразования к ожидаемому типу аргумента.

Возможен и обратный сценарий: копируется весь объект, а вы хотели бы получить ссылку Такое бывает, когда поток обновляет структуру данных, переданную по ссылке, например:

```
void update_data_for_widget(widget_id w,widget_data& data); ← (1)
```

```
void oops_again(widget_id w) {
 widget_data data;
 std::thread t(update_data_for_widget, w, data); ← (2)
 display_status();
 t.join();
 process_widget_data(data);                      ← (3)
}
```

Здесь `update_data_for_widget` **(1)** ожидает, что второй параметр будет передан по ссылке, но конструктор `std::thread` **(2)** не знает об этом: он не в курсе того, каковы типы аргументов, ожидаемых функцией, и просто слепо копирует переданные значения. Поэтому функции `update_data_for_widget` будет передана ссылка на внутреннюю копию `data`, а не на сам объект `data`. Следовательно, по завершении потока от обновлений ничего не останется, так как внутренние копии переданных аргументов уничтожаются, и функция `process_widget_data` получит не обновленные данные, а исходный объект `data` **(3)**. Для читателя, знакомого с механизмом `std::bind`, решение очевидно: нужно обернуть аргументы, которые должны быть ссылками, объектом `std::ref`. В данном случае, если мы напишем

```
std::thread t(update_data_for_widget, w, std::ref(data));
```

то функции `update_data_for_widget` будет правильно передана ссылка на `data`, а не *копия* data.

Если вы знакомы с `std::bind`, то семантика передачи параметров вряд ли вызовет удивление, потому что работа конструктора `std::thread` и функции `std::bind` определяется в терминах одного и того же механизма. Это, в частности, означает, что в качестве функции можно передавать указатель на функцию-член при условии, что в первом аргументе передается указатель на правильный объект:

```
class X {
public:
 void do_lengthy_work();
};

X my_x;
std::thread t(&X::do_lengthy_work, &my_x); ← (1)
```

Здесь мы вызываем `my_x.do_lengthy_work()` в новом потоке, поскольку в качестве указателя на объект передан адрес `my_x` **(1)**. Так вызванной функции-члену можно передавать и аргументы: третий аргумент конструктора `std::thread` станет первым аргументом функции-члена и т.д.

Еще один интересный сценарий возникает, когда передаваемые аргументы нельзя копировать, а можно только *перемещать*: данные, хранившиеся в одном объекте, переносятся в другой, а исходный объект остается «пустым». Примером может служить класс `std::unique_ptr`, который обеспечивает автоматическое управление памятью для динамически выделенных объектов. В каждый момент времени на данный объект может указывать только один экземпляр `std::unique_ptr`, и, когда этот экземпляр уничтожается, объект, на который он указывает, удаляется. *Перемещающий конструктор* и *перемещающий оператор присваивания* позволяют передавать владение объектом от одного экземпляра `std::unique_ptr` другому (о семантике перемещения см. приложение А, раздел А.1.1). После такой передачи в исходном экземпляре остается указатель NULL. Подобное перемещение значений дает возможность передавать такие объекты в качестве параметров

функций или возвращать из функций. Если исходный объект временный, то перемещение производится автоматически, а если это именованное значение, то передачу владения следует запрашивать явно, вызывая функцию `std::move()`. В примере ниже показано применение функции `std::move` для передачи владения динамическим объектом потоку:

```
void process_big_object(std::unique_ptr<big_object>);

std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

Поскольку мы указали при вызове конструктора `std::thread` функцию `std::move`, то владение объектом `big_object` передается объекту во внутренней памяти вновь созданного потока, а затем функции `process_big_object`.

В стандартной библиотеке Thread Library есть несколько классов с такой же семантикой владения, как у `std::unique_ptr`, и `std::thread` — один из них. Правда, экземпляры `std::thread` не владеют динамическими объектами, как `std::unique_ptr`, зато они владеют ресурсами: каждый экземпляр отвечает за управление потоком выполнения. Это владение можно передавать от одного экземпляра другому, поскольку экземпляры `std::thread` *перемещаемые*, хотя и не *копируемые*. Тем самым гарантируется, что в каждый момент времени с данным потоком будет связан только один объект, но в то же время программист вправе передавать владение от одного объекта другому

2.3. Передача владения потоком

Предположим, что требуется написать функцию для создания потока, который должен работать в фоновом режиме, но при этом мы не хотим ждать его завершения, а хотим, чтобы владение новым потоком было передано вызывающей функции. Или требуется сделать обратное — создать поток и передать владение им некоторой функции, которая будет ждать его завершения. В обоих случаях требуется передать владение из одного места в другое.

Именно здесь и оказывается полезной поддержка классом `std::thread` семантики перемещения. В предыдущем разделе отмечалось, что в стандартной библиотеке C++ есть много типов, владеющих ресурсами, например `std::ifstream` и `std::unique_ptr`, которые являются *перемещаемыми*, но не копируемыми, и один из них — `std::thread`. Это означает, что владение потоком можно передавать от одного экземпляра `std::thread` другому, как показано в примере ниже. В нем создается два потока выполнения, владение которыми передается между тремя объектами `std::thread`: `t1`, `t2` и `t3`.

```
void some_function();
void some_other_function();

std::thread t1(some_function);          ← (1)
std::thread t2 = std::move(t1);         ← (2)
t1 = std::thread(some_other_function); ← (3)
std::thread t3;      ← (4)
t3 = std::move(t2); ← (5)
t1 = std::move(t3); ← (6) Это присваивание приводит
;                              к аварийному завершению программы
```

Сначала создастся новый поток **(1)** и связывается с объектом `t1`. Затем владение явно передается объекту `t2` в момент его конструирования путем вызова `std::move()` **(2)**. В этот момент с `t1` уже не связан никакой поток выполнения: поток, в котором исполняется функция `some_function`, теперь связан с `t2`.

Далее создается еще один поток, который связывается с временным объектом типа `std::thread` **(3)**. Для последующей передачи владения объекту `t1` уже не требуется явный вызов `std::move()`, так как владельцем является временный объект, а передача владения от временных объектов производится автоматически и неявно.

Объект `t3` конструируется по умолчанию **(4)**, а это означает, что в момент создания с ним не связывается никакой поток. Владение потоком, который в данный момент связан с `t2`, передастся объекту `t3` **(5)**, опять-таки путем явного обращения к `std::move()`, поскольку `t2` — именованный объект. После всех этих перемещений `t1` оказывается связан с потоком, исполняющим функцию `some_other_function`, `t2` не связан ни с каким потоком, а `t3` связан с потоком, исполняющим функцию `some_function` .

Последнее перемещение **(6)** передает владение потоком, исполняющим `some_function`, обратно объекту `t1`, в котором исполнение этой функции началось. Однако теперь с `t1` уже связан поток (который исполнял функцию `some_other_function`), поэтому вызывается `std::terminate()`, и программа завершается. Так делается ради совместимости с поведением деструктора `std::thread`. В разделе 2.1.1 мы видели, что нужно либо явно ждать завершения потока, либо отсоединить его до момента уничтожения; то же самое

относится и к присваиванию: нельзя просто «прихлопнуть» поток, присвоив новое значение объекту `std::thread`, который им управляет.

Поддержка операции перемещения в классе `std::thread` означает, что владение можно легко передать при возврате из функции, как показано в листинге 2.5.

Листинг 2.5. Возврат объекта `std::thread` из функции

```
std::thread f() {
 void some_function();
 return std::thread(some_function);
}


std::thread g() {
 void some_other_function(int);
 std::thread t(some_other_function, 42);
 return t;
}
```

Аналогично, если требуется передать владение внутрь функции, то достаточно, чтобы она принимала экземпляр `std::thread` по значению в качестве одного из параметров, например:

```
void f(std::thread t);


void g() {
 void some_function();
 f(std::thread(some_function));
 std::thread t(some_function);
 f(std::move(t));
}
```

Одно из преимуществ, которые даёт поддержка перемещения в классе `std::thread`, заключается в том, что мы можем модифицировать класс `thread_guard` из листинга 2.3, так чтобы он принимал владение потоком. Это позволит избежать неприятностей в случае, когда время жизни объекта `thread_guard` оказывает больше, чем время жизни потока, на который он ссылается, а, кроме того, это означает, что никто другой не сможет присоединиться к потоку или отсоединить его, так как владение было передано объекту `thread_guard`. Поскольку основное назначение этого класса гарантировать завершение потока до выхода из области видимости, я назвал его `scoped_thread`. Реализация и простой пример использования приведены в листинге 2.6.

Листинг 2.6. Класс `scoped_thread` и пример его использования

```
class scoped_thread {
 std::thread t;
public:
 explicit scoped_thread(std::thread t_) : ← (1)
 t(std::move(t_)) {
 if (!t.joinable()) ← (2)
  throw std::logic_error("No thread");
 }
 ~scoped_thread() {
  t.join();          ← (3)
 }
 scoped_thread(scoped_thread const&)=delete;
```

```
 scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func; ← см. листинг 2.1

void f() {
 int some_local_state;
 scoped_thread t(std::thread(func(some_local_state))); ← (4)
 do_something_in_current_thread();
}                                                    ← (5)
```

Этот пример очень похож на приведенный в листинге 2.3, только новый поток теперь передается непосредственно конструктору `scoped_thread` **(4)**, вместо того чтобы создавать для него отдельную именованную переменную. Когда новый поток достигает конца `f` **(5)**, объект `scoped_thread` уничтожается, а затем поток соединяется **(3)** с потоком, переданным конструктору **(1)**. Если в классе `thread_guard` из листинга 2.3 деструктор должен был проверить, верно ли, что поток все еще допускает соединение, то теперь мы можем сделать это в конструкторе **(2)** и возбудить исключение, если это не так.

Поддержка перемещения в классе `std::thread` позволяет также хранить объекты этого класса в контейнере при условии, что класс контейнера поддерживает перемещение (как, например, модифицированный класс `std::vector<>`). Это означает, что можно написать код, показанный в листинге 2.7, который запускает несколько потоков, а потом ждет их завершения.

Листинг 2.7. Запуск нескольких потоков и ожидание их завершения

```
void do_work(unsigned id);

void f() {
 std::vector<std::thread> threads;
 for (unsigned i = 0; i < 20; ++i) {                  | Запуск
  threads.push_back(std::thread(do_work(i))); ←┘ потоков
 }                                                    | Поочередный
 std::for_each(threads.begin(), threads.end(),        | вызов join()
 std::mem_fn(&std::thread::join));                ←┘ для каждого потока
}
```

Если потоки применяются для разбиения алгоритма на части, то зачастую такой подход именно то, что требуется: перед возвратом управления вызывающей программе все потоки должны завершиться. Разумеется, столь простая структура, как в листинге 2.7, предполагает, что каждый поток выполняет независимую работу, а единственным результатом является побочный эффект, заключающийся в изменении разделяемых данных. Если бы функция `f()` должна была вернуть вызывающей программе значение, зависящее от результатов операций, выполненных в потоках, то при такой организации получить это значение можно было бы только путем анализа разделяемых данных по завершении всех потоков. В главе 4 обсуждаются альтернативные схемы передачи результатов работы из одного потока в другой.

Хранение объектов `std::thread` в векторе `std::vector` — шаг к автоматизации управления потоками: вместо тот чтобы создавать отдельные переменные для потоков и выполнять соединение напрямую, мы можем рассматривать группу потоков. Можно пойти

еще дальше и создавать не фиксированное число потоков, как в листинге 2.7, а определять нужное количество динамически, во время выполнения.

2.4. Задание количества потоков во время выполнения

В стандартной библиотеке C++ есть функция `std::thread::hardware_concurrency()`, которая поможет нам решить эту задачу. Она возвращает число потоков, которые могут работать по-настоящему параллельно. В многоядерной системе это может быть, например, количество процессорных ядер. Возвращаемое значение всего лишь оценка; более того, функция может возвращать 0, если получить требуемую информацию невозможно. Однако эту оценку можно с пользой применить для разбиения задачи на несколько потоков.

В листинге 2.8 приведена простая реализация параллельной версии `std::accumulate`. Она распределяет работу между несколькими потоками и, чтобы не создавать слишком много потоков, задает ограничение снизу на количество элементов, обрабатываемых одним потоком. Отмстим, что в этой реализации предполагается, что ни одна операция не возбуждает исключений, хотя в принципе исключения возможны; например, конструктор `std::thread` возбуждает исключение, если не может создать новый поток. Но если добавить в этот алгоритм обработку исключений, он перестанет быть таким простым; эту тему мы рассмотрим в главе 8.

Листинг 2.8. Наивная реализация параллельной версии алгоритма `std::accumulate`

```
template<typename Iterator, typename T>
 struct accumulate_block {
 void operator()(Iterator first, Iterator last, T& result) {
  result = std::accumulate(first, last, result);
 }
};

template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
 unsigned long const length = std::distance(first, last);
 if (!length) ← (1)
  return init;

 unsigned long const min_per_thread = 25;
 unsigned long const max_threads =
  (length+min_per_thread - 1) / min_per_thread; ← (2)

 unsigned long const hardware_threads =
  std::thread::hardware_concurrency();

 unsigned long const num_threads = ← (3)
  std::min(
   hardware.threads != 0 ? hardware_threads : 2, max_threads);

 unsigned long const block_size = length / num_threads; ← (4)

 std::vector<T> results(num_threads);
 std::vector<std::thread> threads(num_threads - 1); ← (5)

 Iterator block_start = first;
```

```
    for(unsigned long i = 0; i < (num_threads - 1); ++i) {
     Iterator block_end = block_start;
     std::advance(block_end, block_size); ← (6)

     threads[i] = std::thread( ← (7)
      accumulate_block<Iterator, T>(),
      block_start, block_end, std::ref(results(i)));
     block_start = block_end;  ← (8)
    }
    accumulate_block()(
     block_start, last, results[num_threads-1]); ← (9)

    std::for_each(threads.begin(), threads.end(),
    std::mem_fn(&std::thread::join)); ← (10)

    return
     std::accumulate(results.begin(), results.end(), init); ← (11)
}
```

Хотя функция довольно длинная, по существу она очень проста. Если входной диапазон пуст **(1)**, то мы сразу возвращаем начальное значение `init`. В противном случае диапазон содержит хотя бы один элемент, поэтому мы можем разделить количество элементов на минимальный размер блока и получить максимальное число потоков **(2)**.

Это позволит избежать создания 32 потоков на 32-ядерной машине, если диапазон состоит всего из пяти элементов.

Число запускаемых потоков равно минимуму из только что вычисленного максимума и количества аппаратных потоков **(3)**: мы не хотим запускать больше потоков, чем может поддержать оборудование (это называется *превышением лимита*), так как из-за контекстных переключений при большем количестве потоков производительность снизится. Если функция `std::thread::hardware_concurrency()` вернула 0, то мы берем произвольно выбранное число, я решил остановиться на 2. Мы не хотим запускать слишком много потоков, потому что на одноядерной машине это только замедлило бы программу. Но и слишком мало потоков тоже плохо, так как это означало бы отказ от возможного параллелизма.

Каждый поток будет обрабатывать количество элементов, равное длине диапазона, поделенной на число потоков **(4)**. Пусть вас не пугает случай, когда одно число нацело не делится на другое, — ниже мы рассмотрим его.

Теперь, зная, сколько необходимо потоков, мы можем создать вектор `std::vector<T>` для хранения промежуточных результатов и вектор `std::vector<std::thread>` для хранения потоков **(5)**. Отметим, что запускать нужно на один поток меньше, чем `num_threads`, потому что один поток у нас уже есть.

Запуск потоков производится в обычном цикле: мы сдвигаем итератор `block_end` в конец текущего блока **(6)** и запускаем новый поток для аккумулирования результатов по этому блоку **(7)**. Начало нового блока совпадает с концом текущего **(8)**.

После того как все потоки запущены, главный поток может обработать последний блок **(9)**. Именно здесь обрабатывается случай деления с остатком: мы знаем, что конец последнего блока — `last`, а сколько в нем элементов, не имеет значения.

Аккумулировав результаты но последнему блоку, мы можем дождаться завершения всех запущенных потоков с помощью алгоритма `std::for_each` **(10)**, а затем сложить частичные результаты, обратившись к `std::accumulate` **(11)**.

Прежде чем расстаться с этим примером, полезно отметить, что в случае, когда оператор сложения, определенный в типе `T`, не ассоциативен (например, если `T` — это `float` или `double`), результаты, возвращаемые алгоритмами `parallel_accumulate` и `std::accumulate`, могут различаться из-за разбиения диапазона на блоки. Кроме того, к итераторам предъявляются более жесткие требования: они должны быть по меньшей мере *однонаправленными*, тогда как алгоритм `std::accumulate` может работать и с однопроходными *итераторами ввода*. Наконец, тип `T` должен допускать конструирование по умолчанию (удовлетворять требованиям концепции *DefaultConstructible*), чтобы можно было создать вектор `results`. Такого рода изменения требований довольно типичны для параллельных алгоритмов: но самой своей природе они отличаются от последовательных алгоритмов, и это приводит к определенным последствиям в части как результатов, так и требований. Более подробно параллельные алгоритмы рассматриваются в главе 8. Стоит также отметить, что из-за невозможности вернуть значение непосредственно из потока, мы должны передавать ссылку на соответствующий элемент вектора `results`. Другой способ возврата значений из потоков, *с помощью будущих результатов*, рассматривается в главе 4.

В данном случае вся необходимая потоку информация передавалась в момент его запуска — в том числе и адрес, но которому необходимо сохранить результат вычисления. Так бывает не всегда; иногда требуется каким-то образом идентифицировать потоки во время работы. Конечно, можно было бы передать какой-то идентификатор, например значение `i` в листинге 2.7, но если вызов функции, которой этот идентификатор нужен, находится несколькими уровнями стека глубже, и эта функция может вызываться из любого потока, то поступать так неудобно. Проектируя библиотеку C++ Thread Library, мы предвидели этот случай, поэтому снабдили каждый поток уникальным идентификатором.

2.5. Идентификация потоков

Идентификатор потока имеет тип `std::thread::id`, и получить его можно двумя способами. Во-первых, идентификатор потока, связанного с объектом `std::thread`, возвращает функция-член `get_id()` этого объекта. Если с объектом `std::thread` не связан никакой поток, то `get_id()` возвращает сконструированный по умолчанию объект типа `std::thread::id`, что следует интерпретировать как «не поток». Идентификатор текущего потока можно получить также, обратившись к функции `std::this_thread::get_id()`, которая также определена в заголовке `<thread>`.

Объекты типа `std::thread::id` можно без ограничений копировать и сравнивать, в противном случае они вряд ли могли бы играть роль идентификаторов. Если два объекта типа `std::thread::id` равны, то либо они представляют один и тот же поток, либо оба содержат значение «не поток». Если же два таких объекта не равны, то либо они представляют разные потоки, либо один представляет поток, а другой содержит значение «не поток».

Библиотека Thread Library не ограничивается сравнением идентификаторов потоков на равенство, для объектов типа `std::thread::id` определен полный спектр операторов сравнения, то есть на множестве идентификаторов потоков задан полный порядок. Это позволяет использовать их в качестве ключей ассоциативных контейнеров, сортировать и сравнивать любым интересующим программиста способом. Поскольку операторы сравнения определяют полную упорядоченность различных значений типа `std::thread::id`, то их поведение интуитивно очевидно: если `a<b` и `b<c` то `a<c` и так далее. В стандартной библиотеке имеется также класс `std::hash<std::thread::id>`, поэтому значения типа `std::thread::id` можно использовать и в качестве ключей новых неупорядоченных ассоциативных контейнеров.

Объекты `std::thread::id` часто применяются для того, чтобы проверить, должен ли поток выполнить некоторую операцию. Например, если потоки используются для разбиения задач, как в листинге 2.8, то начальный поток, который запускал все остальные, может вести себя несколько иначе, чем прочие. В таком случае этот поток мог бы сохранить значение `std::this_thread::get_id()` перед тем, как запускать другие потоки, а затем в основной части алгоритма (общей для всех потоков) сравнить собственный идентификатор с сохраненным значением.

```
std::thread::id master_thread;

void some_core_part_of_algorithm() {
 if (std::this_thread::get_id() == master_thread) {
  do_master_thread_work();
 }
 do_common_work();
}
```

Или можно было бы сохранить `std::thread::id` текущего потока в некоторой структуре данных в ходе выполнения какой-то операции. В дальнейшем при операциях с той же структурой данных можно было сравнить сохраненный идентификатор с идентификатором потока, выполняющего операцию, и решить, какие операции разрешены или необходимы.

Идентификаторы потоков можно было бы также использовать как ключи ассоциативных контейнеров, если с потоком нужно ассоциировать какие-то данные, а другие механизмы,

например поточно-локальная память, не подходят. Например, управляющий поток мог бы сохранить в таком контейнере информацию о каждом управляемом им потоке. Другое применение подобного контейнера — передавать информацию между потоками.

Идея заключается в том, что в большинстве случаев `std::thread::id` вполне может служить обобщенным идентификатором потока и лишь, если с идентификатором необходимо связать какую-то семантику (например, использовать его как индекс массива), может потребоваться другое решение. Можно даже выводить объект `std::thread::id` в выходной поток, например `std::cout`:

```
std::cout << std::this_thread::get_id();
```

Точный формат вывода зависит от реализации; стандарт лишь гарантирует, что результаты вывода одинаковых идентификаторов потоков будут одинаковы, а разных — различаться. Поэтому для отладки и протоколирования это может быть полезно, но так как никакой семантики у значений идентификаторов нет, то сделать на их основе какие-то другие выводы невозможно.

2.6. Резюме

В этой главе мы рассмотрели основные средства управления потоками, имеющиеся в стандартной библиотеке C++: запуск потоков, ожидание завершения потока и *отказ* от ожидания вследствие того, что поток работает в фоновом режиме. Мы также научились передавать аргументы функции потока при запуске и передавать ответственность за управление потоком из одной части программы в другую. Кроме того, мы видели, как можно использовать группы потоков для разбиения задачи на части. Наконец, мы обсудили механизм идентификации потоков, позволяющий ассоциировать с потоком данные или поведение в тех случаях, когда использовать другие средства неудобно. Даже совершенно независимые потоки позволяют сделать много полезного, как видно из листинга 2.8, но часто требуется, чтобы работающие потоки обращались к каким-то общим данным. В главе 3 рассматриваются проблемы, возникающие при разделении данных между потоками, а в главе 4 — более общие вопросы синхронизации операций с использованием и без использования разделяемых данных.

Глава 3.
Разделение данных между потоками

В этой главе:

- Проблемы разделения данных между потоками.
- Защита данных с помощью мьютексов.
- Альтернативные средства защиты разделяемых данных.

Одно из основных достоинств применения потоков для реализации параллелизма — возможность легко и беспрепятственно разделять между ними данные, поэтому, уже зная, как создавать потоки и управлять ими, мы обратимся к вопросам, связанным с разделением данных.

Представьте, что вы живете в одной квартире с приятелем. В квартире только одна кухня и только одна ванная. Если ваши отношения не особенно близки, то вряд ли вы будете пользоваться ванной одновременно, поэтому, когда сосед слишком долго занимает ванную, у вас возникает законное недовольство. Готовить два блюда одновременно, конечно, можно, но если у вас духовка совмещена с грилем, то ничего хорошего не выйдет, когда один пытается жарить сосиски, а другой — печь пирожные. Ну и все мы знаем, какую досаду испытываешь, когда, сделав половину работы, обнаруживаешь, что кто-то забрал нужный инструмент или изменил то, что вы уже сделали.

То же самое и с потоками. Если потоки разделяют какие-то данные, то необходимы правила, регулирующие, какой поток в какой момент к каким данным может обращаться и как сообщить об изменениях другим потокам, использующим те же данные. Легкость, с которой можно разделять данные между потоками в одном процессе, может обернуться не только благословением, но проклятием. Некорректное использование разделяемых данных — одна из основных причин ошибок, связанных с параллелизмом, и последствия могут оказаться куда серьезнее, чем пропахшие сосисками пирожные.

Эта глава посвящена вопросу о том, как безопасно разделять данные между потоками в программе на C++, чтобы избежать возможных проблем и достичь оптимального результата.

3.1. Проблемы разделения данных между потоками

Все проблемы разделения данных между потоками связаны с последствиями модификации данных. *Если разделяемые данные только читаются, то никаких сложностей не возникает, поскольку любой поток может читать данные независимо от того, читают их в то же самое время другие потоки или нет.* Но стоит одному или нескольким потокам начать модифицировать разделяемые данные, как могут возникнуть неприятности. В таком случае ответственность за правильную работу ложится на программиста.

При рассуждениях о поведении программы часто помогает понятие *инварианта* — утверждения о структуре данных, которое всегда должно быть истинным, например, «значение этой переменной равно числу элементов в списке». В процессе обновления инварианты часто нарушаются, особенно если структура данных сложна или обновление затрагивает несколько значений.

Рассмотрим двусвязный список, в котором каждый узел содержит указатели на следующий и предыдущий узел. Один из инвариантов формулируется так: если «указатель на следующий» в узле A указывает на узел B, то «указатель на предыдущий» в узле B указывает на узел A. Чтобы удалить узел из списка, необходимо обновить узлы по обе стороны от него, так чтобы они указывали друг на друга. После обновления одного узла инвариант оказывается нарушен и остается таковым, пока не будет обновлен узел по другую сторону. После того как обновление завершено, инвариант снова выполняется.

Шаги удаления узла из списка показаны на рис. 3.1:

1. Найти подлежащий удалению узел (*N*).
2. Изменить «указатель на следующий» в узле, предшествующем *N*, так чтобы он указывал на узел, следующий за *N*.
3. Изменить «указатель на предыдущий» в узле, следующем за *N*, так чтобы он указывал на узел, предшествующий *N*.
4. Удалить узел *N*.

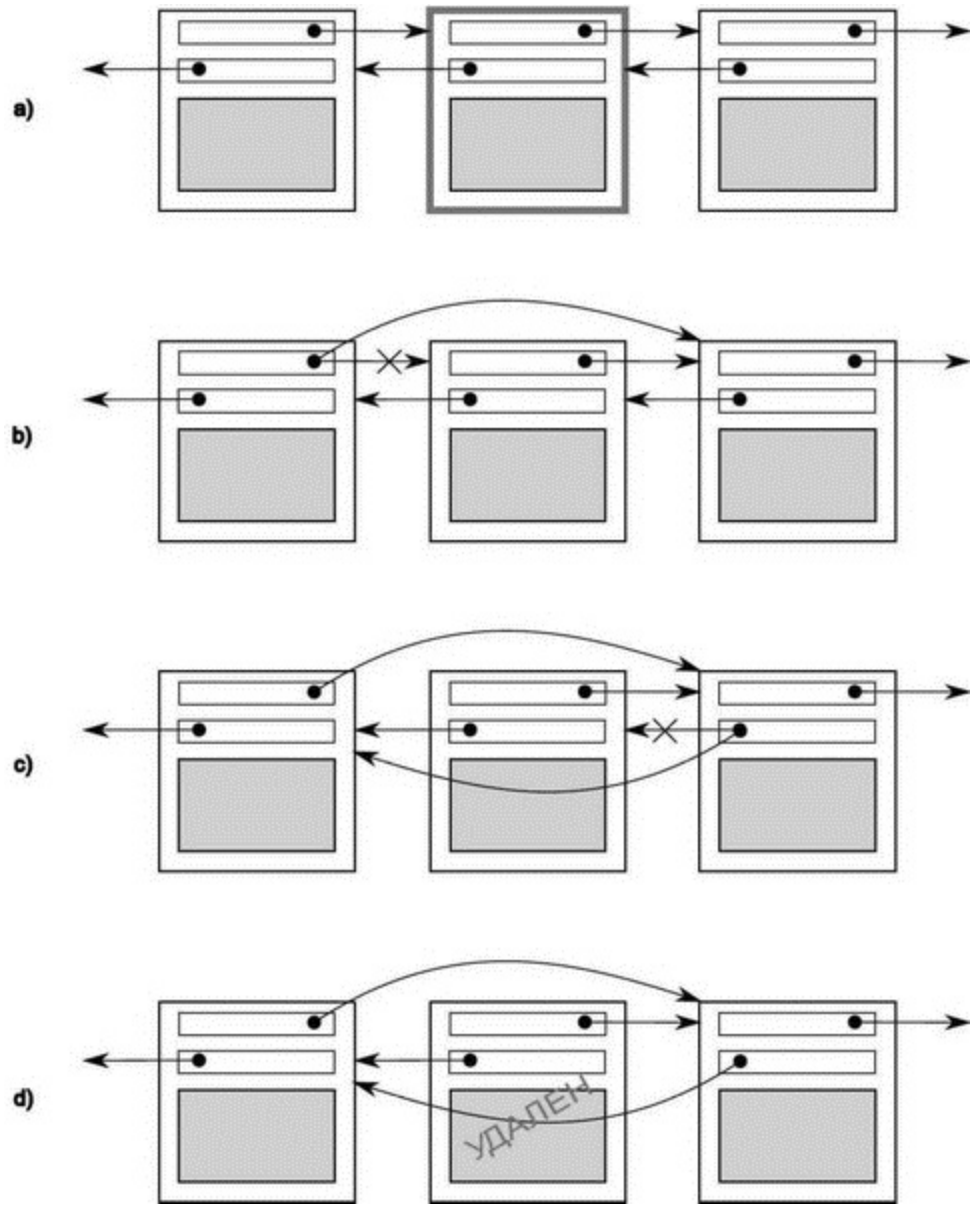

Рис. 3.1. Удаление узла из двусвязного списка

Как видите, между шагами **b** и **c** указатели в одном направлении не согласуются с указателями в другом направлении, и инвариант нарушается.

Простейшая проблема, которая может возникнуть при модификации данных, разделяемых несколькими потоками, — нарушение инварианта. Если не предпринимать никаких мер, то в случае, когда один поток читает двусвязный список, а другой в это же время удаляет из списка узел, вполне может случиться, что читающий поток увидит список, из которого узел удален лишь частично (потому что изменен только один указатель, как на шаге **b** на рис. 3.1), так что инвариант нарушен. Последствия могут быть разными — если поток читает список слева направо, то он просто пропустит удаляемый узел. Но если другой поток пытается удалить самый правый узел, показанный на рисунке, то он может навсегда повредить структуру данных, и в конце концов это приведет к аварийному завершению программы. Как бы то ни было, этот пример иллюстрирует одну из наиболее распространенных причин ошибок в параллельном коде: *состояние гонки* (race condition).

3.1.1. Гонки

Предположим, вы покупаете билеты в кино. Если кинотеатр большой, то в нем может

быть несколько касс, так что в каждый момент времени билеты могут покупать несколько человек. Если кто-то покупает билет на тот же фильм, что и вы, но в другой кассе, то какие места вам достанутся, зависит от того, кто был первым. Если осталось всего несколько мест, то разница может оказаться решающей: за последние билеты возникает гонка в самом буквальном смысле. Это и есть пример *состояния гонки*: какие места вам достанутся (да и достанутся ли вообще), зависит от относительного порядка двух покупок.

В параллельном программировании под состоянием гонки понимается любая ситуация, исход которой зависит от относительного порядка выполнения операций в двух или более потоках — потоки конкурируют за право выполнить операции первыми. Как правило, ничего плохого в этом нет, потому что все исходы приемлемы, даже если их взаимный порядок может меняться. Например, если два потока добавляют элементы в очередь для обработки, то вообще говоря неважно, какой элемент будет добавлен первым, лишь бы не нарушались инварианты системы. Проблема возникает, когда гонка приводит к нарушению инвариантов, как в приведенном выше примере удаления из двусвязного списка. В контексте параллельного программирования *состоянием гонки* обычно называют именно такую *проблематичную* гонку — безобидные гонки не так интересны и к ошибкам не приводят. В стандарте C++ определен также термин *гонка за данными* (data race), означающий ситуацию, когда гонка возникает из-за одновременной модификации одного объекта (детали см. в разделе 5.1.2); гонки за данными приводят к внушающему ужас *неопределенному поведению*.

Проблематичные состояния гонки обычно возникают, когда для завершения операции необходимо модифицировать два или более элементов данных, например два связующих указателя в примере выше. Поскольку элементов несколько, то их модификация производится разными командами, и может случиться, что другой поток обратится к структуре данных в момент, когда завершилась только одна команда. Зачастую состояние гонки очень трудно обнаружить и воспроизвести, поскольку она происходит в очень коротком интервале времени, — если модификации производятся последовательными командами процессора, то вероятность возникновения проблемы при конкретном прогоне очень мала, даже если к структуре данных одновременно обращается другой поток. По мере увеличения нагрузки на систему и количества выполнений операции вероятность проблематичной последовательности выполнения возрастает. И, разумеется, почти всегда такие ошибки проявляются в самый неподходящий момент. Поскольку состояние гонки так чувствительно ко времени, оно может вообще не возникнуть при запуске приложения под отладчиком, так как отладчик влияет на хронометраж программ, пусть и незначительно.

При написании многопоточных программ гонки могут изрядно отравить жизнь — своей сложностью параллельные программы в немалой степени обязаны стараниям избежать проблематичных гонок.

3.1.2. Устранение проблематичных состояний гонки

Существует несколько способов борьбы с проблематичными гонками. Простейший из них - снабдить структуру данных неким защитным механизмом, который гарантирует, что только поток, выполняющий модификацию, может видеть промежуточные состояния, в которых инварианты нарушены; с точки зрения всех остальных потоков, обращающихся к той же структуре данных, модификация либо еще не началась, либо уже завершилась. В стандартной библиотеке C++ есть несколько таких механизмов, и в этой главе мы их

опишем.

Другой вариант — изменить дизайн структуры данных и ее инварианты, так чтобы модификация представляла собой последовательность неделимых изменений, каждое из которых сохраняет инварианты. Этот подход обычно называют *программированием без блокировок* (lock-free programming) и реализовать его правильно очень трудно; если вы работаете на этом уровне, то приходится учитывать нюансы модели памяти и разбираться, какие потоки потенциально могут увидеть те или иные наборы значений. Модель памяти обсуждается в главе 5, а программирование без блокировок — в главе 7.

Еще один способ справиться с гонками — рассматривать изменения структуры данных как *транзакцию*, то есть так, как обрабатываются обновления базы данных внутри транзакции. Требуемая последовательность изменений и чтений данных сохраняется в журнале транзакций, а затем атомарно фиксируется. Если фиксация невозможна, потому что структуру данных в это время модифицирует другой поток, то транзакция перезапускается. Это решение называется *программной транзакционной памятью* (Software Transactional Memory — STM), в настоящее время в этой области ведутся активные исследования. Мы не будем рассматривать STM в этой книге, потому что в C++ для нее нет поддержки. Однако к самой идее о том, чтобы выполнить какую-то последовательность действий и за один шаг зафиксировать результаты, я еще вернусь.

Самый простой механизм защиты разделяемых данных из описанных в стандарте C++ — это *мьютекс,* с него мы и начнем рассмотрение.

3.2. Защита разделяемых данных с помощью мьютексов

Итак, у нас есть разделяемая структура данных, например связанный список из предыдущего раздела, и мы хотим защитить его от гонки и нарушения инвариантов, к которым она приводит. Как было бы здорово, если бы мы могли пометить участки кода, в которых производятся обращения к этой структуре данных, *взаимно исключающими*, так что если один поток начинает выполнять такой участок, то все остальные потоки должны ждать, пока первый не завершит обработку данных. Тогда ни один поток, кроме выполняющего модификацию, не смог бы увидеть нарушенный инвариант.

Что ж, это вовсе не сказка — именно такое поведение вы получаете при использовании примитива синхронизации, который называется *мьютекс* (слово mutex происходит от mutual exclusion — взаимное исключение). Перед тем как обратиться к структуре данных, программа *захватывает* (lock) мьютекс, а по завершении операций с ней *освобождает* (unlock) его. Библиотека Thread Library гарантирует, что если один поток захватил некоторый мьютекс, то все остальные потоки, пытающиеся захватить тот же мьютекс, будут вынуждены ждать, пока удачливый конкурент не освободит его. В результате все потоки видят согласованное представление разделяемых данных, без нарушенных инвариантов.

Мьютексы — наиболее общий механизм защиты данных в C++, но панацеей они не являются; важно структурировать код так, чтобы защитить нужные данные (см. раздел 3.2.2), и избегать состояний гонки, внутренне присущих интерфейсам (см раздел 3.2.3). С мьютексами связаны и собственные проблемы, а именно: *взаимоблокировки* (deadlock) (см. раздел 3.2.4), а также защита слишком большого или слишком малого количества данных (см. раздел 3.2.8). Но начнем с простого.

3.2.1. Использование мьютексов в C++

В C++ для создания мьютекса следует сконструировать объект типа `std::mutex`, для захвата мьютекса служит функция-член `lock()`, а для освобождения — функция-член `unlock()`. Однако вызывать эти функции напрямую не рекомендуется, потому что в этом случае необходимо помнить о вызове `unlock()` на каждом пути выхода из функции, в том числе и вследствие исключений. Вместо этого в стандартной библиотеке имеется шаблон класса `std::lock_guard`, который реализует идиому RAII — захватывает мьютекс в конструкторе и освобождает в деструкторе, — гарантируя тем самым, что захваченный мьютекс обязательно будет освобожден. В листинге 3.1 показано, как с помощью классов `std::mutex` и `std::lock_guard` защитить список, к которому могут обращаться несколько потоков. Оба класса определены в заголовке `<mutex>`.

Листинг 3.1. Защита списка с помощью мьютекса

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list; ← (1)
std::mutex some_mutex;    ← (2)
```

```
void add_to_list(int new_value) {
 std::lock_guard<std::mutex> guard(some_mutex); ← (3)
 some_list.push_back(new_value);
}

bool list_contains(int value_to_find) {
 std::lock_guard<std::mutex> guard(some_mutex); ← (4)
 return
  std::find(some_list.begin(), some_list.end(), value_to_find) !=
  some_list.end();
}
```

В листинге 3.1 есть глобальный список **(1)**, который защищен глобальным же объектом `std::mutex` **(2)**. Вызов `std::lock_guard<std::mutex>` в `add_to_list()` **(3)** и `list_contains()` **(4)** означает, что доступ к списку из этих двух функций является взаимно исключающим: `list_contains()` никогда не увидит промежуточного результата модификации списка, выполняемой в `add_to_list()`.

Хотя иногда такое использование глобальных переменных уместно, в большинстве случаев мьютекс и защищаемые им данные помещают в один класс, а не в глобальные переменные. Это не что иное, как стандартное применение правил объектно-ориентированного проектирования; помещая обе сущности в класс, вы четко даете понять, что они взаимосвязаны, а, кроме того, обеспечиваете инкапсулирование функциональности и ограничение доступа. В данном случае функции `add_to_list` и `list_contains` следует сделать функциями-членами класса, а мьютекс и защищаемые им данные — закрытыми переменными-членами класса. Так будет гораздо проще понять, какой код имеет доступ к этим данным и, следовательно, в каких участках программы необходимо захватывать мьютекс. Если все функции-члены класса захватывают мьютекс перед обращением к каким-то другим данным-членам и освобождают по завершении действий, то данные оказываются надежно защищены от любопытствующих.

Впрочем, это не *совсем* верно, проницательный читатель мог бы заметить, что если какая-нибудь функция-член возвращает указатель или ссылку на защищенные данные, то уже неважно, правильно функции-члены управляют мьютексом или нет, ибо вы проделали огромную брешь в защите. *Любой код, имеющий доступ к этому указателю или ссылке, может прочитать (и, возможно, модифицировать) защищенные данные, не захватывая мьютекс.* Таким образом, для защиты данных с помощью мьютекса требуется тщательно проектировать интерфейс, гарантировать, что перед любым доступном к защищенным данным производится захват мьютекса, и не оставлять черных ходов.

3.2.2. Структурирование кода для защиты разделяемых данных

Как мы только что видели, для защиты данных с помощью мьютекса недостаточно просто «воткнуть» объект `std::lock_guard` в каждую функцию-член: один-единственный «отбившийся» указатель или ссылка сводит всю защиту на нет. На некотором уровне проверить наличие таких отбившихся указателей легко — если ни одна функция-член не передает вызывающей программе указатель или ссылку на защищенные данные в виде возвращаемого значения или выходного параметра, то данные в безопасности. Но стоит копнуть чуть глубже, как выясняется, что всё не так просто, — а просто никогда не бывает.

Недостаточно проверить, что функции-члены не возвращают указатели и ссылки вызывающей программе, нужно еще убедиться, что такие указатели и ссылки не передаются в виде *входных* параметров вызываемым ими функциям, которые вы не контролируете. Это ничуть не менее опасно — что, если такая функция сохранит где-то указатель или ссылку, а потом какой-то другой код обратится к данным, не захватив предварительно мьютекс? Особенно следует остерегаться функций, которые передаются во время выполнения в виде аргументов или иными способами, как показано в листинге 3.2.

Листинг 3.2. Непреднамеренная передача наружу ссылки на защищённые данные

```
class some_data {
 int a;
 std::string b;
public:
 void do_something();
};

class data_wrapper {
private:
 some_data data;
 std::mutex m;
public :
 template<typename Function>
 void process_data(Function func)
 {
  std::lock_guard<std::mutex> l(m);
  func(data);
 }
};

some_data* unprotected;

void malicious_function(some_data& protected_data) {
 unprotected = &protected_data;
}

data_wrapper x;

void foo
{
 x.process_data(malicious_function);
 unprotected->do_something();
}
```

(1) Передаем "защищенные" данные пользовательской функции

(2) Передаем вредоносную функцию

(3) Доступ к "защищенным" данным в обход защиты

В этом примере функция-член `process_data` выглядит вполне безобидно, доступ к данным охраняется объектом `std::lock_guard`, однако наличие обращения к переданной пользователем функции `func` **(1)** означает, что `foo` может передать вредоносную функцию `malicious_function`, чтобы обойти защиту **(2)**, а затем вызвать `do_something()`, не захватив предварительно мьютекс **(3)**.

Здесь фундаментальная проблема заключается в том, что мы не сделали того, что

собирались сделать: пометить все участки кода, в которых имеется доступ к структуре данных, как *взаимно исключающие*. В данном случае мы забыли о коде внутри `foo()`, который вызывает `unprotected->do_something()`. К сожалению, в этом стандартная библиотека C++ нам помочь не в силах: именно программист должен позаботиться о том, чтобы защитить данные мьютексом. Но не всё так мрачно — следование приведенной ниже рекомендации выручит в таких ситуациях. *Не передавайте указатели и ссылки на защищенные данные за пределы области видимости блокировки никаким способом, будь то возврат из функции, сохранение в видимой извне памяти или передача в виде аргумента пользовательской функции*.

Хотя описанная только что ситуация — самая распространенная ошибка при защите разделяемых данных, перечень подводных камней ей отнюдь не исчерпывается. В следующем разделе мы увидим, что гонка возможна даже, если данные защищены мьютексом.

3.2.3. Выявление состояний гонки, внутренне присущих интерфейсам

Тот факт, что вы пользуетесь мьютексами или другим механизмом для защиты разделяемых данных, еще не означает, что гонок можно не опасаться, — следить за тем, чтобы данные были защищены, все равно нужно. Вернемся снова к примеру двусвязного списка. Чтобы поток мог безопасно удалить узел, необходимо предотвратить одновременный доступ к трем узлам: удаляемому и двум узлам но обе стороны от него. Заблокировав одновременный доступ к указателям на каждый узел но отдельности, мы не достигнем ничего по сравнению с вариантом, где мьютексы вообще не используются, поскольку гонка по-прежнему возможна. Защищать нужно не отдельные узлы на каждом шаге, а структуру данных в целом на все время выполнения операции удаления. Простейшее решение в данном случае — завести один мьютекс, который будет защищать весь список, как в листинге 3.1.

Однако и после обеспечения безопасности отдельных операций наши неприятности еще не закончились — гонки все еще возможны, даже для самого простого интерфейса. Рассмотрим структуру данных для реализации стека, например, адаптер контейнера `std::stack`, показанный в листинге 3.3. Помимо конструкторов и функции `swap()`, имеется еще пять операций со стеком: `push()` заталкивает в стек новый элемент, `pop()` выталкивает элемент из стека, `top()` возвращает элемент, находящийся на вершине стека, `empty()` проверяет, пуст ли стек, и `size()` возвращает размер стека. Если изменить `top()`, так чтобы она возвращала копию, а не ссылку (в соответствии с рекомендацией из раздела 3.2.2), и защитить внутренние данные мьютексом, то и тогда интерфейс уязвим для гонки. Проблема не в реализации на основе мьютексов, она присуща самому интерфейсу, то есть гонка может возникать даже в реализации без блокировок.

Листинг 3.3. Интерфейс адаптера контейнера `std::stack`

```
template<typename T, typename Container = std::deque<T> >
class stack {
public:
 explicit stack(const Container&);
 explicit stack(Container&& = Container());
 template <class Alloc> explicit stack(const Alloc&);
 template <class Alloc> stack(const Container&, const Alloc&);
```

```
 template <class Alloc> stack(Container&&, const Alloc&);
 template <class Alloc> stack(stack&&, const Alloc&);
 bool empty() const;
 size_t size() const;
 T& top();
 T const& top() const;
 void push(T const&);
 void push(T&&);
 void pop();
 void swap(stack&&);
};
```

Проблема в том, что на результаты, возвращенные функциями `empty()` и `size()`, нельзя полагаться — хотя в момент вызова они, возможно, и были правильны, но после возврата из функции любой другой поток может обратиться к стеку и затолкнуть в него новые элементы, либо вытолкнуть существующие, причем это может произойти до того, как у потока, вызвавшего `empty()` или `size()`, появится шанс воспользоваться полученной информацией.

Если экземпляр `stack` *не является разделяемым,* то нет ничего страшного в том, чтобы проверить, пуст ли стек с помощью `empty()`, а затем, если стек не пуст, вызвать `top()` для доступа к элементу на вершине стека:

```
stack<int> s;
if (!s.empty())                ← (1)
{
 int const value = s.top();    ← (2)
 s.pop();                      ← (3)
 do_something(value);
}
```

Такой подход в однопоточном коде не только безопасен, но и единственно возможен: вызов `top()` для пустого стека приводит к неопределенному поведению. Но если объект `stack` является разделяемым, то *такая последовательность операций уже не безопасна,* так как между вызовами `empty()` **(1)** и `top()` **(2)** другой поток мог вызвать `pop()` и удалить из стека последний элемент. Таким образом, мы имеем классическую гонку, и использование внутреннего мьютекса для защиты содержимого стека ее не предотвращает. Это следствие дизайна интерфейса.

И что же делать? Поскольку проблема коренится в дизайне интерфейса, то и решать ее надо путем изменения интерфейса. Но возникает вопроса — как его изменить? В простейшем случае мы могли бы просто декларировать, что `top()` возбуждает исключение, если в момент вызова в стеке нет ни одного элемента. Формально это решает проблему, но затрудняет программирование, поскольку теперь мы должны быть готовы к перехвату исключения, даже если вызов `empty()` вернул `false`. По сути дела, вызов `empty()` вообще оказывается ненужным.

Внимательно присмотревшись к показанному выше фрагменту, мы обнаружим еще одну потенциальную гонку, на этот раз между вызовами `top()` **(2)** и `pop()` **(3)**. Представьте, что этот фрагмент исполняют два потока, ссылающиеся на один и тот же объект `s` типа `stack`. Ситуация вполне обычная: при использовании потока для повышения производительности часто бывает так, что несколько потоков исполняют один и тот же код для разных данных, и разделяемый объект `stack` идеально подходит для разбиения работы между потоками. Предположим, что первоначально в стеке находится два элемента, поэтому можно с уверенностью сказать, что между `empty()` и `top()` не будет гонки ни в одном потоке. Теперь

рассмотрим возможные варианты выполнения программы.

Если стек защищен внутренним мьютексом, то в каждый момент времени лишь один поток может исполнять любую функцию-член стека, поэтому обращения к функциям-членам строго чередуются, тогда как вызовы `do_something()` могут исполняться параллельно. Вот одна из возможных последовательностей выполнения:

Поток A -	- **Поток B**
`if (!s.empty())`	
	`if (!s.empty())`
`int const value = s.top();`	
	`int const value = s.top();`
`s.pop();`	
`do_something(value);`	`s.pop();`
	`do_something(value);`

Как видите, если работают только эти два потока, то между двумя обращениями к `top()` никто не может модифицировать стек, так что оба потока увидят одно и то же значение. Однако беда в том, что *между обращениями к `pop()` нет обращений к `top()`*. Следовательно, одно из двух хранившихся в стеке значений никто даже не прочитает, оно будет просто отброшено, тогда как другое будет обработано дважды. Это еще одно состояние гонки, и куда более коварное, чем неопределенное поведение в случае гонки между `empty()` и `top()`, — на первый взгляд, ничего страшного не произошло, а последствия ошибки проявятся, скорее всего, далеко от места возникновения, хотя, конечно, всё зависит от того, что именно делает функция `do_something()`.

Для решения проблемы необходимо более радикальное изменение интерфейса — выполнение обеих операций `top()` и `pop()` под защитой одного мьютекса. Том Каргилл[4] указал, что такой объединенный вызов приводит к проблемам в случае, когда копирующий конструктор объектов в стеке может возбуждать исключения. С точки зрения безопасности относительно исключений, задачу достаточно полно решил Герб Саттер[5], однако возможность возникновения гонки вносит в нее новый аспект.

Для тех, кто незнаком с историей вопроса, рассмотрим класс `stack<vector<int>>`. Вектор — это контейнер с динамически изменяемым размером, поэтому при копировании вектора библиотека должна выделить из кучи память. Если система сильно загружена или имеются жесткие ограничения на ресурсы, то операция выделения памяти может завершиться неудачно, и тогда копирующий конструктор вектора возбудит исключение `std::bad_alloc`. Вероятность такого развития событий особенно велика, если вектор содержит много элементов. Если бы функция `pop()` возвращала вытолкнутое из стека значение, а не только удаляла его из стека, то мы получили бы потенциальную проблему: вытолкнутое значение возвращается вызывающей программе только после модификации стека, но в процессе копирования возвращаемых данных может возникнуть исключение. Если такое случится, то только что вытолкнутые данные будут потеряны — из стека они удалены, но никуда не скопированы! Поэтому проектировщики интерфейса `std::stack` разбили операцию на две: получить элемент, находящийся на вершине (`top()`), а затем удалить его из стека (`pop()`). Теперь, данные, которые не удалось скопировать, остаются в стеке; если проблема связана с нехваткой памяти в куче, то, возможно, приложение сможет освободить немного памяти и попытаться выполнить операцию еще раз.

Увы, это как раз то разбиение, которого мы пытались избежать в попытке уйти от гонки! К счастью, альтернативы имеются, но они не бесплатны.

Вариант 1: передавать ссылку

Первый вариант решения — передавать функции `pop()` ссылку на переменную, в которую она должна будет поместить вытолкнутое из стека значение:

```
std::vector<int> result;
some_stack.pop(result);
```

Во многих случаях это приемлемо, но есть и очевидный недостаток: вызывающая программа должна до обращения к функции сконструировать объект того типа, которым конкретизирован стек, чтобы передать его в качестве аргумента. Для некоторых типов это не годится, так как конструирование дорого обходится с точки зрения времени или потребления ресурсов. Для других типов это вообще может оказаться невозможно, так как конструкторы требуют параметров, которые в данной точке программы могут быть недоступны. Наконец, требуется, чтобы хранящийся в стеке тип допускал присваивание. Это существенное ограничение, многие пользовательские типы не поддерживают присваивание, хотя могут поддерживать конструирование перемещением и даже копированием (и потому допускают возврат по значению).

Вариант 2: потребовать наличия копирующего конструктора, не возбуждающего исключений, или перемещающего конструктора

Проблема с безопасностью относительно исключений в варианте функции pop(), возвращающей значение, проявляется только тогда, когда исключение может возникать в процессе возврата значения. Во многих типах имеются копирующие конструкторы, которые не возбуждают исключений, а после поддержки в стандарте C++ ссылок на r-значения (см. приложение A, раздел A.1), появилось еще много типов, в которых перемещающий конструктор не возбуждает исключений, даже если копирующий конструктор может их возбуждать. Один из вариантов решения заключается в том, чтобы наложить на потокобезопасный стек ограничение: в нем можно хранить только типы, поддерживающие возврат по значению без возбуждения исключений.

Это решение, пусть и безопасное, не идеально. Хотя на этапе компиляции можно узнать, существует ли копирующий или перемещающий конструктор, который не возбуждает исключений, — с помощью концепций `std::is_nothrow_copy_constructible`, `std::is_nothrow_move_constructible` и характеристик типов, но это слишком ограничительное требование. Пользовательских типов, в которых копирующий конструктор может возбуждать исключение и перемещающего конструктора нет, гораздо больше, чем типов, в которых копирующий и (или) перемещающий конструктор гарантированно не возбуждают исключений (хотя ситуация может измениться, когда разработчики привыкнут к появившейся в C++11 поддержке ссылок на r-значения). Было бы крайне нежелательно запрещать хранение таких объектов в потокобезопасном стеке.

Вариант 3: возвращать указатель на вытолкнутый элемент

Третий вариант — возвращать не копию вытолкнутого элемента по значению, а

указатель на него. Его достоинство в том, указатели можно копировать, не опасаясь исключений, поэтому указанную Каргиллом проблему мы обходим. А недостаток в том, что возврат указателя заставляет искать средства для управления выделенной объекту памятью, так что для таких простых типов, как целые числа, накладные расходы на управление памятью могут превысить затраты на возврат типа по значению. В любом интерфейсе, где применяется этот вариант, в качестве типа указателя было бы разумно избрать `std::shared_ptr`; мало того что это предотвращает утечки памяти, поскольку объект уничтожается вместе с уничтожением последнего указателя на него, так еще и библиотека полностью контролирует схему распределения памяти и не требует использования `new` и `delete`. Это существенно с точки зрения оптимизации — требование, чтобы память для всякого хранящегося в стеке объекта выделялась с помощью `new`, повлекло бы заметные накладные расходы по сравнению с исходной версией, небезопасной относительно потоков.

Вариант 4: реализовать одновременно вариант 1 и один из вариантов 2 или 3

Никогда не следует пренебрегать гибкостью, особенно в обобщенном коде. Если остановиться на варианте 2 или 3, то будет сравнительно нетрудно реализовать и вариант 1, а это оставит пользователю возможность выбрать наиболее подходящее решение ценой очень небольших накладных расходов.

Пример определения потокобезопасного стека

В листинге 3.4 приведено определение класса стека со свободным от гонок интерфейсом. В нем реализованы приведенные выше варианты 1 и 3: имеется два перегруженных варианта функции-члена `pop()` — один принимает ссылку на переменную, в которой следует сохранить значение, а второй возвращает `std::shared_ptr<>`. Интерфейс предельно прост, он содержит только функции: `push()` и `pop()`.

Листинг 3.4. Определение класса потокобезопасного стека

```
#include <exception>

struct empty_stack: std::exception {
 const char* what() const throw();
};

template<typename T>
class threadsafe_stack {
public:
 threadsafe_stack();
 threadsafe_stack(const threadsafe_stack&);
 threadsafe_stack& operator=(const threadsafe_stack&)
  = delete;← (1)

 void push(T new_value);
 std::shared_ptr<T> pop();
 void pop(T& value);
 bool empty() const;
};
```

Упростив интерфейс, мы добились максимальной безопасности — даже операции со

стеком в целом ограничены: стек нельзя присваивать, так как оператор присваивания удален **(1)** (см. приложение А, раздел А.2) и функция `swap()` отсутствует. Однако стек можно копировать в предположении, что можно копировать его элементы. Обе функции `pop()` возбуждают исключение `empty_stack`, если стек пуст, поэтому программа будет работать, даже если стек был модифицирован после вызова `empty()`. В описании варианта 3 выше отмечалось, что использование `std::shared_ptr` позволяет стеку взять на себя распределение памяти и избежать лишних обращений к `new` и `delete`. Теперь из пяти операций со стеком осталось только три: `push()`, `pop()` и `empty()`. И даже `empty()` лишняя. Чем проще интерфейс, тем удобнее контролировать доступ к данным — можно захватывать мьютекс на все время выполнения операции. В листинге 3.5 приведена простая реализация в виде обертки вокруг класс `std::stack<>`.

Листинг 3.5. Определение класса потокобезопасного стека

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>

struct empty_stack: std::exception {
 const char* what() const throw();
};

template<typename T>
class threadsafe_stack {
private:
 std::stack<T> data;
 mutable std::mutex m;
public:
 threadsafe_stack(){}
 threadsafe_stack(const threadsafe_stack& other) {
  std::lock_guard<std::mutex> lock(other.m);
  data = other.data; ←┐ (1) Копирование производится в теле
 }                   | конструктора
 threadsafe_stack& operator=(const threadsafe_stack&) = delete;

 void push(T new_value) {
  std::lock_guard<std::mutex> lock(m);
  data.push(new_value);
 }

 std::shared_ptr<T> pop()| Перед тем как выталкивать значение,
 {                       ←┘ проверяем, не пуст ли стек
  std::lock_guard<std::mutex> lock(m);
  if (data.empty()) throw empty_stack();
  std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
  data.pop(); ←┐ Перед тем как модифицировать стек
  return res;  | в функции pop(), выделяем память
 }             | для возвращаемого значения

 void pop(T& value) {
```

```
    std::lock_guard<std::mutex> lock(m);
    if (data.empty()) throw empty_stack();
    value = data.top();
    data.pop();
   }

   bool empty() const {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
   }
};
```

Эта реализация стека даже *допускает копирование* — копирующий конструктор захватывает мьютекс в объекте-источнике, а только потом копирует внутренний стек. Копирование производится в теле конструктора **(1)**, а не в списке инициализации членов, чтобы мьютекс гарантированно удерживался в течение всей операции.

Обсуждение функций `top()` и `pop()` показывает, что проблематичные гонки в интерфейсе возникают из-за слишком малой гранулярности блокировки — защита не распространяется на выполняемую операцию в целом. Но при использовании мьютексов проблемы могут возникать также из-за слишком большой гранулярности, крайним проявление этого является применение одного глобального мьютекса для защиты всех разделяемых данных. В системе, где разделяемых данных много, такой подход может свести на нет все преимущества параллелизма, постольку потоки вынуждены работать но очереди, даже если обращаются к разным элементам данных. В первых версиях ядра Linux для многопроцессорных систем использовалась единственная глобальная блокировка ядра. Это решение работало, но получалось, что производительность одной системы с двумя процессорами гораздо ниже, чем двух однопроцессорных систем, а уж сравнивать производительность четырёхпроцессорной системы с четырьмя однопроцессорными вообще не имело смысла — конкуренция за ядро оказывалась настолько высока, что потоки, исполняемые дополнительными процессорами, не могли выполнять полезную работу. В последующих версиях Linux гранулярность блокировок ядра уменьшилась, и в результате производительность четырёхпроцессорной системы приблизилась к идеалу — четырехкратной производительности однопроцессорной системы, так как конкуренция за ядро значительно снизилась.

При использовании мелкогранулярных схем блокирования иногда для защиты всех данных, участвующих в операции, приходится захватывать более одного мьютекса. Как отмечалось выше, бывают случаи, когда лучше повысить гранулярность защищаемых данных, чтобы для их защиты хватило одного мьютекса. Но это не всегда желательно, например, если мьютексы защищают отдельные экземпляры класса. В таком случае блокировка «на уровень выше» означает одно из двух: передать ответственность за блокировку пользователю или завести один мьютекс, который будет защищать все экземпляры класса. Ни одно из этих решений не вызывает восторга.

Но когда для защиты одной операции приходится использовать два или более мьютексов, всплывает очередная проблема: *взаимоблокировка*. По природе своей она почти противоположна гонке: если в случае гонки два потока состязаются, кто придет первым, то теперь каждый поток ждет другого, и в результате ни тот, ни другой не могут продвинуться ни на шаг.

3.2.4. Взаимоблокировка: проблема и решение

Представьте игрушку, состоящую из двух частей, причем для игры необходимы обе части, — например, игрушечный барабан и палочки. Теперь вообразите двух ребятишек, которые любят побарабанить. Если одному дать барабан с палочками, то он будет радостно барабанить, пока не надоест. Если другой тоже хочет поиграть, то ему придётся подождать, как бы это ни было печально. А теперь представьте, что барабан и палочки закопаны где-то в ящике для игрушек (порознь), и оба малыша захотели поиграть с ними одновременно. Один отыскал барабан, а другой палочки. И оба оказались в тупике — если кто-то один не решится уступить и позволить поиграть другому, то каждый будет держаться за то, что имеет, требуя, чтобы другой отдал недостающее. В результате побарабанить не сможет никто.

А теперь от детей и игрушек перейдём к потокам, ссорящимся по поводу захвата мьютексов, — оба потока для выполнения некоторой операции должны захватить два мьютекса, но сложилось так, что каждый поток захватил только один мьютекс и ждет другого. Ни один поток не может продолжить, так как каждый ждет, пока другой освободит нужный ему мьютекс. Такая ситуация называется *взаимоблокировкой*; это самая трудная из проблем, возникающих, когда для выполнения операции требуется захватить более одного мьютекса.

Общая рекомендация, как избежать взаимоблокировок, заключается в том, чтобы всегда захватывать мьютексы в одном и том же порядке, — если мьютекс A всегда захватывается раньше мьютекса B, то взаимоблокировка не возникнет. Иногда это просто, потому что мьютексы служат разным целям, а иногда совсем не просто, например, если каждый мьютекс защищает отдельный объект одного и того же класса. Рассмотрим, к примеру, операцию сравнения двух объектов одного класса. Чтобы сравнению не мешала одновременная модификация, необходимо захватить мьютексы для обоих объектов. Однако, если выбрать какой-то определенный порядок (например, сначала захватывается мьютекс для объекта, переданного в первом параметре, а потом — для объекта, переданного во втором параметре), то легко можно получить результат, обратный желаемому: стоит двум потокам вызвать функцию сравнения, передав ей одни и те же объекты в разном порядке, как мы получим взаимоблокировку!

К счастью, в стандартной библиотеке есть на этот случай лекарство в виде функции `std::lock`, которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку. В листинге 3.6 показано, как воспользоваться ей для реализации простой операции обмена.

Листинг 3.6. Применение `std::lock` и `std::lock_guard` для реализации операции обмена

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);

class X {
private:
 some_big_object some_detail;
 std::mutex m;
public:
 X(some_big_object const& sd) : some_detail(sd) {}
 friend void swap(X& lhs, X& rhs) {
```

```
    if (&lhs == &rhs)
     return;
    std::lock(lhs.m, rhs.m); ← (1)
    std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);← (2)
    std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);← (3)
    swap(lhs.some_detail,rhs.some_detail);
   }
};
```

Сначала проверяется, что в аргументах переданы разные экземпляры, постольку попытка захватить `std::mutex`, когда он уже захвачен, приводит к неопределенному поведению. (Класс мьютекса, допускающего несколько захватов в одном потоке, называется `std::recursive_mutex`. Подробности см. в разделе 3.3.3.) Затем мы вызываем `std::lock()` **(1)**, чтобы захватить оба мьютекса, и конструируем два экземпляра `std::lock_guard` **(2)**, **(3)** — по одному для каждого мьютекса. Помимо самого мьютекса, конструктору передается параметр `std::adopt_lock`, сообщающий объектам `std::lock_guard`, что мьютексы уже захвачены, и им нужно лишь принять владение существующей блокировкой, а не пытаться еще раз захватить мьютекс в конструкторе.

Это гарантирует корректное освобождение мьютексов при выходе из функции даже в случае, когда защищаемая операция возбуждает исключение, а также возврат результата сравнения в случае нормального завершения. Стоит также отметить, что попытка захвата любого мьютекса `lhs.m` или `rhs.m` внутри `std::lock` может привести к исключению; в этом случае исключение распространяется на уровень функции, вызвавшей `std::lock`. Если `std::lock` успешно захватила первый мьютекс, но при попытке захватить второй возникло исключение, то первый мьютекс автоматически освобождается; `std::lock` обеспечивает семантику «все или ничего» в части захвата переданных мьютексов.

Хотя `std::lock` помогает избежать взаимоблокировки в случаях, когда нужно захватить сразу два или более мьютексов, она не в силах помочь, если мьютексы захватываются порознь, — в таком случае остается полагаться только на дисциплину программирования. Это нелегко, взаимоблокировки — одна из самых неприятных проблем в многопоточной программе, часто они возникают непредсказуемо, хотя в большинстве случаев все работает нормально. Однако все же есть несколько относительно простых правил, помогающих писать свободный от взаимоблокировок код.

3.2.5. Дополнительные рекомендации, как избежать взаимоблокировок

Взаимоблокировка может возникать не только при захвате мьютексов, хотя это и наиболее распространенная причина. Нарваться на взаимоблокировку можно и тогда, когда есть два потока и никаких мьютексов; достаточно, чтобы каждый поток вызвал функцию `join()` объекта `std::thread`, связанного с другим потоком. В этом случае ни один поток не сможет продолжить выполнение, потому что будет ждать завершения другого потока, — точно так же, как дети, ссорящиеся по поводу игрушек. Такой простой цикл может возникнуть всюду, где один поток ждет завершения другого для продолжения работы, а этот другой поток одновременно ждет завершения первого. Причем потоков необязательно должно быть два — цикл, в котором участвуют три и более потоков, также приведёт к взаимоблокировке. Общая рекомендация во всех случаях одна: не ждите завершения другого потока, если есть малейшая возможность, что он будет дожидаться вас. Ниже приведены

конкретные советы, как выявить и устранить такую возможность.

Избегайте вложенных блокировок

Идея проста — не захватывайте мьютекс, если уже захватили какой-то другой. Если строго придерживаться этой рекомендации, то взаимоблокировка, обусловленная одними лишь захватами мьютексов, никогда не возникнет, потому что каждый поток в любой момент времени владеет не более чем одним мьютексом. Конечно, можно получить взаимоблокировку по другим причинам (например, из-за взаимного ожидания потоков), но захват мьютексов — наиболее распространенная. Если требуется захватить сразу несколько мьютексов, делайте это атомарно с помощью `std::lock`, так вы сможете избежать взаимоблокировки.

Старайтесь не вызывать пользовательский код, когда удерживаете мьютекс

По существу, это простое развитие предыдущей рекомендации. Поскольку код написан пользователем, вы не можете знать, что он делает. А делать он может все, что угодно, в том числе захватывать мьютекс. Если вы вызываете пользовательский код, не освободив предварительно мьютекс, а этот код захватывает какой-то мьютекс, то оказывается нарушена рекомендация избегать вложенных блокировок, и может возникнуть взаимоблокировка. Иногда избежать этого невозможно: если вы пишете обобщенный код, например класс стека из раздела 3.2.3, то каждая операция над типом или типами параметров — не что иное, как пользовательский код. В таком случае прислушайтесь к следующему совету.

Захватывайте мьютексы в фиксированном порядке

Если без захвата нескольких мьютексов никак не обойтись и захватить их в одной операции типа `std::lock` не получается, то следует прибегнуть к другому способу — захватывать их во всех потоках в одном и том же порядке. Мы уже говорили об этом в разделе 3.2.4, как о способе избежать взаимоблокировки при захвате двух мьютексов; идея в том, чтобы четко определить порядок захвата и соблюдать его во всех потоках. Иногда это сравнительно просто. Например, в случае стека из раздела 3.2.3 мьютекс хранится в каждом экземпляре стека, но для операций над хранящимися в стеке элементами необходимо вызывать пользовательский код. Однако можно добавить ограничение: никакая операция над хранящимися в стеке данными не должна производить какие-либо действия с самим стеком. Это возлагает определенную ответственность на пользователя стека, но на практике редко бывает, чтобы хранящимся в контейнере данным нужно было обращаться к самому контейнеру, а если такое и случается, то сразу видно. Поэтому бремя ответственности не слишком тяжело.

Но не всегда всё так просто, и пример мы видели при рассмотрении оператора сравнения в разделе 3.2.4. В этом конкретном случае есть возможность захватить мьютексы одновременно, но так бывает не всегда. Пример связанного списка из раздела 3.1 дает еще

один способ защитить список — хранить мьютекс в каждом узле. Тогда, чтобы получить доступ к списку, поток должен будет захватить мьютекс для каждого интересующего его узла. Так, чтобы удалить элемент, надо будет захватить мьютексы трех узлов — удаляемого, предшествующего и последующего, — постольку все они так или иначе модифицируются. Аналогично для обхода списка поток должен удерживать мьютекс текущего узла, пока не захватит мьютекс следующего за ним; это гарантирует, что никто не может изменить указатель на следующий узел. Захватив мьютекс следующего узла, можно освободить мьютекс текущего, так как больше он не понадобится.

Такой способ «передачи из рук в руки» позволяет нескольким потокам одновременно обходить список при условии, что разные потоки обращаются к разным узлам. Но чтобы предотвратить взаимоблокировку, узлы следует обходить в одном и том же порядке; если один поток обходит список в одном направлении, а другой в противоположном, то при передаче мьютексов «из рук в руки» в середине списка может произойти взаимоблокировка. Если узлы A и B соседние, то поток, который обходит список в прямом направлении, попытается захватить мьютекс B, удерживая мьютекс A. В то же время поток, который обходит список в обратном направлении, попытается захватить мьютекс A, удерживая мьютекс B. Вот мы и получили классическую взаимоблокировку.

Рассмотрим еще ситуацию удаления узла B, расположенного между A и C. Если поток захватывает мьютекс B раньше, чем мьютексы A и C, то возможна взаимоблокировка с потоком, который обходит список. Такой поток попытается сначала захватить мьютекс A или C (в зависимости от направления обхода), но потом обнаружит, что не может захватить мьютекс B, потому что поток, выполняющий удаление, удерживает этот мьютекс, пытаясь в то же время захватить мьютексы A и C.

Предотвратить в этом случае взаимоблокировку можно, определив порядок обхода, так что поток всегда должен захватывать мьютекс A раньше мьютекса B, а мьютекс B раньше мьютекса C. Это устранило бы возможность взаимоблокировки, но ценой запрета обхода в обратном направлении. Подобные соглашения можно принять и для других структур данных.

Пользуйтесь иерархией блокировок

Являясь частным случаем фиксированного порядка захвата мьютексов, иерархия блокировок в то же время позволяет проверить соблюдение данного соглашения во время выполнения. Идея в том, чтобы разбить приложение на отдельные слои и выявить все мьютексы, которые могут быть захвачены в каждом слое. Программе будет отказано в попытке захватить мьютекс, если она уже удерживает какой-то мьютекс из нижележащего слоя. Чтобы проверить это во время выполнения, следует приписать каждому мьютексу номер слоя и вести учет мьютексам, захваченным каждым потоком. В следующем листинге приведен пример двух потоков, пользующихся иерархическим мьютексом.

Листинг 3.7. Использование иерархии блокировок для предотвращения взаимоблокировки

```
hierarchical_mutex high_level_mutex(10000); ← (1)
hierarchical_mutex low_level_mutex(5000);   ← (2)
```

```
int do_low_level_stuff();

int low_level_func() {
 std::lock_guard<hierarchical_mutex> lk(low_level_mutex); ← (3)
 return do_low_level_stuff();
}

void high_level_stuff(int some_param);

void high_level_func() {
 std::lock_guard<hierarchical_mutex> lk(high_level_mutex); ← (4)
 high_level_stuff(low_level_func());                       ← (5)
}

void thread_a() { ← (6)
 high_level_func();
}

hierarchical_mutex other_mutex(100); ← (7)

void do_other_stuff();

void other_stuff() {
 high_level_func(); ← (8)
 do_other_stuff();
}

void thread_b() { ← (9)
 std::lock_guard<hierarchical_mutex> lk(other_mutex); ← (10)
 other_stuff();
}
```

Поток `thread_a()` **(6)** соблюдает правила и выполняется беспрепятственно. Напротив, поток `thread_b()` **(9)** нарушает правила, поэтому во время выполнения столкнется с трудностями. Функция `thread_a()` вызывает `high_level_func()`, которая захватывает мьютекс `high_level_mutex` **(4)** (со значением уровня иерархии 10000 **(1)**), а затем вызывает `low_level_func()` **(5)** (мьютекс в этот момент уже захвачен), чтобы получить параметр, необходимый функции `high_level_stuff()`. Далее функция `low_level_func()` захватывает мьютекс `low_level_mutex` **(3)**, и в этом нет ничего плохого, так как уровень иерархии для него равен 5000 **(2)**, то есть меньше, чем для `high_level_mutex`.

С другой стороны, функция `thread_b()` некорректна. Первым делом она захватывает мьютекс `other_mutex` **(10)**, для которого уровень иерархии равен всего 100 **(7)**. Это означает, что мьютекс призван защищать только данные очень низкого уровня. Следовательно, когда функция `other_stuff()` вызывает `high_level_func()` **(8)**, она нарушает иерархию — `high_level_func()` пытается захватить мьютекс `high_level_mutex`, уровень иерархии которого (10000) намного больше текущего уровня иерархии 100. Поэтому `hierarchical_mutex` сообщит об ошибке, возбудив исключение или аварийно завершив программу. Таким образом, взаимоблокировки между иерархическими мьютексами

невозможны, так как они сами следят за порядком захвата. Это означает, что программа не может удерживать одновременно два мьютекса, находящихся на одном уровне иерархии, поэтому в схемах «передачи из рук в руки» требуется, чтобы каждый мьютекс в цепочке имел меньшее значение уровня иерархии, чем предыдущий, — на практике удовлетворить такому требованию не всегда возможно.

На этом примере демонстрируется еще один момент — использование шаблона `std::lock_guard<>`, конкретизированного определенным пользователем типом мьютекса. Тип `hierarchical_mutex` не определен в стандарте, но написать его несложно — простая реализация приведена в листинге 3.8. Хотя этот тип определен пользователем, его можно употреблять совместно с `std::lock_guard<>`, потому что в нем имеются все три функции-члена, необходимые для удовлетворения требований концепции мьютекса: `lock()`, `unlock()` и `try_lock()`. Мы еще не видели, как используется функция t`ry_lock()`, но ничего хитрого в ней нет — если мьютекс захвачен другим потоком, то функция сразу возвращает `false`, а не блокирует вызывающий поток в ожидании освобождения мьютекса. Она может вызываться также из функции `std::lock()` для реализации алгоритма предотвращения взаимоблокировок.

Листинг 3.8. Простая реализация иерархического мьютекса

```
class hierarchical_mutex {
 std::mutex internal_mutex;
 unsigned long const hierarchy_value;
 unsigned previous_hierarchy_value;
 static thread_local
  unsigned long this_thread_hierarchy_value;← (1)

 void check_for_hierarchy_violation() {
  if (this_thread_hierarchy_value <= hierarchy_value) ← (2)
  {
   throw std::logic_error("mutex hierarchy violated");
  }
 }

 void update_hierarchy_value() {
  previous_hierarchy_value = this_thread_hierarchy_value; ← (3)
  this_thread_hierarchy_value = hierarchy_value;
 }

public:
 explicit hierarchical_mutex(unsigned long value):
  hierarchy_value(value),
  previous_hierarchy_value(0) {}

 void lock() {
  check_for_hierarchy_violation();
  internal_mutex.lock();    ← (4)
  update_hierarchy_value(); ← (5)
 }

 void unlock() {
```

```
    this_thread_hierarchy_value = previous_hierarchy_value; ← (6)
    internal_mutex.unlock();
   }

   bool try_lock() {
    check_for_hierarchy_violation();
    if (!internal_mutex.try_lock()) ← (7)
     return false;
    update_hierarchy_value();
    return true;
   }
  };

  thread_local unsigned long
  hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);← (8)
```

Главное здесь — использование значения типа `thread_local` для представления уровня иерархии в текущем потоке, `this_thread_hierarchy_value` **(1)**. Оно инициализируется максимально возможным значением **(8)**, так что в начальный момент можно захватить любой мьютекс. Поскольку переменная имеет тип `thread_local`, то в каждом потоке хранится отдельная ее копия, то есть состояние этой переменной в одном потоке не зависит от ее состояния в любом другом. Дополнительные сведения о `thread_local` см. в разделе А.8 приложения А.

Итак, при первом захвате потоком объекта `hierarchical_mutex` значение `this_thread_hierarchy_value` в нем будет равно `ULONG_MAX`. Это число по определению больше любого другого представимого в программе, потому проверка в функции `check_for_hierarchy_violation()` **(2)** проходит. Раз так, то функция `lock()` просто захватывает внутренний мьютекс **(4)**. Успешно выполнив эту операцию, мы можем изменить значение уровня иерархии **(5)**.

Если теперь попытаться захватить *другой* объект `hierarchical_mutex`, не освободив первый, то в переменной `this_thread_hierarchy_value` будет находиться уровень иерархии первого мьютекса. Чтобы проверка **(2)** завершилась успешно, уровень иерархии второго мьютекса должен быть меньше уровня уже удерживаемого.

Теперь мы должны сохранить предыдущее значение уровня иерархии в текущем потоке, чтобы его можно было восстановить в функции `unlock()` **(6)**. В противном случае нам больше никогда не удалось бы захватить мьютекс с более высоким уровнем иерархии, даже если поток не удерживает ни одного мьютекса. Поскольку мы сохраняем предыдущий уровень иерархии только в случае, когда удерживаем `internal_mutex` **(3)**, и восстанавливаем его *перед* тем, как освободить этот внутренний мьютекс **(6)**, то можем безопасно сохранить его в самом объекте `hierarchical_mutex`, где его защищает захваченный внутренний мьютекс.

Функция `try_lock()` работает так же, как `lock()`, с одним отличием — если вызов `try_lock()` для `internal_mutex` завершается ошибкой **(7)**, то мы не владеем мьютексом и, следовательно, не изменяем уровень иерархии, а вместо `true` возвращаем `false`.

Все проверки производятся на этапе выполнения, но, по крайней мере, они не зависят от времени — нет нужды дожидаться, пока сложатся редкие условия, при которых возникает взаимоблокировка. Кроме того, ход мыслей проектировщика, направленный на подобное отделение логики приложения от мьютексов, помогает предотвратить многие возможные

причины взаимоблокировок еще до того, как они прокрадутся в код. Такое мысленное упражнение полезно проделать даже в том случае, когда проектировщик не собирается фактически кодировать проверки во время выполнения.

Применение данных рекомендаций не ограничивается блокировками

Я уже упоминал в начале этого раздела, что взаимоблокировка может возникать не только вследствие захвата мьютекса, а вообще в любой конструкции синхронизации, сопровождающейся циклом ожидания. Поэтому стоит обобщить приведенные выше рекомендации и на такие случаи. Например, мы говорили, что следует по возможности избегать вложенных блокировок, и точно так же не рекомендуется ждать поток, удерживая мьютекс, потому что этому потоку может потребоваться тот же самый мьютекс для продолжения работы. Аналогично, если вы собираетесь ждать завершения потока, то будет разумно определить иерархию потоков, так чтобы любой поток мог ждать только завершения потоков, находящихся ниже него в иерархии. Простой способ реализовать эту идею — сделать так, чтобы присоединение потоков происходило в той же функции, которая их запускала (как описано в разделах 3.1.2 и 3.3).

Функция `std::lock()` и шаблон класса `std::lock_guard` покрывают большую часть простых случаев блокировки, по иногда этого недостаточно. Поэтому в стандартную библиотеку включен также шаблон `std::unique_lock`. Подобно `std::lock_guard`, этот шаблон класса параметризован типом мьютекса и реализует такое же управление блокировками в духе RAII, что и `std::lock_guard`, однако обладает чуть большей гибкостью.

3.2.6. Гибкая блокировка с помощью `std::unique_lock`

Шаблон `std::unique_lock` обладает большей гибкостью, чем `std::lock_guard`, потому что несколько ослабляет инварианты — экземпляр `std::unique_lock` не обязан владеть ассоциированным с ним мьютексом. Прежде всего, в качестве второго аргумента конструктору можно передавать не только объект `std::adopt_lock`, заставляющий объект управлять захватом мьютекса, но и объект `std::defer_lock`, означающий, что в момент конструирования мьютекс не должен захватываться. Захватить его можно будет позже, вызвав функцию-член `lock()` объекта `std::unique_lock` (а *не* самого мьютекса) или передав функции `std::lock()` сам объект `std::unique_lock`. Код в листинге 3.6 можно было бы с тем же успехом написать, как показало в листинге 3.9, с применением `std::unique_lock` и `std::defer_lock()` **(1)** вместо `std::lock_guard` и `std::adopt_lock`. В новом варианте столько же строк, и он эквивалентен исходному во всем, кроме одной детали, — `std::unique_lock` потребляет больше памяти и выполняется чуть дольше, чем `std::lock_guard`. Та гибкость, которую мы получаем, разрешая экземпляру `std::unique_lock` *не* владеть мьютексом, обходится не бесплатно — дополнительную информацию надо где-то хранить и обновлять.

Листинг 3.9. Применение `std::lock()` и `std::unique_guard` для реализации операции обмена

```
class some_big_object;
```

```
void swap(some_big_object& lhs,some_big_object& rhs);

class X {
private:
 some_big_object some_detail;
 std::mutex m;
public:
 X(some_big_object const& sd): some_detail(sd) {}

 friend void swap(X& lhs, X& rhs) {
  if (&lhs == &rhs)
   return;
   std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);
   std::unique_lock<std::mutex> lock_b(rhs.m, std::defer_lock);
   std::lock(lock_a, lock_b);
   swap(lhs.some_detail, rhs.some_detail);
 }
};
```

std::defer_lock оставляет мьютексы не захваченными (1)

(2) Мьютексы захватываются

В листинге 3.9 объекты `std::unique_lock` можно передавать функции `std::lock()` **(2)**, потому что в классе `std::unique_lock` имеются функции-члены `lock()`, `try_lock()` и `unlock()`. Для выполнения реальной работы они вызывают одноименные функции контролируемого мьютекса, а сами только поднимают в экземпляре `std::unique_lock` флаг, показывающий, что в данный момент этот экземпляр владеет мьютексом. Флаг необходим для того, чтобы деструктор знал, вызывать ли функцию `unlock()`. Если экземпляр *действительно* владеет мьютексом, то деструктор *должен* вызвать `unlock()`, в противном случае — *не должен*. Опросить состояние флага позволяет функция-член `owns_lock()`.

Естественно, этот флаг необходимо где-то хранить. Поэтому размер объекта `std::unique_lock` обычно больше, чем объекта `std::lock_guard`, и работает `std::unique_lock` чуть медленнее `std::lock_guard`, потому что флаг нужно проверять и обновлять. Если класс `std::lock_guard` отвечает вашим нуждам, то я рекомендую использовать его. Тем не менее, существуют ситуации, когда `std::unique_lock` лучше отвечает поставленной задаче, так как без свойственной ему дополнительной гибкости не обойтись. Один из примеров — показанный выше отложенный захват; другой — необходимость передавать владение мьютексом из одного контекста в другой.

3.2.7. Передача владения мьютексом между контекстами

Поскольку экземпляры `std::unique_lock` не владеют ассоциированными мьютексами, то можно передавать владение от одного объекта другому путем *перемещения*. В некоторых случаях передача производится автоматически, например при возврате объекта из функции, а иногда это приходится делать явно, вызывая `std::move()`. Ситуация зависит от того, является ли источник *l-значением* — именованной переменной или ссылкой на нее — или *r-значением* — временным объектом. Если источник — r-значение, то передача владения происходит автоматически, в случае же l-значение это нужно делать явно, чтобы не получилось так, что переменная потеряет владение непреднамеренно. Класс `std::unique_lock` дает пример *перемещаемого*, но не *копируемого* типа. Дополнительные сведения о семантике перемещения см. в разделе A.1.1 приложения A.

Одно из возможных применений — разрешить функции захватить мьютекс, а потом передать владение им вызывающей функции, чтобы та могла выполнить дополнительные действия под защитой того же мьютекса. Ниже приведен соответствующий пример — функция `get_lock()` захватывает мьютекс, подготавливает некоторые данные, а потом возвращает мьютекс вызывающей программе:

```
std::unique_lock<std::mutex> get_lock() {
 extern std::mutex some_mutex;
 std::unique_lock<std::mutex> lk(some_mutex);
 prepare_data();
 return lk; ← (1)
}

void process_data() {
 std::unique_lock<std::mutex> lk(get_lock()); ← (2)
 do_something();
}
```

Поскольку `lk` — автоматическая переменная, объявленная внутри функции, то ее можно возвращать непосредственно **(1)**, не вызывая `std:move()`; компилятор сам позаботится о вызове перемещающего конструктора. Затем функция `process_data()` может передать владение своему экземпляру `std::unique_lock` **(2)**, и `do_something()` может быть уверена, что подготовленные данные не были изменены каким-то другим потоком.

Обычно подобная схема применяется, когда подлежащий захвату мьютекс зависит от текущего состояния программы или от аргумента, переданного функции, которая возвращает объект `std::unique_lock`. Например, так имеет смысл делать, когда блокировка возвращается не напрямую, а является членом какого-то класса-привратника, обеспечивающего корректный доступ к разделяемым данным под защитой мьютекса. В таком случае любой доступ к данным производится через привратник, то есть предварительно необходимо получить его экземпляр (вызвав функцию, подобную `get_lock()` в примере выше), который захватит мьютекс. Затем для доступа к данным вызываются функции-члены объекта-привратника. По завершении операции привратник уничтожается, при этом мьютекс освобождается, открывая другим потокам доступ к защищенным данным. Такой объект-привратник вполне может быть перемещаемым (чтобы его можно было возвращать из функции), и тогда тот его член, в котором хранится блокировка, также должен быть перемещаемым.

Класс `std::unique_lock` также позволяет экземпляру освобождать блокировку без уничтожения. Для этого служит функция-член `unlock()`, как и в мьютексе; `std::unique_lock` поддерживает тот же базовый набор функций-членов для захвата и освобождения, что и мьютекс, чтобы его можно было использовать в таких обобщенных функциях, как `std::lock`. Наличие возможности освобождать блокировку до уничтожения объекта `std::unique_lock` означает, что освобождение можно произвести досрочно в какой-то ветке кода, если ясно, что блокировка больше не понадобится. Иногда это позволяет повысить производительность приложения, ведь, удерживая блокировку дольше необходимого, вы заставляете другие потоки впустую ждать, когда они могли бы работать.

3.2.8. Выбор правильной гранулярности блокировки

О гранулярности блокировок я уже упоминал в разделе 3.2.3: под этим понимается объем данных, защищаемых блокировкой. Мелкогранулярные блокировки защищают мало данных, крупногранулярные — много. Важно не только выбрать подходящую гранулярность, но и позаботиться о том, чтобы блокировка удерживалась не дольше, чем реально необходимо. Все мы сталкивались с ситуацией, когда очередь к кассе в супермаркете перестает двигаться из-за того, что обслуживаемый покупатель вдруг выясняет, что забыл прихватить баночку соуса, и отправляется за ней, заставляя всех ждать, или из-за того, что кассирша уже готова принять деньги, а покупатель только— только полез за кошельком. Насколько было бы проще, если бы каждый подходил к кассе только после того, как купил все необходимое и подготовился оплатить покупки.

Вот так и с потоками: если несколько потоков ждут одного ресурса (кассира), то, удерживая блокировку дольше необходимого, они заставляют другие потоки проводить в очереди больше времени (не начинайте искать баночку соуса, когда уже подошли к кассе). По возможности захватывайте мьютекс непосредственно перед доступом к разделяемым данным; старайтесь производить обработку данных, не находясь под защитой мьютекса. В частности, не начинайте длительных операций, например файловый ввод/вывод, когда удерживаете мьютекс. Ввод/вывод обычно выполняется в сотни (а то и в тысячи) раз медленнее чтения или записи того же объема данных в памяти. Поэтому если блокировка не нужна для защиты доступа к файлу, то удерживание блокировки заставляет другие потоки ждать без необходимости (так как они не могут захватить мьютекс), и тем самым вы можете свести на нет весь выигрыш от многопоточной работы.

Объект `std::unique_lock` отлично приспособлен для таких ситуаций, потому что можно вызвать его метод `unlock()`, когда программе не нужен доступ к разделяемым данным, а затем вызвать `lock()`, если доступ снова понадобится:

```
void get_and_process_data()
{
 std::unique_lock<std::mutex> my_lock(the_mutex);
 some_class data_to_process = get_next_data_chunk();
 my_lock.unlock();
 result_type result = process(data_to_process);
 my_lock.lock();
 write_result(data_to_process, result);
}
```

(1) Во время работы process() захватывать мьютекс не нужно

Снова захватить мьютекс перед записью (2) результатов

Удерживать мьютекс на время выполнения `process()` нет необходимости, поэтому мы вручную освобождаем его перед вызовом **(1)** и снова захватываем после возврата **(2)**.

Очевидно, что если один мьютекс защищает структуру данных целиком, то не только возрастает конкуренция за него, но и шансов снизить время удержания остается меньше. Поскольку под защитой одного мьютекса приходится выполнять больше операций, то и удерживать его нужно дольше. Такая двойная угроза должна вдвое усилить стремление всюду, где возможно, использовать мелкогранулярные блокировки.

Как следует из примера, выбор подходящей гранулярности определяется не только объемом защищаемых данных, но и временем удержания блокировки и тем, какие операции выполняются под ее защитой. *В общем случае блокировку следует удерживать ровно столько времени, сколько необходимо для завершения требуемых операций.* Это также означает, что длительные операции, например захват другой блокировки (даже если известно, что это не приведет к взаимоблокировке) или ожидание завершения ввода/вывода,

не следует выполнять под защитой блокировки, если только это не является абсолют пой необходимостью.

В листингах 3.6 и 3.9 мы захватывали два мьютекса для операции обмела, которая очевидно требует одновременного доступа к обоим объектам. Предположим, однако, что требуется произвести сравнение простых членов данных типа `int`. В чем разница? Копирование целых чисел — дешевая операция, поэтому вполне можно было бы скопировать данные из каждого объекта под защитой мьютекса, а затем сравнить копии. Тогда мьютекс удерживался бы минимальное время, и к тому же не пришлось бы захватывать новый мьютекс, когда один уже удерживается. В следующем листинге показам как раз такой класс `Y` и пример реализации в нем оператора сравнения на равенство.

Листинг 3.10. Поочерёдный захват мьютексов в операторе сравнения

```
class Y {
private:
 int some_detail;
 mutable std::mutex m;

 int get_detail() const {
  std::lock_guard<std::mutex> lock_a(m); ← (1)
  return some_detail;
 }
public:
 Y(int sd): some_detail(sd) {}

 friend bool operator==(Y const& lhs, Y const& rhs) {
  if (&lhs == &rhs)
   return true;
  int const lhs_value = lhs.get_detail(); ← (2)
  int const rhs_value = rhs.get_detail(); ← (3)
  return lhs_value == rhs_value; ← (4)
 }
};
```

В данном случае оператор сравнения сначала получает сравниваемые значения, вызывая функцию-член `get_detail()` **(2)**, **(3)**. Эта функция извлекает значение, находясь под защитой мьютекса **(1)**. После этого оператор сравнивает полученные значения **(4)**. Отметим, однако, что наряду с уменьшением времени удержания блокировки за счет того, что в каждый момент захвачен только один мьютекс (и, стало быть, исключена возможность взаимоблокировки), мы немного изменили семантику операции по сравнению с реализацией, в которой оба мьютекса захватываются вместе. Если оператор в листинге 3.10 возвращает `true`, то это означает лишь, что значение `lhs.some_detail` в один момент времени равно значению `rhs.some_detail` в другой момент времени. Между двумя операциями считывания значения могли измениться как угодно; например, между точками **(2)** и **(3)** программа могла обменять их местами, и тогда сравнение оказалось бы вообще бессмысленным. Таким образом, возврат оператором сравнения значения `true`, означает, что значения были равны, пусть даже ни в какой момент времени фактическое равенство не наблюдалось. Очень важно следить, чтобы такие изменения семантики операций не приводили к проблемам: *если блокировка не удерживается на протяжении всей операции, то возникает риск гонки*.

Иногда подходящего уровня гранулярности просто не существует, потому что не все операции доступа к структуре данных требуют одного и того же уровня защиты. В таком случае вместо простого класса `std::mutex` стоит поискать альтернативный механизм.

3.3. Другие средства защиты разделяемых данных

Хотя мьютексы и представляют собой наиболее общий механизм, но они не единственные игроки на поле защиты разделяемых данных. Есть и другие механизмы, обеспечивающие защиту в специальных случаях.

Один такой крайний (но на удивление распространенный) случай возникает, когда разделяемые данные нуждаются в защите от одновременного доступа только на этапе инициализации, а потом уже никакой синхронизации не требуется. Так может быть, например, потому что после инициализации данные только читаются или потому что необходимая защита обеспечивается неявно как часть операций над данными. Как бы то ни было, захватывать мьютекс после того, как данные инициализированы, совершенно не нужно, это только снизило бы производительность. Поэтому в стандарте C++ предусмотрен механизм, служащий исключительно для защиты разделяемых данных во время инициализации.

3.3.1. Защита разделяемых данных во время инициализации

Предположим, имеется разделяемый ресурс, конструирование которого обходится настолько дорого, что мы хотим делать это, лишь когда действительно возникает необходимость; быть может, конструктор открывает базу данных или выделяет очень много памяти. Такая *отложенная инициализация* часто встречает в однопоточных программах — всякая операция, нуждающаяся в ресурсе, сначала проверяет, инициализирован ли он, и, если нет, выполняет инициализацию:

```
std::shared_ptr<some_resource> resource_ptr;
void foo() {
 if (!resource_ptr) {
  resource_ptr.reset(new some_resource); ← (1)
 }
 resource_ptr->do_something();
}
```

Если сам разделяемый ресурс безопасен относительно одновременного доступа, то при переходе к многопоточной реализации единственная нуждающаяся в защите часть — инициализация **(1)**, однако наивный подход, показанный в листинге ниже, может привести к ненужной сериализации использующих ресурс потоков. Дело в том, что каждый поток должен ждать освобождения мьютекса, чтобы проверить, был ли ресурс уже инициализирован.

Листинг 3.11. Потокобезопасная отложенная инициализация с помощью мьютекса

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex; ←┐ В этой точке все потоки
                            │ сериализуются
void foo() {
 std::unique_lock<std::mutex> lk(resource_mutex);
 if (!resource_ptr) {
  resource_ptr.reset(new some_resource); ←┐ в защите нуж-
 }                                        │ дается только
```

```
 lk.unlock();
 resource_ptr->do_something();
}
```

Этот код встречается настолько часто, а ненужная сериализация вызывает столько проблем, что многие предпринимали попытки найти более приемлемое решение, в том числе печально известный паттерн *блокировка с двойной проверкой* (Double-Checked Locking): сначала указатель читается без захвата мьютекса **(1)** (см. код ниже), а захват производится, только если оказалось, что указатель равен `NULL`. Затем, когда мьютекс захвачен **(2)**, указатель проверяется *еще раз* (отсюда и слова «двойная проверка») на случай, если какой-то другой поток уже выполнил инициализацию в промежутке между первой проверкой и захватом мьютекса:

```
void undefined_behaviour_with_double_checked_locking() {
 if (!resource_ptr)                                  ← (1)
 {
  std::lock_guard<std::mutex> lk(resource_mutex);
  if (!resource_ptr)                                 ← (2)
  {
   resource_ptr.reset(new some_resource);            ← (3)
  }
 }
 resource_ptr->do_something();                       ← (4)
}
```

«Печально известным» я назвал этот паттерн не без причины: он открывает возможность для крайне неприятного состояния гонки, потому что чтение без мьютекса **(1)** не синхронизировано с записью в другом потоке с уже захваченным мьютексом **(3)**. Таким образом, возникает гонка, угрожающая не самому указателю, а объекту, на который он указывает; даже если один поток видит, что указатель инициализирован другим потоком, он может не увидеть вновь созданного объекта `some_resource`, и, следовательно, вызов `do_something()` **(4)** будет применен не к тому объекту, что нужно. Такого рода гонка в стандарте C++ называется *гонкой за данными* (*data race*), она отнесена к категории *неопределенного поведения*.

Комитет по стандартизации C++ счел этот случай достаточно важным, поэтому в стандартную библиотеку включен класс `std::once_flag` и шаблон функции `std::call_once`. Вместо того чтобы захватывать мьютекс и явно проверять указатель, каждый поток может просто вызвать функцию `std::call_once`, твердо зная, что к моменту возврата из нее указатель уже инициализирован каким-то потоком (без нарушения синхронизации). Обычно издержки, сопряженные с использованием `std::call_once`, ниже, чем при явном применении мьютекса, поэтому такое решение следует предпочесть во всех случаях, когда оно не противоречит требованиям задачи. В примере ниже код из листинга 3.11 переписан с использованием `std::call_once`. В данном случае инициализация производится путем вызова функции, но ничто не мешает завести для той же цели класс, в котором определен оператор вызова. Как и большинство функций в стандартной библиотеке, принимающих в качестве аргументов функции или предикаты, `std::call_once` работает как с функциями, так и с объектами, допускающими вызов.

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;← (1)
```

```
void init_resource() {
 resource_ptr.reset(new some_resource);
}
```

| Инициализация производится

```
void foo() { ←┘ ровно один раз
 std::call_once(resource_flag, init_resource);
 resource_ptr->do_something();
}
```

Здесь переменная типа `std::once_flag` **(1)** и инициализируемый объект определены в области видимости пространства имен, но `std::call_once()` вполне можно использовать и для отложенной инициализации членов класса, как показано в следующем листинге.

Листинг 3.12. Потокобезопасная отложенная инициализация члена класса с помощью функции `std::call_once()`

```
class X {
private:
 connection_info connection_details;
 connection_handle connection;
 std::once_flag connection_init_flag;

 void open_connection() {
  connection = connection_manager.open(connection_details);
 }

public:
 X(connection_info const& connection_details_):
  connection_details(connection_details_) {}

 void send_data(data_packet const& data)← (1)
 {
  std::call_once(
   connection_init_flag, &X::open_connection, this);←┐
  connection.send_data(data);                       │
 }                                                  │
 data_packet receive_data() { ← (3)                 │
  std::call_once(                                   │
   connection_init_flag, &X::open_connection, 2)    (2)
   this);                                          ←┘
  return connection.receive_data();
 }
};
```

В этом примере инициализация производится либо при первом обращении к `send_data()` **(1)**, либо при первом обращении к `receive_data()` **(3)**. Поскольку данные инициализируются функцией-членом `open_connection()`, то требуется передавать также указатель `this`. Как и во всех функциях из стандартной библиотеки, которые принимают объекты, допускающие вызов, (например, конструктор `std::thread` и функция `std::bind()`), это делается путем передачи `std::call_once()` дополнительного аргумента **(2)**.

Следует отметить, что, как и в случае `std:mutex`, объекты типа `std::once_flag` нельзя ни копировать, ни перемещать, поэтому, если вы собираетесь использовать их как члены

классы, то соответствующие конструкторы придется определить явно (если это необходимо).

Возможность гонки при инициализации возникает, в частности, при объявлении локальной переменной с классом памяти `static`. По определению, инициализация такой переменной происходит, когда поток управления программы первый раз проходит через ее объявление. Но если функция вызывается в нескольких потоках, то появляется потенциальная возможность гонки за то, кто определит переменную первым. Во многих компиляторах, выпущенных до утверждения стандарта C++11, эта гонка действительно приводит к проблемам, потому что любой из нескольких потоков, полагая, что успел первым, может попытаться инициализировать переменную. Может также случиться, что некоторый поток попытается использовать переменную после того, как инициализация началась в другом потоке, но до того, как она закончилась. В C++11 эта проблема решена: по определению, инициализация производится ровно в одном потоке, и никакому другому потоку не разрешено продолжать выполнение, пока инициализация не завершится, поэтому потоки конкурируют лишь за право выполнить инициализацию первым, ничего более серьёзного случиться не может. Это свойство можно использовать как альтернативу функции `std::call_once`, когда речь идет об инициализации единственной глобальной переменной:

```
class my_class;
 my_class& get_my_class_instance() {
 static my_class instance; ←┐ Гарантируется, что инициализация
 return instance;           (1) потокобезопасна
}
```

Теперь несколько потоков могут вызывать функцию `get_my_class_instance()` **(1)**, не опасаясь гонки при инициализации.

Защита данных только на время инициализации — частный случай более общего сценария: доступ к редко обновляемой структуре данных. Обычно к такой структуре обращаются для чтения, когда ни о какой синхронизации можно не беспокоиться. Но иногда требуется обновить данные в ней. Нам необходим такой механизм защиты, который учитывал бы эти особенности.

3.3.2. Защита редко обновляемых структур данных

Рассмотрим таблицу, в которой хранится кэш записей DNS, необходимых для установления соответствия между доменными именами и IP-адресами. Как правило, записи DNS остаются неизменными в течение длительного времени — зачастую многих лет. Новые записи, конечно, добавляются — скажем, когда открывается новый сайт — но на протяжении всей своей жизни обычно не меняются. Периодически необходимо проверять достоверность данных в кэше, но и тогда обновление требуется, лишь если данные действительно изменились.

Но хотя обновления происходят редко, они все же случаются, и если к кэшу возможен доступ со стороны нескольких потоков, то необходимо обеспечить надлежащую защиту, чтобы ни один поток, читающий кэш, не увидел наполовину обновленной структуры данных. Если структура данных не специализирована для такого способа использования (как описано в главах 6 и 7), то поток, который хочет обновить данные, должен получить монопольный доступ к структуре на все время выполнения операции. После того как операция обновления завершится, структуру данных снова смогут одновременно читать несколько потоков.

Использование `std::mutex` для защиты такой структуры данных излишне пессимистично, потому что при этом исключается даже возможность одновременного чтения, когда никакая модификация не производится. Нам необходим какой-то другой вид мьютекса. Такой мьютекс есть, и обычно его называют мьютексом *чтения-записи* (reader-writer mutex), потому что он допускает два режима: монопольный доступ со стороны одного «потока-писателя» и параллельный доступ со стороны нескольких «потоков-читателей».

В новой стандартной библиотеке C++ такой мьютекс не предусмотрен, хотя комитету и было подано предложение[6]. Поэтому в этом разделе мы будем пользоваться реализацией из библиотеки Boost, которая основана на отвергнутом предложении. В главе 8 вы увидите, что использование такого мьютекса — не панацея, а его производительность зависит от количества участвующих процессоров и относительного распределения нагрузки между читателями и писателями. Поэтому важно профилировать работу программу в целевой системе и убедиться, что добавочная сложность действительно дает какой-то выигрыш.

Итак, вместо `std::mutex` мы воспользуемся для синхронизации объектом `boost::shared_mutex`. При выполнении обновления мы будем использовать для захвата мьютекса шаблоны `std::lock_guard<boost::shared_mutex>` и `std::unique_lock<boost::shared_mutex>`, параметризованные классом `boost::shared_mutex`, а не `std::mutex`. Они точно так же гарантируют монопольный доступ. Те же потоки, которым не нужно обновлять структуру данных, могут воспользоваться классом `boost::shared_lock<boost::shared_mutex>` для получения *разделяемого* доступа. Применяется он так же, как `std::unique_lock`, но в семантике имеется одно важное отличие: несколько потоков могут одновременно получить разделяемую блокировку на один и тот же объект `boost::shared_mutex`. Однако если какой-то поток уже захватил разделяемую блокировку, то любой поток, который попытается захватить монопольную блокировку, будет приостановлен до тех пор, пока все прочие потоки не освободят свои блокировки. И наоборот, если какой-то поток владеет монопольной блокировкой, то никакой другой поток не сможет получить ни разделяемую, ни монопольную блокировку, пока первый поток не освободит свою.

В листинге ниже приведена реализация простого DNS-кэша, в котором данные хранятся в контейнере `std::map`, защищенном с помощью `boost::shared_mutex`.

Листинг 3.13. Защита структуры данных с помощью `boost::shared_mutex`

```
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>


class dns_entry;
class dns_cache {
 std::map<std::string, dns_entry> entries;
 mutable boost::shared_mutex entry_mutex;
public:
 dns_entry find_entry(std::string const& domain) const {
  boost::shared_lock<boost::shared_mutex> lk(entry_mutex); ← (1)
  std::map<std::string, dns_entry>::const_iterator const it =
   entries.find(domain);
  return (it == entries.end()) ? dns_entry() : it->second;
 }
```

```
 void update_or_add_entry(std::string const& domain,
  dns_entry const& dns_details) {
  std::lock_guard<boost::shared_mutex> lk(entry_mutex); ← (2)
  entries[domain] = dns_details;
 }
};
```

В листинге 3.13 в функции `find_entry()` используется объект `boost::shared_lock<>`, обеспечивающий разделяемый доступ к данным для чтения **(1)**; следовательно, ее можно спокойно вызывать одновременно из нескольких потоков. С другой стороны, в функции `update_or_add_entry()` используется объект `std::lock_guard<>`, который обеспечивает монопольный доступ на время обновления таблицы **(2)**, и, значит, блокируются не только другие потоки, пытающиеся одновременно выполнить `update_or_add_entry()`, но также потоки, вызывающие `find_entry()`.

3.3.3. Рекурсивная блокировка

Попытка захватить `std::mutex` в потоке, который уже владеет им, является ошибкой и приводит к *неопределенному поведению*. Однако бывают случаи, когда потоку желательно повторно захватывать один и тот же мьютекс, не освобождая его предварительно. Для этого в стандартной библиотеке C++ предусмотрен класс `std::recursive_mutex`. Работает он аналогично `std::mutex`, но с одним отличием: один и тот же поток может многократно захватывать данный мьютекс. Но перед тем как этот мьютекс сможет захватить другой поток, его нужно освободить столько раз, сколько он был захвачен. Таким образом, если функция `lock()` вызывалась три раза, то и функцию unlock() нужно будет вызвать трижды. При правильном использовании `std::lock_guard<std::recursive_mutex>` и `std::unique_lock<std::recursive_mutex>` это гарантируется автоматически.

Как правило, программу, в которой возникает необходимость в рекурсивном мьютексе, лучше перепроектировать. Типичный пример использования рекурсивного мьютекса возникает, когда имеется класс, к которому могут обращаться несколько потоков, так что для защиты его данных необходим мьютекс. Каждая открытая функция-член захватывает мьютекс, что-то делает, а затем освобождает его. Но бывает, что одна открытая функция-член вызывает другую, и в таком случае вторая также попытается захватить мьютекс, что приведет к неопределенному поведению. Тогда, чтобы решить проблему по-быстрому, обычный мьютекс заменяют рекурсивным. Это позволит второй функции захватить мьютекс и продолжить работу.

Однако такое решение *не рекомендуется*, потому что является признаком небрежного и плохо продуманного проектирования. В частности, при работе под защитой мьютекса часто нарушаются инварианты класса, а это означает, что вторая функция-член должна правильно работать даже в условиях, когда некоторые инварианты не выполняются. Обычно лучше завести новую закрытую функцию-член, которая вызывается из обеих открытых и не захватывает мьютекс (то есть предполагает, что мьютекс уже захвачен). Затем следует тщательно продумать, при каких условиях эта новая функция может вызываться и в каком состоянии будут при этом находиться данные.

3.4. Резюме

В этой главе мы рассмотрели, к каким печальным последствиям могут приводить проблематичные гонки, когда возможно разделение данных между потоками, и как с помощью класса `std::mutex` и тщательного проектирования интерфейса этих неприятностей можно избежать. Мы видели, что мьютексы — не панацея, поскольку им свойственны собственные проблемы в виде взаимоблокировки, хотя стандартная библиотека C++ содержит средство, позволяющее избежать их — класс `std::lock()`. Затем мы обсудили другие способы избежать взаимоблокировок и кратко обсудили передачу владения блокировкой и вопросы, касающиеся выбора подходящего уровня гранулярности блокировки. Наконец, я рассказал об альтернативных механизмах защиты данных, применяемых в специальных случаях: `std::call_once()` и `boost::shared_mutex`.

А вот чего мы пока не рассмотрели, так это ожидание поступления входных данных из других потоков. Наш потокобезопасный стек просто возбуждает исключение при попытке извлечения из пустого стека. Поэтому если один поток хочет дождаться, пока другой поток поместит в стек какие-то данные (а это, собственно, и есть основное назначение потокобезопасного стека), то должен будет раз за разом пытаться извлечь значение, повторяя попытку в случае исключения. Это приводит лишь к бесцельной трате процессорного времени на проверку; более того, такая повторяющаяся проверка может замедлить работу программы, поскольку не дает выполняться другим потокам. Нам необходим какой-то способ, который позволил бы одному потоку ждать завершения операции в другом потоке, не потребляя процессорное время. В главе 4, которая опирается на рассмотренные выше средства защиты разделяемых данных, мы познакомимся с различными механизмами синхронизации операций между потоками в C++, а в главе 6 увидим, как с помощью этих механизмов можно строить более крупные структуры данных, допускающие повторное использование.

Глава 4.
Синхронизация параллельных операций

В этой главе:

- Ожидание события.
- Ожидание однократного события с будущими результатами
- Ожидание с ограничением по времени.
- Использование синхронизации операций для упрощения программы.

В предыдущей главе мы рассмотрели различные способы защиты данных, разделяемых между потоками. Но иногда требуется не только защитить данные, но и синхронизировать действия, выполняемые в разных потоках. Например, возможно, что одному потоку перед тем как продолжить работу, нужно дождаться, пока другой поток завершит какую-то операцию. В общем случае, часто возникает ситуация, когда поток должен ожидать какого-то события или истинности некоторого условия. Конечно, это можно сделать, периодически проверяя разделяемый флаг «задача завершена» или что-то в этом роде, но такое решение далеко от идеала. Необходимость в синхронизации операций — настолько распространенный сценарий, что в стандартную библиотеку C++ включены специальные механизмы для этой цели — *условные переменные* и *будущие результаты* (future).

В этой главе мы рассмотрим, как реализуется ожидание событий с помощью условных переменных и будущих результатов и как ими можно воспользоваться, чтобы упростить синхронизацию операций.

4.1. Ожидание события или иного условия

Представьте, что вы едете на поезде ночью. Чтобы не пропустить свою станцию, можно не спать всю ночь и читать названия всех пунктов, где поезд останавливается. Так вы, конечно, не проедете мимо, но сойдете с поезда сильно уставшим. Есть и другой способ — заранее посмотреть в расписании, когда поезд прибывает в нужный вам пункт, поставить будильник и улечься спать. Так вы тоже свою остановку не пропустите, но если поезд задержится в пути, то проснётесь слишком рано. И еще одно — если в будильнике сядут батарейки, то вы можете проспать и проехать мимо нужной станции. В идеале хотелось бы, чтобы кто-то или что-то разбудило вас, когда поезд подъедет к станции, — не раньше и не позже.

Какое отношение всё это имеет к потокам? Самое непосредственное — если один поток хочет дождаться, когда другой завершит некую операцию, то может поступить несколькими способами. Во-первых, он может просто проверять разделяемый флаг (защищенный мьютексом), полагая, что второй поток поднимет этот флаг, когда завершит свою операцию. Это расточительно но двум причинам: на опрос флага уходит процессорное время, и мьютекс, захваченный ожидающим потоком, не может быть захвачен никаким другим потоком. То и другое работает против ожидающего потока, поскольку ограничивает ресурсы, доступные потоку, которого он так ждет, и даже не дает ему возможность поднять флаг, когда работа будет завершена. Это решение сродни бодрствованию всю ночь, скрашиваемому разговорами с машинистом: он вынужден вести поезд медленнее, потому что вы его постоянно отвлекаете, и, значит, до пункта назначения вы доберетесь позже. Вот и ожидающий поток потребляет ресурсы, которые пригодились бы другим потокам, в результате чего ждет дольше, чем необходимо.

Второй вариант — заставить ожидающий поток спать между проверками с помощью функции `std::this_thread::sleep_for()` (см. раздел 4.3):

```
bool flag;
std::mutex m;

void wait_for_flag() {
 std::unique_lock<std::mutex> lk(m); ← (1) Освободить мьютекс
 while (!flag) {
  lk.unlock(); ← (2) Спать 100 мс
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  lk.lock();    ← (3) Снова захватить мьютекс
 }
}
```

В этом цикле функция освобождает мьютекс **(1)** перед тем, как заснуть **(2)**, и снова захватывает его, проснувшись, **(3)**, оставляя другому потоку шанс захватить мьютекс и поднять флаг.

Это уже лучше, потому что во время сна поток не расходует процессорное время. Но трудно выбрать подходящий промежуток времени. Если он слишком короткий, то поток все равно впустую тратит время на проверку; если слишком длинный — то поток будет спать и после того, как ожидание завершилось, то есть появляется ненужная задержка. Редко бывает так, что слишком длительный сон прямо влияет на работу программу, но в динамичной игре это может привести к пропуску кадров, а в приложении реального времени — к исчерпанию

выделенного временного кванта.

Третий — и наиболее предпочтительный - способ состоит в том, чтобы воспользоваться средствами из стандартной библиотеки C++, которые позволяют потоку ждать события. Самый простой механизм ожидания события, возникающего в другом потоке (например, появления нового задания в упоминавшемся выше конвейере), дают *условные переменные*. Концептуально условная переменная ассоциирована с каким-то событием или иным *условием*, причём один или несколько потоков могут *ждать*, когда это условие окажется выполненным. Если некоторый поток решит, что условие выполнено, он может *известить* об этом один или несколько потоков, ожидающих условную переменную, в результате чего они возобновят работу.

4.1.1. Ожидание условия с помощью условных переменных

Стандартная библиотека C++ предоставляет не одну, а *две* реализации условных переменных: `std::condition_variable` и `std::condition_variable_any`. Оба класса объявлены в заголовке `<condition_variable>`. В обоих случаях для обеспечения синхронизации необходимо взаимодействие с мьютексом; первый класс может работать только с `std::mutex`, второй — с любым классом, который отвечает минимальным требованиям к «мьютексоподобию», отсюда и суффикс `_any`. Поскольку класс `std::condition_variable_any` более общий, то его использование может обойтись дороже с точки зрения объема потребляемой памяти, производительности и ресурсов операционной системы. Поэтому, если дополнительная гибкость не требуется, то лучше ограничиться классом `std::condition_variable`.

Ну и как же воспользоваться классом `std::condition_variable` в примере, упомянутом во введении, — как сделать, чтобы поток, ожидающий работу, спал, пока не поступят данные? В следующем листинге приведён пример реализации с использованием условной переменной.

Листинг 4.1. Ожидание данных с помощью `std::condition_variable`

```
std::mutex mut;
std::queue<data_chunk> data_queue; ← (1)
std::condition_variable data_cond;

void data_preparation_thread() {
 while (more_data_to_prepare()) {
  data_chunk const data = prepare_data();
  std::lock_guard<std::mutex> lk(mut);
  data_queue.push(data);  ← (2)
  data_cond.notify_one(); ← (3)
 }
}

void data_processing_thread() {
 while(true) {
  std::unique_lock<std::mutex> lk(mut); ← (4)
  data_cond.wait(
   lk, []{ return !data_queue.empty(); }); ← (5)
```

```
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock(); ← (6)
    process(data);
    if (is_last_chunk(data))
     break;
   }
  }
```

Итак, мы имеем очередь **(1)**, которая служит для передачи данных между двумя потоками. Когда данные будут готовы, поток, отвечающий за их подготовку, помещает данные в очередь, предварительно захватив защищающий ее мьютекс с помощью `std::lock_guard`. Затем он вызывает функцию-член `notify_one()` объекта `std::condition_variable`, чтобы известить ожидающий поток (если таковой существует) **(3)**.

По другую сторону забора находится поток, обрабатывающий данные. Он в самом начале захватывает мьютекс, но с помощью `std::unique_lock`, а не `std::lock_guard` **(4)** — почему, мы скоро увидим. Затем поток вызывает функцию-член `wait()` объекта `std::condition_variable`, передавая ей объект-блокировку и лямбда-функцию, выражающую ожидаемое условие **(5)**. Лямбда-функции — это нововведение в C++11, они позволяют записать анонимную функцию как часть выражения и идеально подходят для задания предикатов для таких стандартных библиотечных функций, как `wait()`. В данном случае простая лямбда-функция `[]{ return !data_queue.empty(); }` проверяет, что очередь `data_queue` не пуста (вызывая ее метод `empty()`), то есть что в ней имеются данные для обработки. Подробнее лямбда-функции описаны в разделе А.5 приложения А.

Затем функция `wait()` проверяет условие (вызывая переданную лямбда-функцию) и возвращает управление, если оно выполнено (то есть лямбда-функция вернула `true`). Если условие не выполнено (лямбда-функция вернула `false`), то `wait()` освобождает мьютекс и переводит поток в состояние ожидания. Когда условная переменная получит извещение, отправленное потоком подготовки данных с помощью `notify_one()`, поток обработки пробудится, вновь захватит мьютекс и еще раз проверит условие. Если условие выполнено, то `wait()` вернет управление, причём мьютекс в этот момент будет захвачен. Если же условие не выполнено, то поток снова освобождает мьютекс и возобновляет ожидание. Именно поэтому нам необходим `std::unique_lock`, а не `std::lock_guard` — ожидающий поток должен освобождать мьютекс, когда находится в состоянии ожидания, и захватывать его но выходе из этого состояния, а `std::lock_guard` такой гибкостью не обладает. Если бы мьютекс оставался захваченным в то время, когда поток обработки спит, поток подготовки данных не смог бы захватить его, чтобы поместить новые данные в очередь, а, значит, ожидаемое условие никогда не было бы выполнено.

В листинге 4.1 используется простая лямбда-функция **(5)**, которая проверяет, что очередь не пуста. Однако с тем же успехом можно было бы передать любую функцию или объект, допускающий вызов. Если функция проверки условия уже существует (быть может, она сложнее показанного в примере простенького теста), то передавайте ее напрямую — нет никакой необходимости обертывать ее лямбда-функцией. Внутри `wait()` условная переменная может проверять условие многократно, но всякий раз это делается после захвата мьютекса, и, как только функция проверки условия вернет `true` (и лишь в этом случае), `wait()` возвращает управление вызывающей программе. Ситуация, когда ожидающий поток

захватывает мьютекс и проверяет условие не в ответ на извещение от другого потока, называется *ложным пробуждением* (spurious wake). Поскольку количество и частота ложных пробуждений по определению недетерминированы, нежелательно использовать для проверки условия функцию с побочными эффектами. В противном случае будьте готовы к тому, что побочный эффект может возникать более одного раза.

Присущая `std::unique_lock` возможность освобождать мьютекс используется не только при обращении к `wait()`, но и непосредственно перед обработкой поступивших данных **(6)**. Обработка может занимать много времени, а, как было отмечено в главе 3, удерживать мьютекс дольше необходимого неразумно.

Применение очереди для передачи данных между потоками (как в листинге 4.1) — весьма распространенный прием. При правильной реализации синхронизацию можно ограничить только самой очередью, что уменьшает количество потенциальных проблем и состояний гонки. Поэтому покажем, как на основе листинга 4.1 построить обобщенную потокобезопасную очередь.

4.1.2. Потокобезопасная очередь на базе условных переменных

Приступая к проектированию обобщенной очереди, стоит потратить некоторое время на обдумывание того, какие понадобятся операции. Именно так мы подходили к разработке потокобезопасного стека в разделе 3.2.3. Возьмем в качестве образца адаптер контейнера `std::queue<>` из стандартной библиотеки C++, интерфейс которого показан в листинге ниже.

Листинг 4.2. Интерфейс класса `std::queue`

```
template <class T, class Container = std::deque<T>>
class queue {
public:
 explicit queue(const Container&);
 explicit queue(Container&& = Container());

 template <class Alloc> explicit queue(const Alloc&);
 template <class Alloc> queue(const Container&, const Alloc&);
 template <class Alloc> queue(Container&&, const Alloc&);
 template <class Alloc> queue(queue&&, const Alloc&);

 void swap(queue& q);

 bool empty() const;
 size_type size() const;

 T& front();
 const T& front() const;
 T& back();
 const T& back() const;

 void push(const T& x);
 void push(T&& x);
 void pop();
 template <class... Args> void emplace(Args&&... args);
```

```
};
```

Если не обращать внимания на конструирование, присваивание и обмен, то останется три группы операций: опрос состояния очереди в целом (`empty()` и `size()`), опрос элементов очереди (`front()` и `back()`) модификация очереди (`push()`, `pop()` и `emplace()`). Ситуация аналогична той, что мы видели в разделе 3.2.3 для стека, поэтому возникают те же — внутренне присущие интерфейсу — проблемы с гонкой. Следовательно, `front()` и `pop()` необходимо объединить в одной функции — точно так же, как мы постудили с `top()` и `pop()` в случае стека. Но в коде в листинге 4.1 есть дополнительный нюанс: если очередь используется для передачи данных между потоками, то поток-получатель часто будет ожидать поступления данных. Поэтому включим два варианта `pop()`: `try_pop()` пытается извлечь значение из очереди, но сразу возвращает управление (с указанием ошибки), если в очереди ничего не было, а `wait_and_pop()` ждет, когда появятся данные. Взяв за образец сигнатуры функций из примера стека, представим интерфейс в следующем виде:

Листинг 4.3. Интерфейс класса `threadsafe_queue`

```
#include <memory>

template<typename T>
class threadsafe_queue {
public:
 threadsafe_queue();
 threadsafe_queue(const threadsafe_queue&);
 threadsafe_queue& operator=(
  const threadsafe_queue&) = delete; ←┐ Для простоты
 void push(T new_value);              | запрещаем присваивание

 bool try_pop(T& value);       ← (1)
 std::shared_ptr<T> try_pop(); ← (2)

 void wait_and_pop(T& value);
 std::shared_ptr<T> wait_and_pop();

 bool empty() const;
};
```

Как и в случае стека, мы для простоты уменьшили число конструкторов и запретили присваивание. И, как и раньше, предлагаем по два варианта функций `try_pop()` и `wait_for_pop()`. Первый перегруженный вариант `try_pop()` **(1)** сохраняет извлеченное значение в переданной по ссылке переменной, а возвращаемое значение использует для индикации ошибки: оно равно `true`, если значение получено, и `false` — в противном случае (см. раздел A.2). Во втором перегруженном варианте **(2)** так поступить нельзя, потому что возвращаемое значение — это данные, извлеченные из очереди. Однако же можно возвращать указатель `NULL`, если в очереди ничего не оказалось.

Ну и как же всё это соотносится с листингом 4.1? В следующем листинге показано, как перенести оттуда код в методы `push()` и `wait_and_pop()`.

Листинг 4.4. Реализация функций `push()` и `wait_and_pop()` на основе кода из листинга 4.1

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue {
private:
 std::mutex mut;
 std::queue<T> data_queue;
 std::condition_variable data_cond;
public:
 void push(T new_value) {
  std::lock_guard<std::mutex> lk(mut);
  data_queue.push(new_value);
  data_cond.notify_one();
 }

 void wait_and_pop(T& value) {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{return !data_queue.empty();});
  value = data_queue.front();
  data_queue.pop();
 }
};

threadsafe_queue<data_chunk> data_queue; ← (1)

void data_preparation_thread() {
 while (more_data_to_prepare()) {
  data_chunk const data = prepare_data();
  data_queue.push(data); ← (2)
 }
}

void data_processing_thread() {
 while (true) {
  data_chunk data;
  data_queue.wait_and_pop(data); ← (3)
  process(data);
  if (is_last_chunk(data))
   break;
 }
}
```

Теперь мьютекс и условная переменная находятся в экземпляре `threadsafe_queue`, поэтому не нужно ни отдельных переменных **(1)**, ни внешней синхронизации при обращении к функции `push()` **(2)**. Кроме того, `wait_and_pop()` берет на себя заботу об ожидании условной переменной **(3)**.

Второй перегруженный вариант `wait_and_pop()` тривиален, а остальные функции можно почти без изменений скопировать из кода стека в листинге 3.5. Ниже приведена окончательная реализация.

Листинг 4.5. Полное определение класса потокобезопасной очереди на базе условных

переменных

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue {
private:
 mutable std::mutex mut;
 std::queue<T> data_queue;
 std::condition_variable data_cond;

public:
 threadsafe_queue() {}
 threadsafe_queue(threadsafe_queue const& other) {
  std::lock_guard<std::mutex> lk(other.mut);
  data_queue = other.data_queue;
 }

 void push(T new_value) {
  std::lock_guard<std::mutex> lk(mut);
  data_queue.push(new_value);
  data_cond.notify_one();
 }

 void wait_and_pop(T& value) {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{ return !data_queue.empty(); });
  value = data_queue.front();
  data_queue.pop();
 }

 std::shared_ptr<T> wait_and_pop() {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{ return !data_queue.empty(); });
  std::shared_ptr<T>
   res(std::make_shared<T>(data_queue.front()));
  data_queue.pop();
  return res;
 }

 bool try_pop(T& value) {
  std::lock_guard<std::mutex> lk(mut);
  if (data_queue.empty())
   return false;
  value = data_queue.front();
  data_queue.pop();
  return true;
 }

 std::shared_ptr<T> try_pop() {
  std::lock_guard<std::mutex> lk(mut);
  if (data_queue.empty())
   return std::shared_ptr<T>();
```

(1) Мьютекс должен быть изменяемым

```
    std::shared_ptr<T>
     res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
   }

   bool empty() const {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
   }
  };
```

Хотя `empty()` — константная функция-член, а параметр копирующего конструктора — `const`-ссылка, другие потоки могут хранить неконстантные ссылки на объект и вызывать изменяющие функции-члены, которые захватывают мьютекс. Поэтому захват мьютекса — это изменяющая операция, следовательно, член `mut` необходимо пометить как `mutable` **(1)**, чтобы его можно было захватить в функции `empty()` и в копирующем конструкторе.

Условные переменные полезны и тогда, когда есть несколько потоков, ожидающих одного события. Если потоки используются для разделения работы и, следовательно, на извещение должен реагировать только один поток, то применима точно такая же структура программы, как в листинге 4.1; нужно только запустить несколько потоков обработки данных. При поступлении новых данных функция `notify_one()` разбудит только один поток, который проверяет условие внутри `wait()`, и этот единственный поток вернет управление из `wait()` (в ответ на помещение нового элемента в очередь `data_queue`). Заранее нельзя сказать, какой поток получит извещение и есть ли вообще ожидающие потоки (не исключено, что все они заняты обработкой ранее поступивших данных).

Альтернативный сценарий — когда несколько потоков ожидают одного события, и отреагировать должны все. Так бывает, например, когда инициализируются разделяемые данные, и все работающие с ними потоки должны ждать, пока инициализация завершится (хотя для этого случая существуют более подходящие механизмы, см. раздел 3.3.1 главы 3), или когда потоки должны ждать обновления разделяемых данных, например, в случае периодической повторной инициализации. В таких ситуациях поток, отвечающий за подготовку данных, может вызвать функцию-член `notify_all()` условной переменной вместо `notify_one()`. Эта функция извещает *все* потоки, ожидающие внутри функции `wait()`, о том, что они должны проверить ожидаемое условие.

Если ожидающий поток собирается ждать условия только один раз, то есть после того как оно станет истинным, он не вернется к ожиданию той же условной переменной, то лучше применить другой механизм синхронизации. В особенности это относится к случаю, когда ожидаемое условие — доступность каких-то данных. Для такого сценария больше подходят так называемые *будущие результаты* (future).

4.2. Ожидание одноразовых событий с помощью механизма будущих результатов

Предположим, вы летите самолетом в отпуск за границу. Вы приехали в аэропорт, прошли регистрацию и прочие процедуры, но должны ждать объявления о посадке — быть может, несколько часов. Можно, конечно, найти себе занятие — например, почитать книжку, побродить в Интернете, поесть в кафе за бешеные деньги, но суть от этого не меняется: вы ждете сигнала о том, что началась посадка в самолет. И есть еще одна особенность — данный рейс вылетает всего один раз; в следующий отпуск вы будете ждать посадки на другой рейс.

В стандартной библиотеке C++ такие одноразовые события моделируются с помощью *будущего результата*. Если поток должен ждать некоего одноразового события, то он каким-то образом получает представляющий его объект-будущее. Затем поток может периодически в течение очень короткого времени ожидать этот объект-будущее, проверяя, произошло ли событие (посмотреть на табло вылетов), а между проверками заниматься другим делом (вкушать в кафе аэропортовскую пищу по несуразным ценам). Можно поступить и иначе — выполнять другую работу до тех пор, пока не наступит момент, когда без наступления ожидаемого события двигаться дальше невозможно, и вот тогда ждать *готовности* будущего результата. С будущим результатом могут быть ассоциированы какие-то данные (например, номер выхода в объявлении на посадку), но это необязательно. После того как событие произошло (то есть будущий результат *готов*), сбросить объект-будущее в исходное состояние уже невозможно.

В стандартной библиотеке C++ есть две разновидности будущих результатов, реализованные в форме двух шаблонов классов, которые объявлены в заголовке `<future>`: *уникальные будущие результаты* (`std::future<>`) и *разделяемые будущие результаты* (`std::shared_future<>`). Эти классы устроены по образцу `std::unique_ptr` и `std::shared_ptr`. На одно событие может ссылаться только один экземпляр `std::future`, но несколько экземпляров `std::shared_future`. В последнем случае все экземпляры оказываются *готовы* одновременно и могут обращаться к ассоциированным с событием данным. Именно из-за ассоциированных данных будущие результаты представлены шаблонами, а не обычными классами; точно так же шаблоны `std::unique_ptr` и `std::shared_ptr` параметризованы типом ассоциированных данных. Если ассоциированных данных нет, то следует использовать специализации шаблонов `std::future<void>` и `std::shared_future<void>`. Хотя будущие результаты используются как механизм межпоточной коммуникации, сами по себе они не обеспечивают синхронизацию доступа. Если несколько потоков обращаются к единственному объекту-будущему, то они должны защитить доступ с помощью мьютекса или какого-либо другого механизма синхронизации, как описано в главе 3. Однако, как будет показано в разделе 4.2.5, каждый из нескольких потоков может работать с собственной копией `std::shared_future<>` безо всякой синхронизации, даже если все они ссылаются на один и тот же асинхронно получаемый результат.

Самое простое одноразовое событие — это результат вычисления, выполненного в фоновом режиме. В главе 2 мы видели, что класс `std::thread` не предоставляет средств для возврата вычисленного значения, и я обещал вернуться к этому вопросу в главе 4. Исполняю

обещание.

4.2.1. Возврат значения из фоновой задачи

Допустим, вы начали какое-то длительное вычисление, которое в конечном итоге должно дать полезный результат, но пока без него можно обойтись. Быть может, вы нашли способ получить ответ на «Главный возрос жизни, Вселенной и всего на свете» из книги Дугласа Адамса[7]. Для вычисления можно запустить новый поток, но придётся самостоятельно позаботиться о передаче в основную программу результата, потому что в классе `std::thread` такой механизм не предусмотрен. Тут-то и приходит на помощь шаблон функции `std::async` (также объявленный в заголовке `<future>`).

Функция `std::async` позволяет запустить *асинхронную задачу*, результат которой прямо сейчас не нужен. Но вместо объекта `std::thread` она возвращает объект `std::future`, который будет содержать возвращенное значение, когда оно станет доступно. Когда программе понадобится значение, она вызовет функцию-член `get()` объекта-будущего, и тогда поток будет приостановлен до готовности будущего результата, после чего вернет значение. В листинге ниже оказан простой пример.

Листинг 4.6. Использование `std::future` для получения результата асинхронной задачи

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();

int main() {
 std::future<int> the_answer =
  std::async(find_the_answer_to_ltuae);
 do_other_stuff();
 std::cout << "Ответ равен " << the_answer.get() << std::endl;
}
```

Шаблон `std::async` позволяет передать функции дополнительные параметры, точно так же, как `std::thread`. Если первым аргументом является указатель на функцию-член, то второй аргумент должен содержать объект, от имени которого эта функция-член вызывается (сам объект, указатель на него или обертывающий его `std::ref`), а все последующие аргументы передаются без изменения функции-члену. В противном случае второй и последующие аргументы передаются функции или допускающему вызов объекту, заданному в первом аргументе. Как и в `std::thread`, если аргументы представляют собой *r*-значения, то создаются их копии посредством *перемещения* оригинала. Это позволяет использовать в качестве объекта-функции и аргументов типы, допускающие только перемещение. Пример см. в листинге ниже.

Листинг 4.7. Передача аргументов функции, заданной в `std::async`

```
#include <string>
#include <future>

struct X {
 void foo(int, std::string const&);
```

```
 std::string bar(std::string const&);
};

X x;
auto f1 = std::async(&X::foo, &x, 42, "hello");
auto f2 = std::async(&X::bar, x, "goodbye");

struct Y {
 double operator()(double);
};

Y y;
auto f3 = std::async(Y(), 3.141);
auto f4 = std::async(std::ref(y), 2.718);

X baz(X&);
std::async(baz, std::ref(x));

class move_only {
public:
 move_only();
 move_only(move_only&&);
 move_only(move_only const&) = delete;
 move_only& operator=(move_only&&);
 move_only& operator=(move_only const&) = delete;
 void operator()();
};
auto f5 = std::async(move_only());
```

| Вызывается p->foo(42,"hello"), где p=&x

вызывается tmpx.bar("goodbye"), где tmpx — копия x

| Вызывается tmpy(3.141), где tmpy создается из Y перемещающим конструктором

Вызывается y(2.718)

Вызывается baz(x)

| Вызывается tmp(), где tmp конструируется с помощью std::move(move_only())

По умолчанию реализации предоставлено право решать, запускает ли `std::async` новый поток или задача работает синхронно, когда программа ожидает будущего результата. В большинстве случаев такое поведение вас устроит, но можно задать требуемый режим в дополнительном параметре `std::async` перед вызываемой функцией. Этот параметр имеет тип `std::launch` и может принимать следующие значения: `std::launch::deferred` — отложить вызов функции до того момента, когда будет вызвана функция-член `wait()` или `get()` объекта-будущего; `std::launch::async` — запускать функцию в отдельном потоке; `std::launch::deferred | std::launch::async` — оставить решение на усмотрение реализации. Последний вариант подразумевается по умолчанию. В случае отложенного вызова функция может вообще никогда не выполниться. Например:

```
auto f6 =
 std::async(std::launch::async, Y(), 1.2);
auto f7 =
 std::async(
  std::launch::deferred, baz, std::ref(x));
auto f8 = std::async(
 std::launch::deferred | std::launch::async,
 baz, std::ref(x));
auto f9 = std::async(baz, std::ref(x));
```

| Выполнять в новом потоке

| Выполнять при вызове wait() или get()

Оставить на

`f7.wait();`← **Вызвать отложенную функцию**

Ниже в этой главе и далее в главе 8 мы увидим, что с помощью `std::async` легко разбивать алгоритм на параллельно выполняемые задачи. Однако это не единственный способ ассоциировать объект `std::future` с задачей; можно также обернуть задачу объектом шаблонного класса `std::packaged_task<>` или написать код, который будет явно устанавливать значения с помощью шаблонного класса `std::promise<>`. Шаблон `std::packaged_task` является абстракцией более высокого уровня, чем `std::promise`, поэтому начнем с него.

4.2.2. Ассоциирование задачи с будущим результатом

Шаблон класса `std::packaged_task<>` связывает будущий результат с функцией или объектом, допускающим вызов. При вызове объекта `std::packaged_task<>` ассоциированная функция или допускающий вызов объект вызывается и делает будущий результат *готовым*, сохраняя возвращенное значение в виде ассоциированных данных. Этот механизм можно использовать для построение пулов потоков (см. главу 9) и иных схем управления, например, запускать каждую задачу в отдельном потоке или запускать их все последовательно в выделенном фоновом потоке. Если длительную операцию можно разбить на автономные подзадачи, то каждую из них можно обернуть объектом `std::packaged_task<>` и передать этот объект планировщику задач или пулу потоков. Таким образом, мы абстрагируем специфику задачи — планировщик имеет дело только с экземплярами `std::packaged_task<>`, а не с индивидуальными функциями.

Параметром шаблона класса `std::packaged_task<>` является сигнатура функции, например `void()` для функции, которая не принимает никаких параметров и не возвращает значения, или `int(std::string&, double*)` для функции, которая принимает неконстантную ссылку на `std::string` и указатель на `double` и возвращает значение типа `int`. При конструировании экземпляра `std::packaged_task` вы обязаны передать функцию или допускающий вызов объект, который принимает параметры указанных типов и возвращает значение типа, преобразуемого в указанный тип возвращаемого значения. Точного совпадения типов не требуется; можно сконструировать объект `std::packaged_task<double (double)>` из функции, которая принимает `int` и возвращает `float`, потому что между этими типами существуют неявные преобразования.

Тип возвращаемого значения, указанный в сигнатуре функции, определяет тип объекта `std::future<>`, возвращаемого функцией-членом `get_future()`, а заданный в сигнатуре список аргументов используется для определения сигнатуры оператора вызова в классе упакованной задачи. Например, в листинге ниже приведена часть определения класса `std::packaged_task<std::string(std::vector<char>*, int)>`.

Листинг 4.8. Определение частичной специализации `std::packaged_task`

```
template<>
class packaged_task<std::string(std::vector<char>*, int)> {
public:
 template<typename Callable>
 explicit packaged_task(Callable&& f);
```

```
 std::future<std::string> get_future();
 void operator()(std::vector<char>*, int);
};
```

Таким образом, `std::packaged_task` — допускающий вызов объект, и, значит, его можно обернуть объектом `std::function`, передать `std::thread` в качестве функции потока, передать любой другой функции, которая ожидает допускающий вызов объект, или даже вызвать напрямую. Если `std::packaged_task` вызывается как объект-функция, то аргументы, переданные оператору вызова, без изменения передаются обернутой им функции, а возвращенное значение сохраняется в виде асинхронного результата в объекте `std::future`, полученном от `get_future()`. Следовательно, мы можем обернуть задачу в `std::packaged_task` и извлечь будущий результат перед тем, как передавать объект `std::packaged_task` в то место, из которого он будет в свое время вызван. Когда результат понадобится, нужно будет подождать готовности будущего результата. В следующем примере показано, как всё это делается на практике.

Передача задач между потоками

Во многих каркасах для разработки пользовательского интерфейса требуется, чтобы интерфейс обновлялся только в специально выделенных потоках. Если какому-то другому потоку потребуется обновить интерфейс, то он должен послать сообщение одному из таких выделенных потоков, чтобы тот выполнил операцию. Шаблон `std::packaged_task` позволяет решить эту задачу, не заводя специальных сообщений для каждой относящейся к пользовательскому интерфейсу операции.

Листинг 4.9. Выполнение кода в потоке пользовательского интерфейса с применением `std::packaged_task`

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()>> tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();

void gui_thread() {                                  ← (1)
 while (!gui_shutdown_message_received()) {          ← (2)
  get_and_process_gui_message();                     ← (3)
  std::packaged_task<void()> task; {
   std::lock_guard<std::mutex> lk(m);
   if (tasks empty())                                ← (4)
    continue;
   task = std::move(tasks.front());                  ← (5)
   tasks.pop_front();
  }
```

```
    task();                                          ← (6)
    }
  }

  std::thread gui_bg_thread(gui_thread);

  template<typename Func>
  std::future<void> post_task_for_gui_thread(Func f) {
    std::packaged_task<void()> task(f);              ← (7)
    std::future<void> res = task.get_future();       ← (8)
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));                ← (9)
    return res;                                      ← (10)
  }
```

Код очень простой: поток пользовательского интерфейса **(1)** повторяет цикл, пока не будет получено сообщение о необходимости завершить работу **(2)**. На каждой итерации проверяется, есть ли готовые для обработки сообщения GUI **(3)**, например события мыши, или новые задачи в очереди. Если задач нет **(4)**, программа переходит на начало цикла; в противном случае извлекает задачу из очереди **(5)**, освобождает защищающий очередь мьютекс и исполняет задачу **(6)**. По завершении задачи будет готов ассоциированный с ней будущий результат.

Помещение задачи в очередь ничуть не сложнее: по предоставленной функции создается новая упакованная задача **(7)**, для получения ее будущего результата вызывается функция-член `get_future()` **(8)**, после чего задача помещается в очередь **(9)** еще до того, как станет доступен будущий результат **(10)**. Затем часть программы, которая отправляла сообщение потоку пользовательского интерфейса, может дождаться будущего результата, если хочет знать, как завершилась задача, или отбросить его, если это несущественно.

В этом примере мы использовали класс `std::packaged_task<void()>` для задач, обертывающих функцию или иной допускающий вызов объект, который не принимает параметров и возвращает `void` (если он вернет что-то другое, то возвращенное значение будет отброшено). Это простейшая из всех возможных задач, но, как мы видели ранее, шаблон `std::packaged_task` применим и в более сложных ситуациях — задав другую сигнатуру функции в качестве параметра шаблона, вы сможете изменить тип возвращаемого значения (и, стало быть, тип данных, которые хранятся в состоянии, ассоциированном с будущим объектом), а также типы аргументов оператора вызова. Наш пример легко обобщается на задачи, которые должны выполняться в потоке GUI и при этом принимают аргументы и возвращают в `std::future` какие-то данные, а не только индикатор успешности завершения.

А как быть с задачами, которые нельзя выразить в виде простого вызова функции, или такими, где результат может поступать из нескольких мест? Эти проблемы решаются с помощью еще одного способа создания будущего результата: явного задания значения с помощью шаблона класса `std::promise`[8].

4.2.3. Использование `std::promise`

При написании сетевых серверных программ часто возникает искушение обрабатывать каждый запрос на соединение в отдельном потоке, поскольку при такой структуре порядок коммуникации становится нагляднее и проще для программирования. Этот подход срабатывает, пока количество соединений (и, следовательно, потоков) не слишком велико. Но с ростом числа потоков увеличивается и объем потребляемых ресурсов операционной системы, а равно частота контекстных переключений (если число потоков превышает уровень аппаратного параллелизма), что негативно сказывается на производительности. В предельном случае у операционной системы могут закончиться ресурсы для запуска новых потоков, хотя пропускная способность сети еще не исчерпана. Поэтому в приложениях, обслуживающих очень большое число соединений, обычно создают совсем немного потоков (быть может, всего один), каждый из которых одновременно обрабатывает несколько запросов.

Рассмотрим один из таких потоков. Пакеты данных приходят по разным соединениям в случайном порядке, а потому и порядок помещения исходящих пакетов в очередь отправки тоже непредсказуем. Часто будет складываться ситуация, когда другие части приложения ждут либо успешной отправки данных, либо поступления нового пакета по конкретному сетевому соединению.

Шаблон `std::promise<T>` дает возможность задать значение (типа `T`), которое впоследствии можно будет прочитать с помощью ассоциированного объекта `std::future<T>`. Пара `std::promise`/`std::future` реализует один из возможных механизмов такого рода; ожидающий поток приостанавливается в ожидании будущего результата, тогда как поток, поставляющий данные, может с помощью `promise` установить ассоциированное значение и сделать будущий результат *готовым*.

Чтобы получить объект `std::future`, ассоциированный с данным обещанием `std::promise`, мы должны вызвать функцию-член `get_future()` — так же, как в случае `std::packaged_task`. После установки значения обещания (с помощью функции-члена `set_value()`) будущий результат становится *готовым*, и его можно использовать для получения установленного значения. Если уничтожить объект `std::promise`, не установив значение, то в будущем результате будет сохранено исключение. О передаче исключений между потоками см. раздел 4.2.4.

В листинге 4.10 приведен код потока обработки соединений, написанный по только что изложенной схеме. В данном случае для уведомления об успешной передаче блока исходящих данных применяется пара `std::promise<bool>`/`std::future<bool>`; ассоциированное с будущим результатом значение — это просто булевский флаг успех/неудача. Для входящих пакетов в качестве ассоциированных данных могла бы выступать полезная нагрузка пакета.

Листинг 4.10. Обработка нескольких соединений в одном потоке с помощью объектов-обещаний

```
#include <future>

void process_connections(connection_set& connections) {
 while(!done(connections)) {                    ← (1)
  for (connection_iterator                      ← (2)
   connection = connections.begin(), end = connections.end();
   connection != end;
   ++connection) {
```

```
    if (connection->has_incoming_data()) {← (3)
     data_packet data = connection->incoming();
     std::promise<payload_type>& p =
      connection->get_promise(data.id);   ← (4)
     p.set_value(data.payload);
    }
    if (connection->has_outgoing_data()) {← (5)
     outgoing_packet data =
      connection->top_of_outgoing_queue();
     connection->send(data.payload);
     data.promise.set_value(true);        ← (6)
    }
   }
  }
}
```

Функция `process_connections()` повторяет цикл, пока `done()` возвращает `true` **(1)**. На каждой итерации поочередно проверяется каждое соединение **(2)**; если есть входящие данные, они читаются **(3)**, а если в очереди имеются исходящие данные, они отсылаются **(5)**. При этом предполагается, что в каждом входящем пакете хранится некоторый идентификатор и полезная нагрузка, содержащая собственно данные. Идентификатору сопоставляется объект `std::promise` (возможно, путем поиска в ассоциативном контейнере) **(4)**, значением которого является полезная нагрузка пакета. Исходящие пакеты просто извлекаются из очереди отправки и передаются но соединению. После завершения передачи в обещание, ассоциированное с исходящими данными, записывается значение `true`, обозначающее успех **(6)**. Насколько хорошо эта схема ложится на фактический сетевой протокол, зависит от самого протокола; в конкретном случае схема обещание/будущий результат может и не подойти, хотя структурно она аналогична поддержке асинхронного ввода/вывода в некоторых операционных системах.

В коде выше мы полностью проигнорировали возможные исключения. Хотя мир, в котором всё всегда работает правильно, был бы прекрасен, действительность не так радужна. Переполняются диски, не находятся искомые данные, отказывает сеть, «падает» база данных — всякое бывает. Если бы операция выполнялась в том потоке, которому нужен результат, программа могла бы просто сообщить об ошибке с помощью исключения. Но было бы неоправданным ограничением требовать, чтобы всё работало правильно только потому, что мы захотели воспользоваться классами `std::packaged_task` или `std::promise`.

Поэтому в стандартной библиотеке C++ имеется корректный способ учесть возникновение исключений в таком контексте и сохранить их как часть ассоциированного результата.

4.2.4. Сохранение исключения в будущем результате

Рассмотрим следующий коротенький фрагмент. Если передать функции `square_root()` значение `-1`, то она возбудит исключение, которое увидит вызывающая программа:

```
double square_root(double x) {
 if (x<0) {
  throw std::out_of_range("x<0");
 }
 return sqrt(x);
```

```
}
```

А теперь предположим, что вместо вызова `square_root()` в текущем потоке

```
double y = square_root(-1);
```

мы вызываем ее асинхронно:

```
std::future<double> f = std::async(square_root,-1);
double y = f.get();
```

В идеале хотелось бы получить точно такое же поведение: чтобы поток, вызывающий `f.get()`, мог увидеть не только нормальное значение `y`, но и исключение — как в однопоточной программе.

Что ж, именно так на самом деле и происходит: если функция, вызванная через `std::async`, возбуждает исключение, то это исключение сохраняется в будущем результате вместо значения, а когда будущий результат оказывается *готовым*, вызов `get()` повторно возбуждает сохраненное исключение. (Примечание: стандарт ничего не говорит о том, возбуждается ли исходное исключение или его копия; различные компиляторы и библиотеки вправе решать этот вопрос по-разному.) То же самое происходит, когда функция обернута объектом `std::packaged_task`, — если при вызове задачи обернутая функция возбуждает исключение, то объект исключения сохраняется в будущем результате вместо значения, и это исключение повторно возбуждается при обращении к `get()`.

Разумеется, `std::promise` обеспечивает те же возможности в случае явного вызова функции. Чтобы сохранить исключение вместо значения, следует вызвать функцию-член `set_exception()`, а не `set_value()`. Обычно это делается в блоке `catch`:

```
extern std::promise<double> some_promise;

try {
 some_promise.set_value(calculate_value());
} catch (...) {
 some_promise.set_exception(std::current_exception());
}
```

Здесь мы воспользовались функцией `std::current_exception()`, которая возвращает последнее возбужденное исключение, но могли вызвать `std::copy_exception()`, чтобы поместить в объект-обещание новое исключение, которое никем не возбуждалось:

```
some_promise.set_exception(
 std::copy_exception(std::logic_error("foo"));
```

Если тип исключения заранее известен, то это решение гораздо чище, чем использование блока `try/catch`; мы не только упрощаем код, но и оставляем компилятору возможности для его оптимизации.

Есть еще один способ сохранить исключение в будущем результате: уничтожить ассоциированный с ним объект `std::promise` или `std::packaged_task`, не вызывая функцию установки значения в случае обещания или не обратившись к упакованной задаче. В любом случае деструктор `std::promise` или `std::packaged_task` сохранит в ассоциированном состоянии исключение типа `std::future_error`, в котором код ошибки равен `std::future_errc::broken_promise`, если только будущий результат еще не *готов*; создавая объект-будущее, вы даете обещание предоставить значение или исключение, а, уничтожая объект, не задав ни того, ни другого, вы это обещание нарушаете. Если бы компилятор в этом случае не сохранил ничего в будущем результате, то ожидающие потоки могли бы никогда не выйти из состояния ожидания.

До сих пор мы во всех примерах использовали `std::future`. Однако у этого шаблонного класса есть ограничения, и не в последнюю очередь тот факт, что результата может ожидать

только один поток. Если требуется, чтобы одного события ждали несколько потоков, то придётся воспользоваться классом `std::shared_future`.

4.2.5. Ожидание в нескольких потоках

Хотя класс `std::future` сам заботится о синхронизации, необходимой для передачи данных из одного потока в другой, обращения к функциям-членам одного и того же экземпляра `std::future` не синхронизированы между собой. Работа с одним объектом `std::future` из нескольких потоков без дополнительной синхронизации может закончиться *гонкой за данными* и неопределенным поведением. Так и задумано: `std::future` моделирует единоличное владение результатом асинхронного вычисления, и одноразовая природа `get()` в любом случае делает параллельный доступ бессмысленным — извлечь значение может только один поток, поскольку после первого обращения к `get()` никакого значения не остается.

Но если дизайн вашей фантастической параллельной программы требует, чтобы одного события могли ждать несколько потоков, то не отчаивайтесь: на этот случай предусмотрен шаблон класса `std::shared_future`. Если `std::future` допускает только *перемещение*, чтобы владение можно было передавать от одного экземпляра другому, но в каждый момент времени на асинхронный результат ссылался лишь один экземпляр, то экземпляры `std::shared_future` допускают и *копирование*, то есть на одно и то же ассоциированное состояние могут ссылать несколько объектов.

Но и функции-члены объекта `std::shared_future` не синхронизированы, поэтому во избежание гонки за данными при доступе к одному объекту из нескольких потоков вы сами должны обеспечить защиту. Но более предпочтительный способ — скопировать объект, так чтобы каждый поток работал со своей копией. Доступ к разделяемому асинхронному состоянию из нескольких потоков безопасен, если каждый поток обращается к этому состоянию через свой собственный объект `std::shared_future`. См. Рис. 4.1.

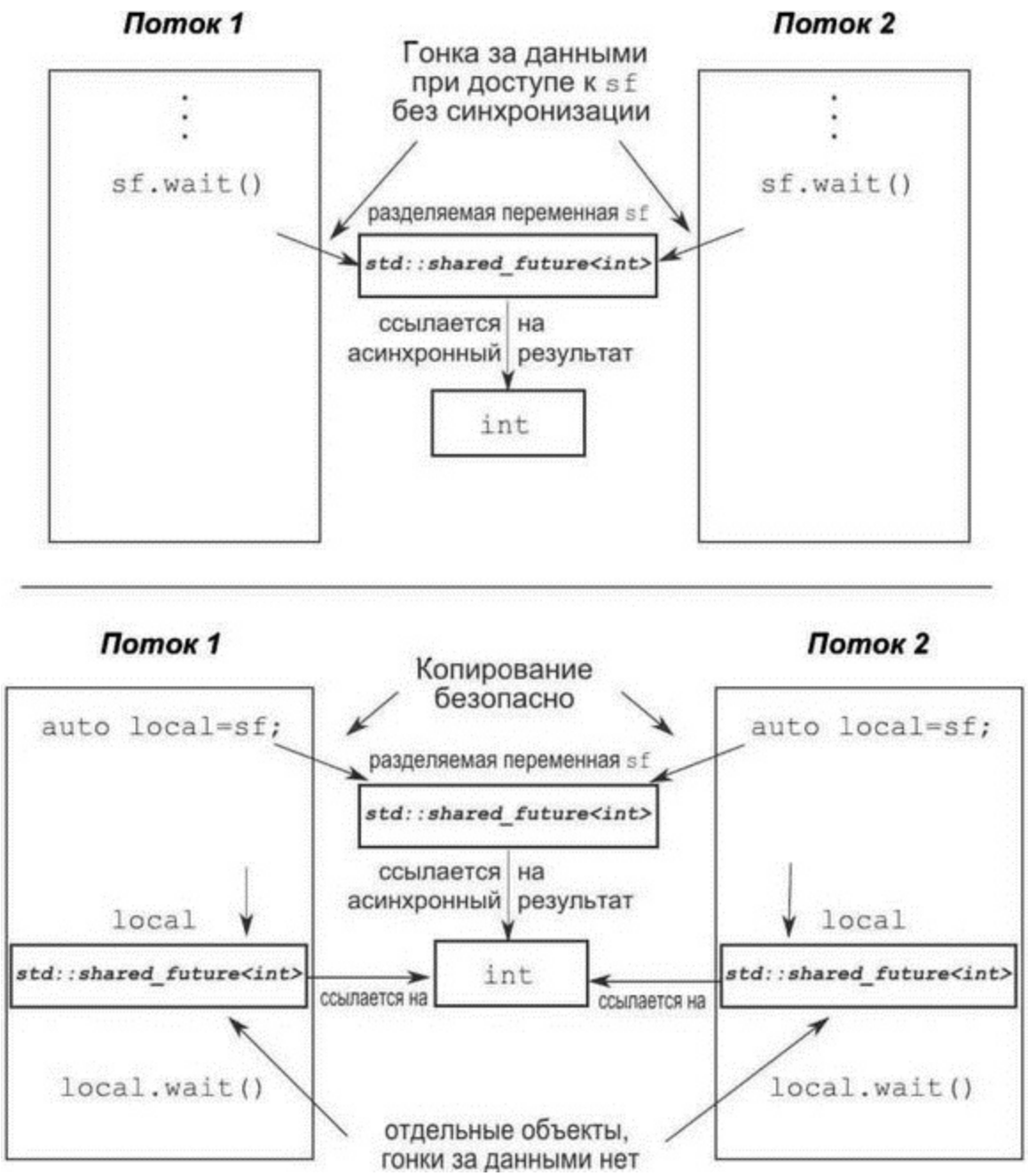

Рис. 4.1. Использование нескольких объектов `std::shared_future`, чтобы избежать гонки за данными

Одно из потенциальных применений `std::shared_future` — реализация параллельных вычислений наподобие применяемых в сложных электронных таблицах: у каждой ячейки имеется единственное окончательное значение, на которое могут ссылаться формулы, хранящиеся в нескольких других ячейках. Формулы для вычисления значений в зависимых ячейках могут использовать `std::shared_future` для ссылки на первую ячейку. Если формулы во всех ячейках вычисляются параллельно, то задачи, которые могут дойти до конца, дойдут, а те, что зависят от результатов вычислений других ячеек, окажутся заблокированы до разрешения зависимостей. Таким образом, система сможет но максимуму задействовать доступный аппаратный параллелизм.

Экземпляры `std::shared_future`, ссылающиеся на некоторое асинхронное состояние, конструируются из экземпляров `std::future`, ссылающихся на то же состояние. Поскольку объект `std::future` не разделяет владение асинхронным состоянием ни с каким другим объектом, то передавать владение объекту `std::shared_future` необходимо с помощью `std::move`, что оставляет `std::future` с пустым состоянием, как если бы он был

сконструирован по умолчанию:

```
std::promise<int> p;
std::future<int> f(p.get_future());← (1) Будущий результат f
assert(f.valid());                   действителен

std::shared_future<int> sf(std::move(f));
assert(!f.valid());← (2) f больше не действителен
assert(sf.valid());← (3) sf теперь действителен
```

Здесь будущий результат `f` в начальный момент действителен `(1)`, потому что ссылается на асинхронное состояние обещания `p`, но после передачи состояния объекту `sf` результат `f` оказывается недействительным **(2)**, а `sf` — действительным **(3)**.

Как и для других перемещаемых объектов, передача владения для r-значения производится неявно, поэтому объект `std::shared_future` можно сконструировать прямо из значения, возвращаемого функцией-членом `get_future()` объекта `std::promise`, например:

```
std::promise<std::string> p;← (1) Неявная передача владения
std::shared_future<std::string> sf(p.get_future());
```

Здесь передача владения неявная; объект `std::shared_future<>` конструируется из r-значения типа `std::future<std::string>` **(1)**.

У шаблона `std::future` есть еще одна особенность, которая упрощает использование `std::shared_future` совместно с новым механизмом автоматического выведения типа переменной из ее инициализатора (см. приложение А, раздел А.6). В шаблоне `std::future` имеется функция-член `share()`, которая создает новый объект `std::shared_future` и сразу передаёт ему владение. Это позволяет сделать код короче и проще для изменения:

```
std::promise<
 std::map<SomeIndexType, SomeDataType, SomeComparator,
          SomeAllocator>::iterator> p;
auto sf = p.get_future().share();
```

В данном случае для `sf` выводится тип `std::shared_future<std::map<SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, такое название произнести-то трудно. Если компаратор или распределитель изменятся, то вам нужно будет поменять лишь тип обещания, а тип будущего результата изменится автоматически.

Иногда требуется ограничить время ожидания события — либо потому что на время исполнения некоторого участка кода наложены жесткие ограничения, либо потому что поток может заняться другой полезной работой, если событие долго не возникает. Для этого во многих функциях ожидания имеются перегруженные варианты, позволяющие задать величину таймаута.

4.3. Ожидание с ограничением по времени

Все блокирующие вызовы, рассмотренные до сих пор, приостанавливали выполнение потока на неопределенно долгое время — до тех пор, пока не произойдёт ожидаемое событие. Часто это вполне приемлемого в некоторых случаях время ожидания желательно ограничить. Например, это может быть полезно, когда нужно отправить сообщение вида «Я еще жив» интерактивному пользователю или другому процессу или когда ожидание действительно необходимо прервать, потому что пользователь устал ждать и нажал **Cancel**.

Можно задать таймаут одного из двух видов: *интервальный*, когда требуется ждать в течение определённого промежутка времени (к примеру, 30 миллисекунд) или *абсолютный*, когда требуется ждать до наступления указанного момента (например, 17:30:15.045987023 UTC 30 ноября 2011 года). У большинства функций ожидания имеются оба варианта. Варианты, принимающие интервальный таймаут, оканчиваются словом `_for`, а принимающие абсолютный таймаут — словом `_until`.

Например, в классе `std::condition_variable` есть по два перегруженных варианта функций-членов `wait_for()` и `wait_until()`, соответствующие двум вариантам `wait()` — первый ждет поступления сигнала или истечения таймаута или ложного пробуждения, второй проверяет при пробуждении переданный предикат и возвращает управление, только если предикат равен `true` (и условной переменной поступил сигнал) или истек таймаут.

Прежде чем переходить к детальному обсуждению функций с таймаутами, рассмотрим, как в C++ задается время, и начнем с часов.

4.3.1. Часы

С точки зрения стандартной библиотеки C++, часы — это источник сведений о времени. Точнее, класс часов должен предоставлять четыре элемента информации:

- текущее время *now*;
- тип значения для представления времени, полученного от часов;
- величина такта часов;
- признак равномерного хода времени, такие часы называются *стабильными*.

Получить от часов текущее время можно с помощью статической функции-члена `now()`; например, функция `std::chrono::system_clock::now()` возвращает текущее время по системным часам. Тип точки во времени для конкретного класса часов определяется с помощью члена `typedef time_point`, поэтому значение, возвращаемое функцией `some_clock::now()` имеет тип `some_clock::time_point`.

Тактовый период часов задается в виде числа долей секунды, которое определяется членом класса `typedef period`; например, если часы тикают 25 раз в секунду, то член `period` будет определён как `std::ratio<1, 25>`, тогда как в часах, тикающих один раз в 2,5 секунды, член `period` определён как `std::ratio<5, 2>`. Если тактовый период не известен до начала выполнения программы или может изменяться во время работы, то `period` можно определить как средний период, наименьший период или любое другое значение, которое сочтет нужным автор библиотеки. Нет гарантии, что тактовый период, наблюдаемый в любом конкретном прогоне программы, соответствует периоду, определённому с помощью члена period.

Если часы *ходят с постоянной частотой* (вне зависимости от того, совпадает эта частота с `period` или нет) и *не допускают подведения*, то говорят, что часы *стабильны*. Статический член `is_steady` класса часов равен `true`, если часы стабильны, и `false` в противном случае. Как правило, часы `std::chrono::system_clock` нестабильны, потому что их можно подвести, даже если такое подведение производится автоматически, чтобы учесть локальный дрейф. Из-за подведения более позднее обращение к `now()` может вернуть значение, меньшее, чем более раннее, а это нарушение требования к равномерному ходу часов. Как мы скоро увидим, стабильность важна для вычислений с таймаутами, поэтому в стандартной библиотеке C++ имеется класс стабильных часов — `std::chrono::steady_clock`. Помимо него, стандартная библиотека содержит класс `std::chrono::system_clock` (уже упоминавшийся выше), который представляет системный генератор «реального времени» и имеет функции для преобразования моментов времени в тип `time_t` и обратно, и класс `std::chrono::high_resolution_clock`, который представляет наименьший возможный тактовый период (и, следовательно, максимально возможное разрешение). Может статься, что этот тип на самом деле является псевдонимом `typedef` какого-то другого класса часов. Все эти классы определены в заголовке `<chrono>` наряду с прочими средствами работы со временем.

Чуть ниже мы рассмотрим представления моментов времени, но сначала познакомимся с представлением интервалов.

4.3.2. Временные интервалы

Интервалы — самая простая часть подсистемы поддержки времени; они представлены шаблонным классом `std::chrono::duration<>` (все имеющиеся в C++ средства работы со временем, которые используются в библиотеке Thread Library, находятся в пространстве имен `std::chrono`). Первый параметр шаблона — это тип представления (`int`, `long` или `double`), второй — дробь, показывающая, сколько секунд представляет один интервал. Например, число минут, хранящееся в значении типа `short`, равно `std::chrono::duration<short, std::ratio<60,1>>`, потому что в одной минуте 60 секунд. С другой стороны, число миллисекунд, хранящееся в значении типа `double`, равно `std::chrono::duration<double, std::ratio<1, 1000>>`, потому что миллисекунда — это 1/1000 секунды.

В пространстве имен `std::chrono` имеется набор предопределенных `typedef`'ов для различных интервалов: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes` и `hours`. В них используется достаточно широкий целочисленный тип, подобранный так, чтобы можно было представить в выбранных единицах интервал продолжительностью свыше 500 лет. Имеются также `typedef` для всех определенных в системе СИ степеней 10 — от `std::atto` (10^{-18}) до `std::exa` (10^{18}) (и более, если платформа поддерживает 128-разрядные целые числа) — чтобы можно было определить нестандартные интервалы, например `std::duration<double, std::centi>` (число сотых долей секунды, хранящееся в значении типа `double`).

Между типами интервалов существует неявное преобразование, если не требуется отсечение (то есть неявно преобразовать часы в секунды можно, а секунды в часы нельзя). Для явного преобразования предназначен шаблон функции `std::chrono::duration_cast<>`:

```
std::chrono::milliseconds ms(54802);
```

```
std::chrono::seconds s =
 std::chrono::duration_cast<std::chrono::seconds>(ms);
```

Результат отсекается, а не округляется, поэтому в данном примере `s` будет равно 54.

Для интервалов определены арифметические операции, то есть сложение и вычитание интервалов, а также умножение и деление на константу базового для представления типа (первый параметр шаблона) дает новый интервал. Таким образом, `5*seconds(1)` — то же самое, что `seconds(5)` или `minutes(1) - seconds(55)`. Количество единиц в интервале возвращает функция-член `count()`. Так, `std::chrono::milliseconds(1234).count()` равно 1234.

Чтобы задать ожидание в течение интервала времени, используется функция `std::chrono::duration<>`. Вот, например, как задается ожидание готовности будущего результата в течение 35 миллисекунд:

```
std::future<int> f = std::async(some_task);
if (f.wait_for(std::chrono::milliseconds(35)) ==
    std::future_status::ready)
 do_something_with(f.get());
```

Все функции ожидания возвращают код, показывающий, истек ли таймаут или произошло ожидаемое событие. В примере выше мы ожидаем будущий результат, поэтому функция вернет `std::future_status::timeout`, если истек таймаут, `std::future_status::ready` — если результат готов, и `std::future_status::deferred` — если будущая задача отложена. Время ожидания измеряется с помощью библиотечного класса стабильных часов, поэтому 35 мс — это всегда 35 мс, даже если системные часы были подведены (вперёд или назад) в процессе ожидания. Разумеется, из-за особенностей системного планировщика и варьирующейся точности часов ОС фактическое время между вызовом функции в потоке и возвратом из нее может оказаться значительно больше 35 мс.

Разобравшись с интервалами, мы можем перейти к моментам времени.

4.3.3. Моменты времени

Момент времени представляется конкретизацией шаблона класса `std::chrono::time_point<>`, в первом параметре которой задаются используемые часы, а во втором — единица измерения (специализация шаблона `std::chrono::duration<>`). Значением момента времени является промежуток времени (измеряемый в указанных единицах) с некоторой конкретной точки на временной оси, которая называется *эпохой* часов. Эпоха часов — это основополагающее свойство, однако напрямую его запросить нельзя, и в стандарте C++ оно не определено. Из типичных эпох можно назвать полночь (00:00) 1 января 1970 года и момент, когда в последний раз был загружен компьютер, на котором исполняется приложение. У разных часов может быть общая или независимые эпохи. Если у двух часов общая эпоха, то псевдоним типа `typedef time_point` в одном классе может ссылаться на другой класс как на тип, ассоциированный с `time_point`. Хотя узнать, чему равна эпоха, невозможно, вы *можете* получить время между данным моментом `time_point` и эпохой с помощью функции-члена `time_since_epoch()`, которая возвращает интервал.

Например, можно задать момент времени `std::chrono::time_point <std::chrono::system_clock, std::chrono::minutes>`. Он представляет время по системным часам, выраженное в минутах, а не в естественных для этих часов единицах (как

правило, секунды или доли секунды).

К объекту `std::chrono::time_point<>` можно прибавить интервал или вычесть из него интервал — в результате получится новый момент времени. Например, `std::chrono::high_resolution_clock::now() + std::chrono::nanoseconds(500)` соответствует моменту времени в будущем, который отстоит от текущего момента на 500 наносекунд. Это удобно для вычисления абсолютного таймаута, когда известна максимально допустимая продолжительность выполнения некоторого участка программы, и внутри этого участка есть несколько обращений к функциям с ожиданием или обращения к функциям, которые ничего не ждут, но предшествуют функции с ожиданием и занимают часть отведенного времени.

Можно также вычесть один момент времени из другого при условии, что они относятся к одним и тем же часам. В результате получиться интервал между двумя моментами. Это полезно для хронометража участков программы, например:

```
auto start = std::chrono::high_resolution_clock::now();
do_something();
auto stop = std::chrono::high_resolution_clock::now();
std::cout << "do_something() заняла "
 << std::chrono::duration<
     double, std::chrono::seconds>(stop-start).count()
 << " секунд" << std::endl;
```

Однако параметр `clock` объекта `std::chrono::time_point<>` не только определяет эпоху. Если передать момент времени функции с ожиданием, принимающей абсолютный таймаут, то указанный в нем параметр `clock` используется для измерения времени. Это существенно в случае, когда часы подводятся, потому что механизм ожидания замечает, что наказания часов изменились, и не дает функции вернуть управление, пока функция-член часов `now()` не вернет значение, большее, чем задано в таймауте. Если часы подведены вперёд, то это может уменьшить общее время ожидания (измеренное но стабильным часам), а если назад — то увеличить.

Как и следовало ожидать, моменты времени применяются в вариантах функций с ожиданием, имена которых заканчиваются словом `_until`. Как правило, таймаут задается в виде смещения от значения *`some-clock`*`::now()`, вычисленного в определенной точке программы, хотя моменты времени, ассоциированные с системными часами, можно получить из `time_t` с помощью статической функции-члена `std::chrono::system_clock::to_time_point()`, если при планировании операций требуется использовать время в понятном пользователю масштабе. Например, если на ожидание события, связанного с условной переменной, отведено не более 500 мс, то можно написать такой код.

Листинг 4.11. Ожидание условной переменной с таймаутом

```
#include <condition_variable>
#include <mutex>
#include <chrono>

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop() {
```

```
    auto const timeout = std::chrono::steady_clock::now() +
                         std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done) {
     if (cv.wait_until(lk, timeout) == std::cv_status::timeout)
      break;
    }
    return done;
}
```

Это рекомендуемый способ ожидания условной переменной с ограничением по времени в случае, когда предикат не указывается. При этом ограничивается общее время выполнения цикла. В разделе 4.1.1 мы видели, что при использовании условных переменных без предиката цикл необходим для защиты от ложных пробуждений. Но если вызывать в цикле `wait_for()`, то может получиться, что функция прождёт почти все отведенное время, а затем произойдёт ложное пробуждение, после чего на следующей итерации отсчет времени начнется заново. И так может происходить сколько угодно раз, в результате чего общее время ожидания окажется неограниченным.

Вооружившись знаниями о том, как задавать таймауты, рассмотрим функции, в которых таймауты используются.

4.3.4. Функции, принимающие таймаут

Простейший случай использования таймаута — задание паузы в потоке, чтобы он не отнимал у других потоков время, когда ему нечего делать. Соответствующий пример был приведён в разделе 4.1, где мы в цикле опрашивали флаг «done». Для этого использовались функции `std::this_thread::sleep_for()` и `std::this_thread::sleep_until()`. Обе работают как будильник: поток засыпает либо на указанный интервал (в случае `sleep_for()`), либо до указанного момента времени (в случае `sleep_until()`). Функцию `sleep_for()` имеет смысл применять в ситуации, описанной в разделе 4.1, когда что-то необходимо делать периодически и важна лишь продолжительность периода. С другой стороны, функция `sleep_until()` позволяет запланировать пробуждение потока в конкретный момент времени, например: запустить в полночь резервное копирование, начать в 6 утра распечатку платёжной ведомости или приостановить поток до момента следующего обновления кадра при воспроизведении видео.

Разумеется, таймаут принимают не только функции типа `sleep`. Выше мы видели, что таймаут можно задавать при ожидании условных переменных и будущих результатов. А также при попытке захватить мьютекс, если сам мьютекс такую возможность поддерживает. Обычные классы `std::mutex` и `std::recursive_mutex` не поддерживают таймаут при захвате, зато его поддерживают классы `std::timed_mutex` и `std::recursive_timed_mutex`. В том и в другом имеются функции-члены `try_lock_for()` и `try_lock_until()`, которые пытаются получить блокировку в течение указанного интервала или до наступления указанного момента времени. В табл. 4.1 перечислены функции из стандартной библиотеки C++, которые принимают таймауты, их параметры и возвращаемые значения. Параметр ***`duration`*** должен быть объектом типа `std::duration<>`, а параметр ***`time_point`*** — объектом типа `std::time_point<>`.

Таблица 4.1. Функции, принимающие таймаут

Класс / пространство имен	Функции	Возвращаемые
`std::this_thread` пространство имен	`sleep_for(`***duration***`)` `sleep_until(`***time_point***`)`	Неприменимо
`std::condition_variable` или `std::condition_variable_any`	`wait_for(`***lock, duration***`)` `wait_until(`***lock, time_point***`)`	`std::cv_status::t` `std::cv_status::n`
	`wait_for(`***lock, duration, predicate***`)` `wait_until(`***lock, time_point, predicate***`)`	`bool` — значение, во предикатом ***predica*** пробуждении
`std::timed_mutex` или `std::recursive_timed_mutex`	`try_lock_for(`***duration***`)` `try_lock_until(`***time_point***`)`	`bool` — `true`, есл захвачен, инач
`std::unique_lock<`**`TimedLockable`**`>`	`unique_lock(`***lockable, duration***`)` `unique_lock(`***lockable, time_point***`)`	Неприменимо — фу `owns_lock()` для вно сконструированного возвращает `true`, ес захвачен, иначе `fals`
	`try_lock_for(`***duration***`)` `try_lock_until(`***time_point***`)`	`bool` — `true`, если м захвачен, иначе `fals`
`std::future<ValueType>` или `std::shared_future<ValueType>`	`wait_for(`***duration***`)` `wait_until(`***time_point***`)`	`std::future_statu` если истек таймаут, `std::future_statu` если будущий резул `std::future_statu` если в будущем резу хранится отложенна которая еще не нача исполняться

Теперь, когда мы рассмотрели условные переменные, будущие результаты, обещания и упакованные задачи, настало время представить более широкую картину их применения для синхронизации операций, выполняемых в разных потоках.

4.4. Применение синхронизации операций для упрощения кода

Использование описанных выше средств синхронизации в качестве строительных блоков позволяет сосредоточиться на самих нуждающихся в синхронизации операциях, а не на механизмах реализации. В частности, код можно упростить, применяя более *функциональный* (в смысле *функционального программирования*) подход к программированию параллелизма. Вместо того чтобы напрямую разделять данные между потоками, мы можем снабдить каждый поток необходимыми ему данными, а результаты вычисления предоставить другим потокам, которые в них заинтересованы, с помощью будущих результатов.

4.4.1. Функциональное программирование с применением будущих результатов

Термином *функциональное программирование* (ФП) называют такой стиль программирования, при котором результат функции зависит только от ее параметров, но не от внешнего состояния. Это напрямую соотносится с понятием функции в математике и означает, что если два раза вызвать функцию с одними и теми же параметрами, то получатся одинаковые результаты. Таким свойством обладают многие математические функции в стандартной библиотеке C++, например `sin`, `cos` и `sqrt`, а также простые операции над примитивными типами, например `3+3`, `6*9` или `1.3/4.7`. *Чистая* функция не *модифицирует* никакое внешнее состояние, она воздействует только на возвращаемое значение.

При таком подходе становится проще рассуждать о функциях, особенно в присутствии параллелизма, поскольку многие связанные с разделяемой памятью проблемы, обсуждавшиеся в главе 3, просто не возникают. Если разделяемые данные не модифицируются, то не может быть никакой гонки и, стало быть, не нужно защищать данные с помощью мьютексов. Это упрощение настолько существенно, что в программировании параллельных систем все более популярны становятся такие языки, как Haskell[9], где все функции чистые *по умолчанию*. В таком окружении *нечистые* функции, которые все же модифицируют разделяемые данные, отчетливо выделяются, поэтому становится проще рассуждать о том, как они укладываются в общую структуру приложения.

Но достоинства функционального программирования проявляются не только в языках, где эта парадигма применяется по умолчанию. C++ — мультипарадигменный язык, и на нем, безусловно, можно писать программы в стиле ФП. С появлением в C++11 лямбда-функций (см. приложение A, раздел A.6), включением шаблона `std::bind` из Boost и TR1 и добавлением автоматического выведения типа переменных (см. приложение A, раздел A.7) это стало даже проще, чем в C++98. Будущие результаты — это последний элемент из тех, что позволяют реализовать на C++ параллелизм в стиле ФП; благодаря передаче будущих результатов результат одного вычисления можно сделать зависящим от результата другого *без явного доступа к разделяемым данным*.

Быстрая сортировка в духе ФП

Чтобы продемонстрировать использование будущих результатов при написании параллельных программ в духе ФП, рассмотрим простую реализацию алгоритма быстрой сортировки Quicksort. Основная идея алгоритма проста: имея список значений, выбрать некий опорный элемент и разбить список на две части — в одну войдут элементы, меньшие опорного, в другую — большие или равные опорному. Отсортированный список получается путем сортировки обоих частей и объединения трех списков: отсортированного множества элементов, меньших опорного элемента, самого опорного элемента и отсортированного множества элементов, больших или равных опорному элементу. На рис. 4.2 показано, как этот алгоритм сортирует список из 10 целых чисел. В листинге ниже приведена последовательная реализация алгоритма в духе ФП; в ней список принимается и возвращается по значению, а не сортируется по месту в `std::sort()`.

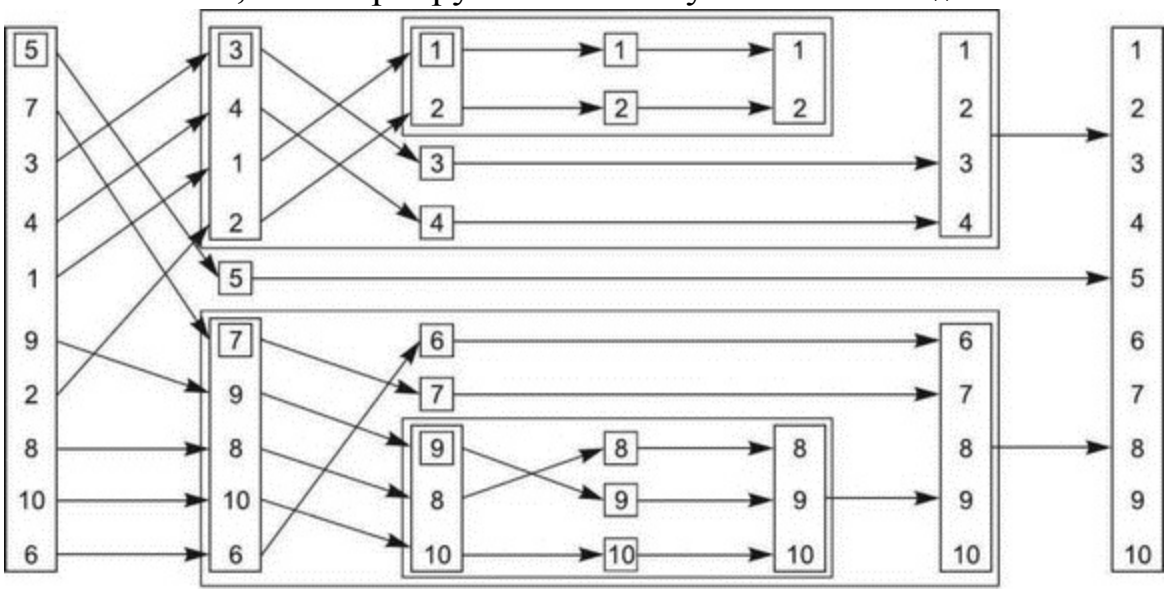

Рис. 4.2. Рекурсивная сортировка в духе ФП

Листинг 4.12. Последовательная реализация Quicksort в духе ФП

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input) {
 if (input.empty()) {
  return input;
 }
 std::list<T> result;
 result.splice(result.begin(), input, input.begin());← (1)

 T const& pivot = *result.begin();                    ← (2)

 auto divide_point = std::partition(input.begin(), input.end(),
  [&](T const& t) { return t < pivot; });← (3)

 std::list<T> lower_part;
 lower_part.splice(
  lower_part.end(), input, input.begin(), divide_point); ← (4)

 auto new_lower(
  sequential_quick_sort(std::move(lower_part)));       ← (5)
 auto new_higher(
```

```
  sequential_quick_sort(std::move(input)));          ← (6)

  result.splice(result.end(), new_higher);  ← (7)
  result.splice(result.begin(), new_lower); ← (8)

  return result;
}
```

Хотя интерфейс выдержан в духе ФП, прямое применение ФП привело бы к неоправданно большому числу операций копирования, поэтому внутри мы используем «обычный» императивный стиль. В качестве опорного мы выбираем первый элемент и отрезаем его от списка с помощью функции `splice()` **(1)**. Потенциально это может привести к неоптимальной сортировке (в терминах количества операций сравнения и обмена), но любой другой подход при работе с `std::list` может существенно увеличить время за счет обхода списка. Мы знаем, что этот элемент должен войти в результат, поэтому можем сразу поместить его в список, где результат будет храниться. Далее мы хотим использовать этот элемент для сравнения, поэтому берем ссылку на него, чтобы избежать копирования **(2)**. Теперь можно с помощью алгоритма `std::partition` разбить последовательность на две части: *меньшие* опорного элемента и *не меньшие* опорного элемента **(3)**. Критерий разбиения проще всего задать с помощью лямбда-функции; мы запоминаем ссылку в замыкании, чтобы не копировать значение `pivot` (подробнее о лямбда-функциях см. в разделе А.5 приложения А).

Алгоритм `std::partition()` переупорядочивает список на месте и возвращает итератор, указывающий на первый элемент, который *не* меньше опорного значения. Полный тип итератора довольно длинный, поэтому мы используем спецификатор типа `auto`, чтобы компилятор вывел его самостоятельно (см. приложение А, раздел А.7).

Раз уж мы выбрали интерфейс в духе ФП, то для рекурсивной сортировки обеих «половин» нужно создать два списка. Для этого мы снова используем функцию `splice()`, чтобы переместить значения из списка `input` до `divide_point` включительно в новый список `lower_part` **(4)**. После этого `input` будет со держать только оставшиеся значения. Далее оба списка можно отсортировать путем рекурсивных вызовов **(5)**, **(6)**. Применяя `std::move()` для передачи списков, мы избегаем копирования — результат в любом случае неявно перемещается. Наконец, мы еще раз вызываем `splice()`, чтобы собрать result в правильном порядке. Значения из списка `new_higher` попадают в конец списка **(7)**, после опорного элемента, а значения из списка `new_lower` — в начало списка, до опорного элемента **(8)**.

Параллельная реализация Quicksort в духе ФП

Раз уж мы все равно применили функциональный стиль программирования, можно без труда распараллелить этот код с помощью будущих результатов, как показано в листинге ниже. Набор операций тот же, что и раньше, только некоторые из них выполняются параллельно.

Листинг 4.13. Параллельная реализация Quicksort с применением будущих результатов

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input) {
 if (input.empty()) {
  return input;
 }

 std::list<T> result;
 result.splice(result.begin(), input, input.begin());
 T const& pivot = *result.begin();

 auto divide_point = std::partition(input.begin(), input.end(),
  [&](T const& t) {return t<pivot;});
 std::list<T> lower_part;
 lower_part.splice(
  lower_part.end(), input, input.begin(), divide_point);

 std::future<std::list<T> > new_lower( ← (1)
  std::async(&parallel_quick_sort<T>, std::move(lower_part)));

 auto new_higher(
  parallel_quick_sort(std::move(input))); ← (2)

 result.splice(result.end(), new_higher); ← (3)

 result.splice(result.begin(), new_lower.get()); ← (4)
 return result;
}
```

Существенное изменение здесь заключается в том, что сортировка нижней части списка производится не в текущем, а в отдельном потоке — с помощью `std::async()` **(1)**. Верхняя часть списка сортируется путем прямой рекурсии, как и раньше **(2)**. Рекурсивно вызывая `parallel_quick_sort()`, мы можем задействовать доступный аппаратный параллелизм. Если `std::async()` создает новый поток при каждом обращении, то после трех уровней рекурсии мы получим восемь работающих потоков, а после 10 уровней (когда в списке примерно 1000 элементов) будет работать 1024 потока, если оборудование позволяет. Если библиотека решит, что запущено слишком много задач (быть может, потому что количество задач превысило уровень аппаратного параллелизма), то может перейти в режим синхронного запуска новых задач. Тогда новая задача будет работать в том же потоке, который обратился к `get()`, а не в новом, так что мы не будем нести издержки на передачу задачи новому потоку, если это не увеличивает производительность. Стоит отметить, что в соответствии со стандартом реализация `std::async` вправе как создавать новый поток для каждой задачи (даже при значительном превышении лимита), если явно не задан флаг `std::launch::deferred`, так и запускать все задачи синхронно, если явно не задан флаг `std::launch::async`. Рассчитывая, что библиотека сама позаботится об автоматическом масштабировании, изучите, что говорится на эту тему в документации, поставляемой вместе с библиотекой.

Можно не использовать `std::async()`, а написать свою функцию `spawn_task()`, которая будет служить оберткой вокруг `std::packaged_task` и `std::thread`, как показано в листинге 4.14; нужно создать объект `std::packaged_task` для хранения результата вызова функции,

получить из него будущий результат, запустить задачу в отдельном потоке и вернуть будущий результат. Само по себе это не дает большого преимущества (и, скорее всего, приведёт к значительному превышению лимита), но пролагает дорогу к переходу на более хитроумную реализацию, которая помещает задачу в очередь, обслуживаемую пулом потоков. Рассмотрение пулов потоков мы отложим до главы 9. Но идти по такому пути вместо использования `std::async` имеет смысл только в том случае, когда вы точно знаете, что делаете, и хотите полностью контролировать, как пул потоков строится и выполняет задачи.

Но вернемся к функции `parallel_quick_sort`. Поскольку для получения `new_higher` мы применяли прямую рекурсию, то и срастить (splice) его можно на месте, как и раньше **(3)**. Но `new_lower` теперь представляет собой не список, а объект `std::future<std::list<T>>`, поэтому сначала нужно извлечь значение с помощью `get()`, а только потом вызывать `splice()` **(4)**. Таким образом, мы дождемся завершения фоновой задачи, а затем *переместим* результат в параметр `splice()`; функция `get()` возвращает ссылку на r-значение — хранимый результат, следовательно, его можно переместить (подробнее о ссылках на r-значения и семантике перемещения см. в разделе A.1.1 приложения A).

Даже в предположении, что `std::async()` оптимально использует доступный аппаратный параллелизм, приведённая реализация Quicksort все равно не идеальна. Основная проблема в том, что `std::partition` делает много работы и остается последовательной операцией, но пока остановимся на этом. Если вас интересует максимально быстрая параллельная реализация, обратитесь к научной литературе.

Листинг 4.14. Простая реализация функции `spawn_task`

```
template<typename F, typename A>
std::future<std::result_of<F(A&&)>::type>
spawn_task(F&& f, A&& a) {
 typedef std::result_of<F(A&&)>::type result_type;
 std::packaged_task<result_type(A&&)>
 task(std::move(f)));
 std::future<result_type> res(task.get_future());
 std::thread t(std::move(task), std::move(a));
 t.detach();
 return res;
}
```

Функциональное программирование — не единственная парадигма параллельного программирования, позволяющая избежать модификации разделяемых данных. Альтернативой является парадигма CSP (Communicating Sequential Processes — взаимодействующие последовательные процессы)[10], в которой потоки концептуально рассматриваются как полностью независимые сущности, без каких бы то ни было разделяемых данных, но соединенные коммуникационными каналами, по которым передаются сообщения. Эта парадигма положена в основу языка программирования Erlang (http://www.erlang.org/) и среды MPI (Message Passing Interface) (http://www.mpi-forum.org/), широко используемой для высокопроизводительных вычислений на C и C++. Уверен, что теперь вы не удивитесь, узнав, что и эту парадигму можно поддержать на C++, если соблюдать определенную дисциплину; в следующем разделе показано, как это можно сделать.

4.4.2. Синхронизация операций с помощью передачи сообщений

Идея CSP проста: если никаких разделяемых данных нет, то каждый поток можно рассматривать независимо от остальных, учитывая лишь его поведение в ответ на получаемые сообщения. Таким образом, поток по существу является конечным автоматом: получив сообщение, он как-то изменяет свое состояние, возможно, посылает одно или несколько сообщений другим потокам и выполняет то или иное вычисление, зависящее от начального состояния. Один из способов такого способа программирования потоков — формализовать это описание и реализовать модель конечного автомата, но этот путь не единственный — конечный автомат может неявно присутствовать в самой структуре приложения. Какой метод будет работать лучше в конкретном случае, зависит от требований к поведению приложения и от опыта разработчиков. Но каким бы образом ни был реализован поток, у разбиения на независимые процессы есть несомненное преимущество — потенциальное устранение многих сложностей, связанных с параллельным доступом к разделяемым данным, и, следовательно, упрощение программирования и снижение количества ошибок.

У настоящих последовательных взаимодействующих процессов вообще нет разделяемых данных, а весь обмен информацией производится через очереди сообщений. Но, поскольку в C++ потоки имеют общее адресное пространство, то обеспечить строгое соблюдение этого требования невозможно. Тут-то и приходит на выручку дисциплина: следить за тем, чтобы никакие данные не разделялись между потоками, — обязанность автора приложения или библиотеки. Разумеется, сами очереди сообщений должны разделяться, иначе потоки не смогут взаимодействовать, но детали этого механизма можно вынести в библиотеку. Представьте, что вам нужно написать программу для банкомата. Она должна поддерживать взаимодействие с человеком, который хочет снять деньги, с соответствующим банком, а также управлять оборудованием, которое принимает платёжную карту, выводит на экран сообщения, обрабатывает нажатия клавиш, выдает деньги и возвращает карту.

Чтобы воплотить все это в жизнь, можно было бы разбить код на три независимых потока: один будет управлять оборудованием, второй — реализовывать логику работы банкомата, а третий — обмениваться информацией с банком. Эти потоки могут взаимодействовать между собой посредством передачи сообщений, а не за счет разделения данных. Например, поток, управляющий оборудованием, будет посылать сообщение потоку логики банкомата о том, что человек вставил карту или нажал кнопку. Поток логики будет посылать потоку, управляющему оборудованием, сообщение о том, сколько денег выдать. И так далее.

Смоделировать логику банкомата можно, например, с помощью конечного автомата. В каждом состоянии поток ждет сообщение, которое затем обрабатывает. Это может привести к переходу в новое состояние, после чего цикл продолжится. На рис. 4.3 показаны состояния, присутствующие в простой реализации программы. Здесь система ждет, пока будет вставлена карта. Когда это произойдёт, система ждет, что пользователь введет свой ПИН-код, по одной цифре за раз. Последнюю введенную цифру пользователь может удалить. После того как будет введено нужное количество цифр, система проверяет ПИН-код. Если он введен неправильно, больше делать нечего — клиенту нужно вернуть карту и ждать, пока будет вставлена следующая карта. Если ПИН-код правильный, то система ждет либо отмены транзакции, либо выбора снимаемой суммы. Если пользователь отменил операцию, ему

нужно вернуть карту и закончить работу. Если он выбрал сумму, то система ждет подтверждения от банка, а затем либо выдает наличные и возвращает карту, либо выводит сообщение «недостаточно средств на счете» и тоже возвращает карту. Понятно, что реальный банкомат гораздо сложнее, но и этого достаточно для иллюстрации идеи.

Рис. 4.3. Модель простого конечного автомата для банкомата

Спроектировав конечный автомат для реализации логики банкомата, мы можем оформить его в виде класса, в котором каждому состоянию соответствует функция-член. Каждая такая функция ждет поступления одного из допустимых сообщений, обрабатывает его и, возможно, инициирует переход в новое состояние. типы сообщений представлены структурами `struct`. В листинге 4.15 приведена часть простой реализации логики банкомата в такой системе — главный цикл и код первого состояния, в котором программа ожидает вставки карты.

Как видите, вся синхронизация, необходимая для передачи сообщений, целиком скрыта в библиотеке (ее простая реализация приведена в приложении С вместе с полным кодом этого примера).

Листинг 4.15. Простая реализация класса, описывающего логику работы банкомата

```
struct card_inserted {
 std::string account;
};

class atm {
 messaging::receiver incoming;
 messaging::sender bank;
 messaging::sender interface_hardware;
 void (atm::*state)();

 std::string account;
 std::string pin;
```

```
 void waiting_for_card() {                                    ←(1)
  interface_hardware.send(display_enter_card());              ←(2)
  incoming.wait()                                             ←(3)
   .handle<card_inserted>(
    [&](card_inserted const& msg) {                           ←(4)
    account = msg.account;
    pin = "";
    interface_hardware.send(display_enter_pin());
    state = &atm::getting_pin;
   }
  );
 }

 void getting_pin();

public:
 void run() {                                                 ←(5)
  state = &atm::waiting_for_card;                             ←(6)
  try {
   for(;;) {
    (this->*state)();                                         ←(7)
   }
  }
  catch(messaging::close_queue const&) {}
 }
};
```

Мы уже говорили, что эта реализация неизмеримо проще логики работы реального банкомата, но она все же дает представление о программировании на основе передачи сообщений. Не нужно думать о проблемах параллельности и синхронизации, наша основная забота — понять, какие входные сообщения допустимы в данной точке и какие сообщения посылать в ответ. Конечный автомат, реализующий логику банкомата, работает в одном потоке, а прочие части системы, например интерфейс с банком и с терминалом, — в других потоках. Такой принцип проектирования программ называется *моделью акторов* — в системе есть несколько акторов (каждый работает в своем потоке), которые посылают друг другу сообщения с просьбой выполнить определённое задание, и никакого разделяемого состояния, помимо передаваемого в составе сообщений, не существует.

Выполнение начинается в функции-члене `run()` **(5)**, которая устанавливает начальное состояние `waiting_for_card` **(6)**, а затем в цикле вызывает функции-члены, представляющие текущее состояние (каким бы оно ни было) **(7)**. Функции состояния — это просто функции-члены класса `atm`. Функция `waiting_for_card` **(1)** тоже не представляет сложности: она посылает сообщение интерфейсу с просьбой вывести сообщение «Вставьте карту» **(2)**, а затем ожидает сообщения, которое могла бы обработать **(3)**. Единственное допустимое в этой точке сообщение — `card_inserted`; оно обрабатывается лямбда-функцией **(4)**. Функции `handle` можно передать любую функцию или объект-функцию, но в таком простом случае лямбда-функции вполне достаточно. Отметим, что вызов функции `handle()` сцеплен с вызовом `wait()`; если получено сообщение недопустимого типа, оно отбрасывается, и поток ждет, пока не придёт подходящее сообщение.

Сама лямбда-функция просто запоминает номер карточного счета в переменной-члене,

очищает текущий ПИН-код и переходит в состояние «получение ПИН». По завершении обработчика сообщений функция состояния возвращает управление главному циклу, который вызывает функцию следующего состояния **(7)**.

Функция состояния `getting_pin` несколько сложнее, потому что может обрабатывать сообщения разных типов, как следует из рис. 4.3. Ниже приведён ее код.

Листинг 4.16. Функция состояния `getting_pin` для простой реализации банкомата

```
void atm::getting_pin() {
 incoming.wait()
 .handle<digit_pressed>(     ← (1)
  [&](digit_pressed const& msg) {
   unsigned const pin_length = 4;
   pin += msg.digit;
   if (pin.length() == pin_length) {
    bank.send(verify_pin(account, pin, incoming));
    state = &atm::verifying_pin;
   }
  }
 )
 .handle<clear_last_pressed>(← (2)
  [&](clear_last_pressed const& msg) {
   if (!pin.empty()) {
    pin.resize(pin.length() - 1);
   }
  }
 )
 .handle<cancel_pressed>(    ← (3)
  [&](cancel_pressed const& msg) {
   state = &atm::done_processing;
  }
 );
}
```

Поскольку теперь допустимы сообщения трех типов, то с функцией `wait()` сцеплены три вызова функции `handle()` **(1)**, **(2)**, **(3)**. В каждом вызове `handle()` в качестве параметра шаблона указан тип сообщения, а в качестве параметра самой функции — лямбда-функция, которая принимает сообщение этого типа. Поскольку вызовы сцеплены, функция `wait()` знает, что может ожидать сообщений `digit_pressed`, `clear_last_pressed` или `cancel_pressed`. Сообщения всех прочих типов игнорируются.

Как видим, теперь состояние изменяется не всегда. Например, при получении сообщения `digit_pressed` мы просто дописываем цифру в конец `pin`, если эта цифра не последняя. Затем главный цикл (**(7)** в листинге 4.15) снова вызовет функцию `getting_pin()`, чтобы ждать следующую цифру (или команду очистки либо отмены).

Это соответствует поведению, изображенному на рис. 4.3. Каждое состояние реализовано отдельной функцией-членом, которая ждет сообщений определенных типов и при необходимости обновляет состояние.

Как видите, такой стиль программирования может заметно упростить проектирование параллельной системы, поскольку все потоки рассматриваются как абсолютно независимые. Таким образом, мы имеем пример использования нескольких потоков для разделения обязанностей, а, значит, от нас требуется явно решить, как распределять между ними задачи.

4.5. Резюме

Синхронизация операций между потоками — важная часть написания параллельного приложения. Если синхронизации нет, то потоки ведут себя независимо, и их вполне можно было бы реализовать как отдельные приложения, запускаемые группой, потому что выполняют взаимосвязанные действия. В этой главе мы рассмотрели различные способы синхронизации операций — простые условные переменные, будущие результаты, обещания и упакованные задачи. Мы также обсудили несколько подходов к решению проблем синхронизации: функциональное программирование, когда каждая задача порождает результат, зависящий только от входных данных, но не от внешнего состояния, и передачу сообщений, когда взаимодействие потоков осуществляется за счет асинхронного обмена сообщениями с помощью какой-либо подсистемы передачи сообщений, играющей роль посредника.

Теперь, когда мы обсудили многие высокоуровневые средства, имеющиеся в C++, настало время познакомиться с низкоуровневыми механизмами, которые приводят всю систему в движение: модель памяти C++ и атомарные операции.

Глава 5.
Модель памяти C++ и атомарные операции

В этой главе:

- Детальные сведения о модели памяти C++.
- Атомарные типы в стандартной библиотеке C++.
- Операции над атомарными типами.
- Как можно использовать эти операции для синхронизации потоков.

Одна из самых важных особенностей стандарта C++11 — та, которую большинство программистов даже не замечают. Это не новые синтаксические конструкции и не новые библиотечные средства, а новая модель памяти, учитывающая многопоточность. Без модели памяти, которая точно определяет, как должны работать основополагающие строительные блоки, ни на одно из описанных выше средств нельзя было бы полагаться. Понятно, почему большинство программистов этого не замечают: если вы пользуетесь для защиты данных мьютексами, а для сигнализации о событиях — условными переменными или будущими результатам, то вопрос о том, *почему* они работают, не так уж важен. И лишь когда вы подбираетесь «ближе к железу», становятся существенны точные детали модели памяти.

C++ используется для решения разных задач, но одна из основных — системное программирование. Поэтому комитет по стандартизации в числе прочих целей ставил и такую: сделать так, чтобы в языке более низкого уровня, чем C++, не возникало необходимости. C++ должен обладать достаточной гибкостью, чтобы программист мог сделать то, что хочет, без помех со стороны языка, в том числе и работать «на уровне железа». Атомарные типы и операции — шаг именно в этом направлении, поскольку они предоставляют низкоуровневые механизмы синхронизации, которые обычно транслируются в одну-две машинные команды.

В этой главе мы начнем с рассмотрения основ модели памяти, затем перейдем к атомарным типам и операциям и в конце обсудим различные виды синхронизации, реализуемые с помощью операций над атомарными типами. Это довольно сложная тема; если вы не планируете писать код, в котором атомарные операции используются для синхронизации (например, структуры данных без блокировок, рассматриваемые в главе 7), то все эти детали вам ни к чему. Но давайте потихоньку двинемся вперёд и начнем с модели памяти.

5.1. Основы модели памяти

У модели памяти есть две стороны: базовые *структурные* аспекты, относящиеся к размещению программы в памяти, и аспекты, связанные с *параллелизмом*. Структурные аспекты важны для параллелизма, особенно если опуститься на низкий уровень атомарных операций, поэтому с них я и начну. В C++ всё вращается вокруг объектов и ячеек памяти.

5.1.1. Объекты и ячейки памяти

Любые данные в программе на C++ состоят из объектов. Это не значит, что можно создать новый класс, производный от `int`, или что у фундаментальных типов есть функции-члены, или вообще нечто такое, что часто имеют в виду, когда говорят «нет ничего, кроме объектов» при обсуждении таких языков, как Smalltalk или Ruby. Это утверждение просто означает, что в C++ данные строятся из объектов. В стандарте C++ объект определяется как «область памяти», хотя далее речь идет о таких свойствах объектов, как тип и время жизни.

Некоторые объекты являются простыми значениями таких фундаментальных типов, как `int` или `float`, другие — экземплярами определенных пользователем классов. У некоторых объектов (например, массивов, экземпляров производных классов и экземпляров классов с нестатическими данными-членами) есть подобъекты, у других — нет.

Вне зависимости от типа объект хранится в одной или нескольких *ячейках памяти*. Каждая такая ячейка — это либо объект (или подобъект) скалярного типа, например `unsigned short` или `my_class*`, либо последовательность соседних битовых полей. Если вы пользуетесь битовыми полями, то имейте в виду один важный момент: хотя соседние битовые поля является различными объектами, они тем не менее считаются одной ячейкой памяти. На рис. 5.1 показано, как структура `struct` представлена в виде совокупности объектов и ячеек памяти.

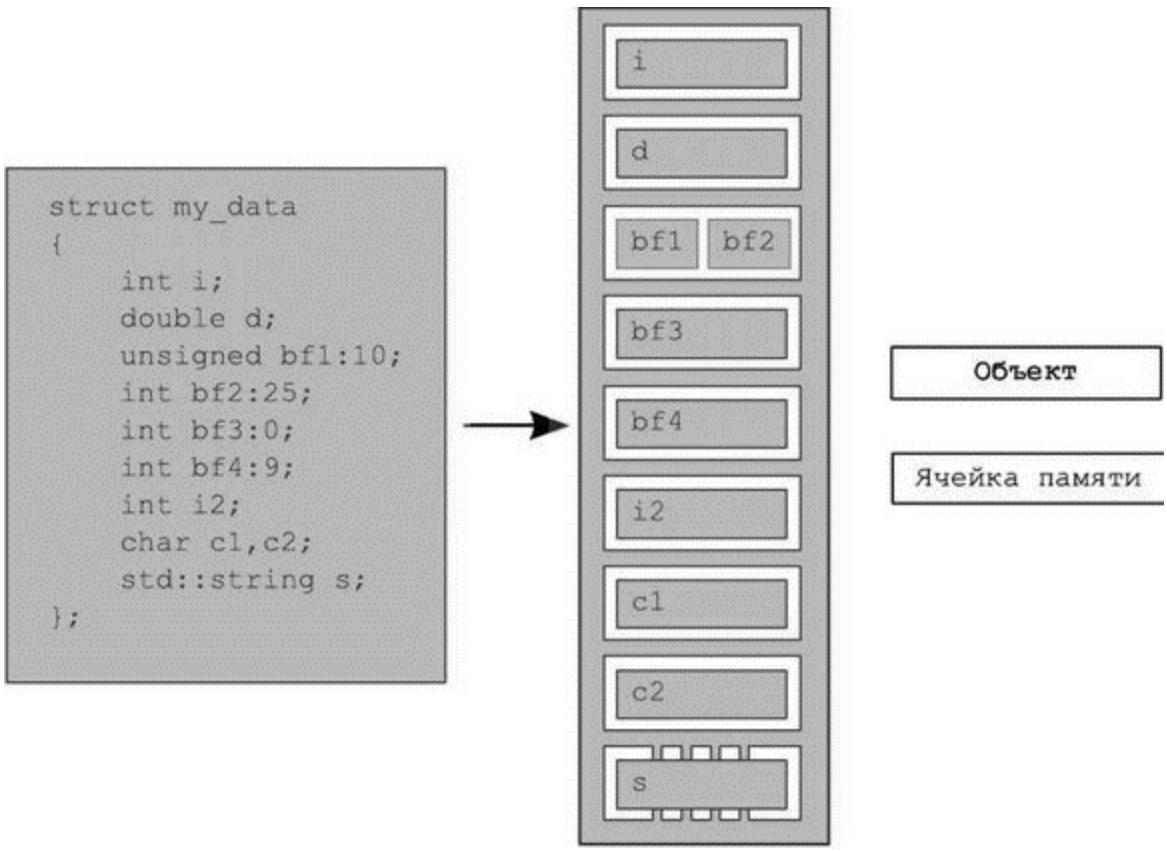

Рис. 5.1. Разбиение `struct` на объекты и ячейки памяти

Во-первых, вся структура — это один объект, который состоит из нескольких подобъектом, по одному для каждого члена данных. Битовые поля `bf1` и `bf2` занимают одну ячейку памяти, объект `s` типа `std::string` занимает несколько ячеек памяти, а для каждого из остальных членов отведена своя ячейка. Обратите внимание, что битовое поле нулевой длины `bf3` заставляет отвести для `bf4` отдельную ячейку.

Отсюда можно сделать несколько важных выводов:

• каждая переменная — объект, в том числе и переменные, являющиеся членами других объектов;

• каждый объект занимает *по меньшей мере* одну ячейку памяти;

• переменные фундаментальных типов, например `int` или `char`, занимают *в точности* одну ячейку памяти вне зависимости от размера, даже если являются соседними или элементами массива;

• соседние битовые поля размещаются в одной ячейке памяти.

Уверен, что вы недоумеваете, какое отношение всё это имеет к параллелизму. Давайте разберемся.

5.1.2. Объекты, ячейки памяти и параллелизм

Для многопоточных приложений на C++ понятие ячейки памяти критически важно. Если два потока обращаются к *разным* ячейкам памяти, то никаких проблем не возникает и всё работает, как надо. Но если потоки обращаются к *одной и той же* ячейке, то необходима осторожность. Если ни один поток не обновляет ячейку памяти, то всё хорошо — доступ к данным для чтения не нуждается ни в защите, ни в синхронизации. Если же какой-то поток модифицирует данные, то возможно состояние гонки, описанное в главе 3.

Чтобы избежать гонки, необходимо принудительно упорядочить обращения из двух потоков. Один из возможных способов такого упорядочения дают мьютексы (см. главу 3) — если захватывать один и тот же мьютекс перед каждым обращением, то одновременно получить доступ к ячейке памяти сможет только один поток, так что упорядочение налицо. Другой способ упорядочить доступ из двух потоков — воспользоваться свойствами синхронизации, присущими атомарным операциям (о том, что это такое, см. раздел 5.2) над теми же или другими ячейками памяти. Такое использование атомарных операций описано в разделе 5.3. Если к одной и той же ячейке обращаются более двух потоков, то упорядочение должно быть определено для каждой пары.

Если два обращения к одной и той же ячейке памяти из разных потоков не упорядочены и одно или оба обращения не являются атомарными и одно или оба обращения являются операциями записи, то имеет место гонка за данными, что приводит к неопределенному поведению.

Эта фраза критически важна: неопределенное поведение — один из самых грязных закоулков C++. Согласно стандарту языка, любое неопределенное поведение отменяет всякие гарантии — поведение всего приложения становится неопределённым, и оно может делать все, что угодно. Я знаю один пример неопределённого поведения, в результате которого загорелся монитор. Хотя маловероятно, что такое приключится с вами, гонка за данными безусловно является серьезной ошибкой, которой следует всеми силами избегать.

В этой фразе есть и еще один важный момент: избежать неопределенного поведения

поможет использование атомарных операций для доступа к ячейке памяти, за которую возможна гонка. Саму гонку это не предотвращает — какая именно атомарная операция первой получит доступ к ячейке памяти, все равно не определено, — но программа тем не менее возвращается в область определённого поведения.

Прежде чем мы перейдем к атомарным операциям, нужно разобраться еще в одной важной концепции, касающейся объектов и ячеек памяти: порядке модификации.

5.1.3. Порядок модификации

Для каждого объекта в программе на C++ определён *порядок модификации,* состоящий из всех операций записи в объект из всех потоков программы, начиная с инициализации объекта. В большинстве случаев порядок меняется от запуска к запуску, но при любом выполнении программы все имеющиеся в системе потоки должны договориться о порядке модификации. Если объект не принадлежит одному из описанных в разделе 5.2 атомарных типов, то вы сами отвечаете за обеспечение синхронизации, достаточной для того, чтобы потоки могли договориться о порядке модификации каждой переменной. Если разные потоки видят разные последовательности значений одной и той же переменной, то имеет место гонка за данными и, как следствие, неопределённое поведение (см. раздел 5.1.2). Если вы используете атомарные операции, то за обеспечение необходимой синхронизации отвечает компилятор.

Это требование означает, что некоторые виды спекулятивного исполнения[11] не разрешены, потому что после того как некоторый поток увидел определённое значение объекта при данном порядке модификации, последующие операции чтения в том же потоке должны возвращать более поздние значения, а последующие операции записи в тот же объект в этом потоке должны происходить позже при данном порядке модификации. Кроме того, операция чтения объекта, следующая за операцией записи в этот объект, должна вернуть либо записанное значение, либо другое значение, которое было записано позже при данном порядке модификации этого объекта. Хотя все потоки обязаны договориться о порядке модификации каждого объекта в программе, не требуется, чтобы они договаривались об относительном порядке операций над разными объектами. Дополнительные сведения об упорядочении операций, выполняемых в разных потоках, см. в разделе 5.3.3.

Итак, что понимается под атомарной операцией и как ими можно воспользоваться для принудительного упорядочения?

5.2. Атомарные операции и типы в C++

Под *атомарными* понимаются неделимые операции. Ни из одного потока в системе невозможно увидеть, что такая операция выполнена наполовину, — она либо выполнена целиком, либо не выполнена вовсе. Если операция загрузки, которая читает значение объекта, *атомарна*, и все операции модификации этого объекта также *атомарны*, то в результате загрузки будет получено либо начальное значение объекта, либо значение, сохраненное в нем после одной из модификаций.

И наоборот, если операция не атомарная, то другой поток может видеть, что она выполнена частично. Если это операция сохранения, то значение, наблюдаемое другим потоком, может не совпадать ни со значением до начала сохранения, ни с сохраненным значением. С другой стороны, операция загрузки может извлечь часть объекта, после чего значение будет модифицировано другим потоком, а затем операция прочитает оставшуюся часть объекта. В результате будет извлечено значение, которое объект не имел ни до, ни после модификации. Это простая проблематичная гонка, описанная в главе 3, но на этом уровне она может представлять собой *гонку за данными* (см. раздел 5.1) и, стало быть, являться причиной неопределённого поведения.

В C++ для того чтобы операция была атомарной, обычно необходимы атомарные типы. Давайте познакомимся с ними.

5.2.1. Стандартные атомарные типы

Все стандартные *атомарные типы* определены в заголовке `<atomic>`. Любые операции над такими типами атомарны, и только операции над этими типами атомарны в смысле принятого в языке определения, хотя мьютексы позволяют реализовать *кажущуюся* атомарность других операций. На самом деле, и сами стандартные атомарные типы могут пользоваться такой эмуляцией: почти во всех имеется функция-член `is_lock_free()`, которая позволяет пользователю узнать, выполняются ли операции над данным типом с помощью действительно атомарных команд (`x.is_lock_free()` возвращает `true`) или с применением некоторой внутренней для компилятора и библиотеки блокировки (`x.is_lock_free()` возвращает `false`).

Единственный тип, в котором функция-член `is_lock_free()` отсутствует, — это `std::atomic_flag`. В действительности это по-настоящему простой булевский флаг, а операции над этим типом *обязаны* быть свободными от блокировок; если имеется простой свободный от блокировок булевский флаг, то на его основе можно реализовать простую блокировку и, значит, все остальные атомарные типы. Говоря *по-настоящему простой*, я именно это и имел в виду: после инициализации объект типа `std::atomic_flag` сброшен, и для него определены всего две операции: проверить и установить (функция-член `test_and_set()`) и очистить (функция-член `clear()`). Это всё — нет ни присваивания, ни копирующего конструктора, ни операции «проверить и очистить», вообще ничего больше.

Доступ ко всем остальным атомарным типам производится с помощью специализаций шаблона класса `std::atomic<>`; их функциональность несколько богаче, но они необязательно свободны от блокировок (как было объяснено выше). На самых распространенных платформах можно ожидать, что атомарные варианты всех встроенных

типов (например, std::atomic<int> и std::atomic<void*>) действительно будут свободны от блокировок, но такого требования не предъявляется. Как мы скоро увидим, интерфейс каждой специализации отражает свойства типа; например, поразрядные операции, например &=, не определены для простых указателей, поэтому они не определены и для атомарных указателей.

Помимо прямого использования шаблона класса std::atomic<>, разрешается использовать имена, приведённые в табл. 5.1, которые ссылаются на определенные в конкретной реализации атомарные типы. Из-за исторических особенностей добавления атомарных типов в стандарт C++ альтернативные имена типов могут ссылаться либо на соответствующую специализацию std::atomic<>, либо на базовый класс этой специализации. Поэтому смешение альтернативных имен и прямых имен специализаций std::atomic<> может сделать программу непереносимой.

Таблица 5.1. Альтернативные имена стандартных атомарных типов и соответствующие им специализации std::atomic<>

Атомарный тип	Соответствующая специализация
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uhar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Помимо основных атомарных типов, в стандартной библиотеке C++ определены также псевдонимы typedef для атомарных типов, соответствующих различным неатомарным библиотечным typedef, например std::size_t. Они перечислены в табл. 5.2.

Таблица 5.2. Соответствие между стандартными атомарными и встроенными typedef

Атомарный typedef	Соответствующий typedef из стандартной библиотеки
atomic_int_least8_t	int_least8_t
atomic_uint_least8_t	uint_least8_t
atomic_int_least16_t	int_least16_t
atomic_uint_least16_t	uint_least16_t
atomic_int_least32_t	int_least32_t
atomic_uint_least32_t	uint_least32_t
atomic_int_least64_t	int_least64_t
atomic_uint_least64_t	uint_least64_t

```
atomic_int_fast8_t    int_fast8_t
atomic_uint_fast8_t   uint_fast8_t
atomic_int_fast16_t   int_fast16_t
atomic_uint_fast16_t  uint_fast16_t
atomic_int_fast32_t   int_fast32_t
atomic_uint_fast32_t  uint_fast32_t
atomic_int_fast64_t   int_fast64_t
atomic_uint_fast64_t  uint_fast64_t
atomic_intptr_t       intptr_t
atomic_uintptr_t      uintptr_t
atomic_size_t         size_t
atomic_ptrdiff_t      ptrdiff_t
atomic_intmax_t       intmax_t
atomic_uintmax_t      uintmax_t
```

Да уж, типов немало! Но есть простая закономерность — атомарный тип, соответствующий стандартному `typedef T`, имеет такое же имя с префиксом `atomic_`: `atomic_T`. То же самое относится и к встроенным типам с тем исключением, что `signed` сокращается до `s`, `unsigned` — до `u`, а `long long` — до `llong`. Вообще говоря, проще написать `std::atomic<T>` для нужного вам типа `T`, чем пользоваться альтернативными именами.

Стандартные атомарные типы не допускают копирования и присваивания в обычном смысле, то есть не имеют копирующих конструкторов и операторов присваивания. Однако им все же можно присваивать значения соответствующих встроенных типов, и они поддерживают неявные преобразования в соответствующие встроенные типы. Кроме того, в них определены функции-члены `load()`, `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`. Поддерживаются также составные операторы присваивания (там, где это имеет смысл) `+=`, `-=`, `*=`, `|=` и т.д., а для целочисленных типов и специализаций `std::atomic<>` для указателей — еще и операторы `++` и `--`. Этим операторам соответствуют также именованные функции-члены с идентичной функциональностью: `fetch_add()`, `fetch_or()` и т.д. Операторы присваивания возвращают сохраненное значение, а именованные функции-члены — значение, которое объект имел до начала операции. Это позволяет избежать потенциальных проблем, связанных с тем, что обычно операторы присваивания возвращают ссылку на объект в левой части. Чтобы получить из такой ссылки сохраненное значение, программа должна была бы выполнить еще одну операцию чтения, но тогда между присваиванием и чтением другой поток мог бы модифицировать значение, открывая дорогу гонке.

Но шаблон класса `std::atomic<>` — не просто набор специализаций. В нем есть основной шаблон, который можно использовать для создания атомарного варианта пользовательского типа. Поскольку это обобщенный шаблон класса, определены только операции `load()`, `store()` (а также присваивание значения пользовательского типа и преобразования в пользовательский тип), `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`.

У любой операции над атомарными типами имеется необязательный аргумент, задающий требования к семантике упорядочения доступа к памяти. Точный смысл различных вариантов упорядочения обсуждается в разделе 5.3. Пока же достаточно знать, что операции разбиты на три категории.

- Операции *сохранения*, для которых можно задавать упорядочение

memory_order_relaxed, memory_order_release и memory_order_seq_cst.

• Операции *загрузки*, для которых можно задавать упорядочение memory_order_relaxed, memory_order_consume, memory_order_acquire и memory_order_seq_cst.

• Операции *чтения-модификации-записи*, для которых можно задавать упорядочение memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel и memory_order_seq_cst.

По умолчанию для всех операций подразумевается упорядочение memory_order_seq_cst.

Теперь рассмотрим, какие операции можно производить над каждым из стандартных атомарных типов, начиная с std::atomic_flag.

5.2.2. Операции над std::atomic_flag

Простейший стандартный атомарный тип std::atomic_flag представляет булевский флаг. Объекты этого типа могут находиться в одном из двух состояний: установлен или сброшен. Этот тип намеренно сделан максимально простым, рассчитанным только на применение в качестве строительного блока. Поэтому увидеть его в реальной программе можно лишь в очень специфических обстоятельствах. Тем не менее, он послужит нам отправной точкой для обсуждения других атомарных типов, потому что на его примере отчетливо видны общие относящиеся к ним стратегии.

Объект типа std::atomic_flag *должен* быть инициализирован значением ATOMIC_FLAG_INIT. При этом флаг оказывается в состоянии *сброшен*. Никакого выбора тут не предоставляется — флаг всегда должен начинать существование в сброшенном состоянии:

```
std::atomic_flag f = ATOMIC_FLAG_INIT;
```

Требование применяется вне зависимости от того, где и в какой области видимости объект объявляется. Это единственный атомарный тип, к инициализации которого предъявляется столь специфическое требование, зато при этом он является также единственным типом, гарантированно свободным от блокировок. Если у объекта std::atomic_flag статический класс памяти, то он гарантированно инициализируется статически, и, значит, никаких проблем с порядком инициализации не будет — объект всегда оказывается инициализированным к моменту первой операции над флагом.

После инициализации с флагом можно проделать только три вещи: уничтожить, очистить или установить, одновременно получив предыдущее значение. Им соответствуют деструктор, функция-член clear() и функция-член test_and_set(). Для обеих функций clear() и test_and_set() можно задать упорядочение памяти. clear() — операция *сохранения*, поэтому варианты упорядочения memory_order_acquire и memory_order_acq_rel к ней неприменимы, а test_and_set() — операция чтения-модификации-записи, так что к ней применимы любые варианты упорядочения. Как и для любой атомарной операции, по умолчанию подразумевается упорядочение memory_order_seq_cst. Например:

```
f.clear(std::memory_order_release);← (1)
bool x = f.test_and_set();          ← (2)
```

Здесь при вызове clear() **(1)** явно запрашивается сброс флага с семантикой освобождения, а при вызове test_and_set() **(2)** подразумевается стандартное упорядочение

для операции установки флага и получения прежнего значения.

Объект `std::atomic_flag` нельзя сконструировать копированием из другого объекта, не разрешается также присваивать один `std::atomic_flag` другому. Это не особенность типа `std::atomic_flag`, а свойство, общее для всех атомарных типов. Любые операции над атомарным типом должны быть атомарными, а для присваивания и конструирования копированием нужны два объекта. Никакая операция над двумя разными объектами не может быть атомарной. В случае копирования и присваивания необходимо сначала прочитать значение первого объекта, а потом записать его во второй. Это две отдельные операции над двумя различными объектами, и их комбинация не может быть атомарной. Поэтому такие операции запрещены.

Такая ограниченность функциональности делает тип `std::atomic_flag` идеальным средством для реализации мьютексов-спинлоков. Первоначально флаг сброшен и мьютекс свободен. Чтобы захватить мьютекс, нужно в цикле вызывать функцию `test_and_set()`, пока она не вернет прежнее значение `false`, означающее, что теперь в *этом* потоке установлено значение флага `true`. Для освобождения мьютекса нужно просто сбросить флаг. Реализация приведена в листинге ниже.

Листинг 5.1. Реализация мьютекса-спинлока с использованием `std::atomic_flag`

```
class spinlock_mutex {
 std::atomic_flag flag;
public:
 spinlock_mutex():
  flag(ATOMIC_FLAG_INIT) {}

 void lock() {
  while (flag.test_and_set(std::memory_order_acquire));
 }

 void unlock() {
  flag.clear(std::memory_order_release);
 }
};
```

Это очень примитивный мьютекс, но даже его достаточно для использования в сочетании с шаблоном `std::lock_guard<>` (см. главу 3). По своей природе, он активно ожидает в функции-члене `lock()`, поэтому не стоит использовать его, если предполагается хоть какая-то конкуренция, однако задачу взаимного исключения он решает. Когда дело дойдет до семантики упорядочения доступа к памяти, мы увидим, как гарантируется принудительное упорядочение, необходимое для захвата мьютекса. Пример будет приведён в разделе 5.3.6.

Тип `std::atomic_flag` настолько ограничен, что его даже нельзя использовать в качестве обычного булевского флага, так как он не допускает проверки без изменения значения. На эту роль больше подходит тип `std::atomic<bool>`, который я рассмотрю ниже.

5.2.3. Операции над `std::atomic<bool>`

Из атомарных целочисленных типов простейшим является `std::atomic<bool>`. Как и следовало ожидать, его функциональность в качестве булевского флага богаче, чем у

std::atomic_flag. Хотя копирующий конструктор и оператор присваивания по-прежнему не определены, но можно сконструировать объект из неатомарного bool, поэтому в начальном состоянии он может быть равен как true, так и false. Разрешено также присваивать объектам типа std::atomic<bool> значения неатомарного типа bool:

```
std::atomic<bool> b(true);
b = false;
```

Что касается оператора присваивания с неатомарным bool в правой части, нужно еще отметить отход от общепринятого соглашения о возврате ссылки на объект в левой части — этот оператор возвращает присвоенное значение типа bool. Такая практика обычна для атомарных типов: все поддерживаемые ими операторы присваивания возвращают значения (соответствующего неатомарного типа), а не ссылки. Если бы возвращалась ссылка на атомарную переменную, то программа, которой нужен результат присваивания, должна была бы явно загрузить значение, открывая возможность для модификации результата другим потоком в промежутке между присваиванием и чтением. Получая же результат присваивания в виде неатомарного значения, мы обходимся без дополнительной операции загрузки и можем быть уверены, что получено именно то значение, которое было сохранено.

Запись (любого значения: true или false) производится не чрезмерно ограничительной функцией clear() из класса std::atomic_flag, а путём вызова функции-члена store(), хотя семантику упорядочения доступа к памяти по-прежнему можно задать. Аналогично вместо test_and_set() используется более общая функция-член exchange(), которая позволяет атомарно заменить ранее сохраненное значение новым и вернуть прежнее значение. Тип std::atomic<bool> поддерживает также проверку значения без модификации посредством неявного преобразования к типу bool или явного обращения к функции load(). Как нетрудно догадаться, store() — это операция сохранения, load() — операция загрузки, а exchange() — операция чтения-модификации-записи:

```
std::atomic<bool> b;
bool x = b.load(std::memory_order_acquire);
b.store(true);
x = b.exchange(false, std::memory_order_acq_rel);
```

Функция exchange() — не единственная операция чтения-модификации-записи, которую поддерживает тип std::atomic<bool>; в нем также определена операция сохранения нового значения, если текущее совпадает с ожидаемым.

Сохранение (или несохранение) нового значения в зависимости от текущего

Новая операция называется «сравнить и обменять» и реализована в виде функций-членов compare_exchange_weak() и compare_exchange_strong(). Эта операция — краеугольный камень программирования с использованием атомарных типов; она сравнивает значение атомарной переменной с указанным ожидаемым значением и, если они совпадают, то сохраняет указанное новое значение. Если же значения не совпадают, то ожидаемое значение заменяется фактическим значением атомарной переменной. Функции сравнения и обмена возвращают значение типа bool, равное true, если сохранение было произведено, и false — в противном случае.

В случае compare_exchange_weak() сохранение может не произойти, даже если текущее значение совпадает с ожидаемым. В таком случае значение переменной не изменится, а

функция вернет `false`. Такое возможно на машинах, не имеющих аппаратной команды сравнить-и-обменять, если процессор не может гарантировать атомарности операции — например, потому что поток, в котором операция выполнялась, был переключён в середине требуемой последовательности команд и замещен другим потоком (когда потоков больше, чем процессоров). Эта ситуация называется *ложным отказом*, потому что причиной отказа являются не значения переменных, а хронометраж выполнения функции.

Поскольку `compare_exchange_weak()` может стать жертвой ложного отказа, обычно ее вызывают в цикле:

```
bool expected = false;
extern atomic<bool> b; // установлена где-то в другом месте
while (!b.compare_exchange_weak(expected, true) && !expected);
```

Этот цикл продолжается, пока `expected` равно `false`, что указывает на ложный отказ `compare_exchange_weak()`.

С другой стороны, `compare_exchange_strong()` гарантированно возвращает `false` только в том случае, когда текущее значение не было равно ожидаемому (`expected`). Это устраняет необходимость в показанном выше цикле, когда нужно только узнать, удалось ли нам изменить переменную или другой поток добрался до нее раньше.

Если мы хотим изменить переменную, каким бы ни было ее текущее значение (при этом новое значение может зависеть от текущего), то обновление `expected` оказывается полезной штукой; на каждой итерации цикла `expected` перезагружается, так что если другой поток не модифицирует значение в промежутке, то вызов `compare_exchange_weak()` или `compare_exchange_strong()` должен оказаться успешным на следующей итерации. Если новое сохраняемое значение вычисляется просто, то выгоднее использовать `compare_exchange_weak()`, чтобы избежать двойного цикла на платформах, где `compare_exchange_weak()` *может* давать ложный отказ (и, следовательно, `compare_exchange_strong()` содержит цикл). С другой стороны, если вычисление нового значения занимает длительное время, то имеет смысл использовать `compare_exchange_strong()`, чтобы не вычислять значение заново, когда `expected` не изменилась. Для типа `std::atomic<bool>` это не столь существенно — в конце концов, есть всего два возможных значения — но для более широких атомарных типов различие может оказаться заметным.

Функции сравнения и обмена необычны еще и тем, что могут принимать *два* параметра упорядочения доступа к памяти. Это позволяет по-разному задавать семантику упорядочения в случае успеха и отказа; быть может, при успешном вызове требуется семантика `memory_order_acq_rel`, а при неудачном — `memory_order_relaxed`. В случае отказа функция сохранить-и-обменять не производит сохранение, поэтому семантика `memory_order_release` или `memory_order_acq_rel` неприменима. Поэтому задавать эти варианты упорядочения для отказа не разрешается. Кроме того, нельзя задавать для отказа более строгое упорядочение, чем для успеха; если вы требуете семантику `memory_order_acquire` или `memory_order_seq_cst` в случае отказа, то должны потребовать такую же и в случае успеха.

Если упорядочение для отказа не задано, то предполагается, что оно такое же, как для успеха, с тем отличием, что часть release заменяется: `memory_order_release` становится `memory_order_relaxed`, а `memory_order_acq_rel` — `memory_order_acquire`. Если не задано ни одно упорядочение, то как обычно предполагается `memory_order_seq_cst`, то есть полное последовательное упорядочение доступа как в случае успеха, так и в случае отказа. Следующие два вызова `compare_exchange_weak()` эквивалентны:

```
std::atomic<bool> b;
bool expected;
b.compare_exchange_weak(expected, true,
memory_order_acq_rel, memory_order_acquire);
b.compare_exchange_weak(expected, true, memory_order_acq_rel);
```

К чему приводит задание того или иного упорядочения, я расскажу в разделе 5.3.

Еще одно отличие `std::atomic<bool>` от `std::atomic_flag` заключается в том, что тип `std::atomic<bool>` не обязательно свободен от блокировок; для обеспечения атомарности реализация библиотеки может захватывать внутренний мьютекс. В тех редких случаях, когда это важно, можно с помощью функции-члена `is_lock_free()` узнать, являются ли операции над `std::atomic<bool>` свободными от блокировок. Это еще одна особенность, присущая всем атомарным типам, кроме `std::atomic_flag`.

Следующими по простоте являются атомарные специализации указателей `std::atomic<T*>`.

5.2.4. Операции над `std::atomic<T*>`: арифметика указателей

Атомарная форма указателя на тип `T` — `std::atomic<T*>` — выглядит так же, как атомарная форма `bool` (`std::atomic<bool>`). Интерфейс по существу такой же, только операции применяются к указателям на значения соответствующего типа, а не к значениям типа `bool`. Как и в случае `std::atomic<bool>`, копирующие конструктор и оператор присваивания не определены, но разрешено конструирование и присваивание на основе подходящих указателей. Помимо обязательной функции `is_lock_free()`, тип `std::atomic<T*>` располагает также функциями `load()`, `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()` с такой же семантикой, как `std::atomic<bool>`, но принимаются и возвращаются значения типа `T*`, а не `bool`.

Новыми в типе `std::atomic<T*>` являются арифметические операции над указателями. Базовые операции предоставляются функциями-членами `fetch_add()` и `fetch_sub()`, которые прибавляют и вычитают целое число из сохраненного адреса, а также операторы `+=`, `-=`, `++` и `--` (последние в обеих формах — пред и пост), представляющие собой удобные обертки вокруг этих функций. Операторы работают так же, как для встроенных типов: если `x` — указатель `std::atomic<Foo*>` на первый элемент массива объектов типа `Foo`, то после выполнения оператора `x+=3` `x` будет указывать на четвертый элемент и при этом возвращается простой указатель `Foo*`, который также указывает на четвертый элемент. Функции `fetch_add()` и `fetch_sub()` отличаются от операторов тем, что возвращают старое значение (то есть `x.fetch_add(3)` изменит `x`, так что оно будет указывать на четвертый элемент, но вернет указатель на первый элемент массива). Эту операцию еще называют *обменять-и-прибавить*, она относится к категории атомарных операций чтения-модификации-записи, наряду с `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`. Как и другие операции такого рода, `fetch_add()` возвращает простой указатель `T*`, а не ссылку на объект `std::atomic<T*>`, поэтому вызывающая программа может выполнять действия над прежним значением:

```
class Foo{};
Foo some_array[5];                          | Прибавить 2 к p
std::atomic<Foo*> p(some_array);            | и вернуть старое
Foo* x = p.fetch_add(2);                    ←┘ значение
```

```
assert(x == some_array);
assert(p.load() == &some_array[2]);
x = (p -= 1);                          ←┐ Вычесть 1 из p
assert(x == &some_array[1]);            │ и вернуть новое
assert(p.load() == &some_array[1]);     │ значение
```

Функциям можно также передать в дополнительном аргументе семантику упорядочения доступа к памяти:

```
p.fetch_add(3, std::memory_order_release);
```

Поскольку `fetch_add()` и `fetch_sub()` — операции чтения-модификации-записи, то они принимают любую семантику упорядочения и могут участвовать в *последовательности освобождений*. Для операторных форм задать семантику невозможно, поэтому предполагается семантика `memory_order_seq_cst`.

Все прочие атомарные типы по существу одинаковы: это атомарные целочисленные типы с общим интерфейсом, различаются они только ассоциированными встроенными типами. Поэтому я опишу их все сразу.

5.2.5. Операции над стандартными атомарными целочисленными типами

Помимо обычного набора операций (`load()`, `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`), атомарные целочисленные типы такие, как `std::atomic<int>` или `std::atomic<unsigned long long>`) обладают целым рядом дополнительных операций: `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, `fetch_xor()`, их вариантами в виде составных операторов присваивания (`+=`, `-=`, `&=`, `|=`, `^=`) и операторами пред- и постинкремента и декремента (`++x`, `x++`, `--x`, `x--`). Это не весь набор составных операторов присваивания, имеющихся у обычного целочисленного типа, но близко к тому — отсутствуют лишь операторы умножения, деления и сдвига. Поскольку атомарные целочисленные значения обычно используются в качестве счетчиков или битовых масок, потеря не слишком велика, а в случае необходимости недостающие операции можно реализовать с помощью вызова функции `compare_exchange_weak()` в цикле.

Семантика операций близка к семантике функций `fetch_add()` и `fetch_sub()` в типе `std::atomic<T*>`; именованные функции выполняют свои операции атомарно и возвращают *старое* значение, а составные операторы присваивания возвращают *новое* значение. Операторы пред- и постинкремента и декремента работают как обычно: `++x` увеличивает значение переменной на единицу и возвращает новое значение, а `x++` увеличивает значение переменной на единицу и возвращает старое значение. Как вы теперь уже понимаете, результатом в обоих случаях является значение ассоциированного целочисленного типа.

Мы рассмотрели все простые атомарные типы; остался только основной обобщенный шаблон класса `std::atomic<>` без специализации.

5.2.6. Основной шаблон класса `std::atomic<>`

Наличие основного шаблона позволяет создавать атомарные варианты пользовательских типов, в дополнение к стандартным атомарным типам. Однако в качестве параметра шаблона `std::atomic<>` может выступать только тип, удовлетворяющий определенным условиям. Точнее, чтобы тип `UDT` мог использоваться в конструкции `std::atomic<UDT>`, в

нем должен присутствовать *тривиальный* оператор присваивания. Это означает, что в типе не должно быть виртуальных функций или виртуальных базовых классов, а оператор присваивания должен генерироваться компилятором. Более того, в каждом базовом классе и нестатическом члене данных также должен быть тривиальный оператор присваивания. Это позволяет компилятору использовать для присваивания функцию `memcpy()` или эквивалентную ей, поскольку исполнять написанный пользователем код не требуется.

Наконец, тип должен допускать *побитовое сравнение на равенство*. Это требование из того же разряда, что требования к присваиванию — должна быть не только возможность колировать объекты с помощью `memcpy()`, но и сравнивать их с помощью `memcmp()`. Это необходимо для правильной работы операции сравнить-и-обменять.

Чтобы понять, чем вызваны такие ограничения, вспомните рекомендацию из главы 3: не передавать ссылки и указатели на защищенные данные за пределы области видимости в виде аргументов предоставленной пользователем функции. В общем случае компилятор не в состоянии сгенерировать свободный от блокировок код для типа `Std::atomic<UDT>`, поэтому он вынужден применять внутренние блокировки. Если бы пользовательские операторы присваивания и сравнения были разрешены, то пришлось бы передавать ссылку на защищенные данные в пользовательскую функцию, нарушая тем самым приведённую выше рекомендацию. Кроме того, библиотека вправе использовать единую блокировку для всех нуждающихся в ней атомарных операций, поэтому, разрешив вызывать пользовательские функции в момент, когда эта блокировка удерживается, мы могли бы получить взаимоблокировку или надолго задержать другие потоки, если сравнение занимает много времени. Наконец, эти ограничения повышают шансы на то, что компилятор сумеет сгенерировать для `std::atomic<UDT>` код, содержащий истинно атомарные команды (и тем самым обойтись в данной конкретизации вообще без блокировок), поскольку в этой ситуации он вправе рассматривать определенный пользователем тип как неструктурированную последовательность байтов.

Отметим, что несмотря на то, что типы `std::atomic<float>` и `std::atomic<double>` формально разрешены, так как встроенные типы с плавающей точкой удовлетворяют сформулированным выше критериям на использование `memcpy` и `memcmp`, их поведение в части функции `compare_exchange_strong` может оказаться неожиданным. Операция может завершиться отказом, даже если ранее сохраненное значение численно равно ожидаемому, но имеет другое внутреннее представление. Отметим также, что над числами с плавающей точкой не определены атомарные арифметические операции. Аналогичное поведение `compare_exchange_strong` вы получите, если конкретизируете `std::atomic<>` пользовательским типом, в котором оператор сравнения на равенство определён, но отличается от сравнения с помощью `memcmp` — операция может завершиться отказом, потому что равные значения имеют различное представление.

Если размер пользовательского типа `UDT` равен (или меньше) размеру `int` или `void*`, то на большинстве платформ для типа `std::atomic<UDT>` можно сгенерировать код, содержащий только атомарные команды. На некоторых платформах подобный код можно сгенерировать и в случае, когда размер пользовательского типа в два раза превышает размер `int` или `void*`. Обычно это платформы, на которых имеется команда сравнения и обмена двойных слов *double-word-compare-and-swap (DWCAS)*, соответствующая функциям `compare_exchange_xxx`.

В главе 7 мы увидим, что такая поддержка может быть полезна для написания кода без

блокировок. В силу описанных ограничений вы не можете создать, к примеру, тип `std::atomic<std::vector<int>>`, но можете использовать для параметризации классы, содержащие счетчики, флаги, указатели и даже массивы простых элементов. Обычно это не проблема; чем сложнее структура данных, тем больше вероятность, что в ней нужно будет определить какие-то другие операции, помимо простейшего присваивания и сравнения. Но в таком случае лучше воспользоваться классом `std::mutex`, который гарантирует надлежащую защиту данных при выполнении этих операций (см. главу 3).

Интерфейс шаблона `std::atomic<T>`, конкретизированного пользовательским типом `T`, ограничен набором операций, доступных классу `std::atomic<bool>`: `load()`, `store()`, `exchange()`, `compare_exchange_weak()`, `compare_exchange_strong()`, присваивание значения типа `T` и преобразование в значение типа `T`.

В табл. 5.3 перечислены операции, доступные для всех атомарных типов.

Таблица 5.3. Операции над атомарными типами

Операция	atomic_flag	atomic <bool>	atomic <T*>	atomic <integral-type>	atomic <other-type>
test_and_set	√				
clear	√				
is_lock_free		√	√	√	√
load		√	√	√	√
store		√	√	√	√
exchange		√	√	√	√
compare_exchange_weak, compare_exchange_strong		√	√	√	√
fetch_add, +=			√	√	
fetch_sub, -=			√	√	
fetch_or, |=				√	
fetch_and, &=				√	
fetch_xor, ^=				√	
++, --			√	√	

5.2.7. Свободные функции для атомарных операций

До сих пор я описывал только те операции над атомарными типами, которые реализованы функциями-членами. Однако для всех этих операций существуют также эквивалентные функции, не являющиеся членами классов. Как правило, имена свободных функций строятся по единому образцу: имя соответствующей функции-члена с префиксом `atomic_` (например, `std::atomic_load()`). Затем эти функции перегружаются для каждого атомарного типа. Если имеется возможность задать признак упорядочения доступа к памяти, то предлагаются две разновидности функции: одна без признака, другая — ее имя заканчивается суффиксом `_explicit` — с одним или несколькими дополнительными параметрами для задания признаков (например, `std::atomic_store(&atomic_var,`

new_value) и std::atomic_store_explicit(&atomic_var, new_value, std::memory_order_release). Если в случае функций-членов объект атомарного типа задается неявно, то все свободные функции принимают в первом параметре указатель на такой объект.

Например, для функции std::atomic_is_lock_free() есть только одна разновидность (хотя и перегруженная для всех типов), причём std::atomic_is_lock_free(&a) возвращает то же значение, что a.is_lock_free() для объекта a атомарного типа. Аналогично std::atomic_load(&a) — то же самое, что a.load(), а эквивалентом a.load(std::memory_order_acquire) является std::atomic_load_explicit(&a, std::memory_order_acquire).

Свободные функции совместимы с языком C, то есть во всех случаях принимают указатели, а не ссылки. Например, первый параметр функций-членов compare_exchange_weak() и compare_exchange_strong() (ожидаемое значение) — ссылка, но вторым параметром std::atomic_compare_exchange_weak() (первый — это указатель на объект) является указатель. Функция std::atomic_compare_exchange_weak_explicit() также требует задания двух параметров, определяющих упорядочение доступа к памяти в случае успеха и отказа, тогда как функции-члены для сравнения с обменом имеют варианты как с одним параметром (второй по умолчанию равен std::memory_order_seq_cst), так и с двумя.

Операции над типом std::atomic_flag нарушают традицию, поскольку в именах функций присутствует дополнительное слово «flag»: std::atomic_flag_test_and_set(), std::atomic_flag_clear(), но у вариантов с параметрами, задающими упорядочение доступа, суффикс _explicit по-прежнему имеется: std::atomic_flag_test_and_set_explicit() и std::atomic_flag_clear_explicit().

В стандартной библиотеке C++ имеются также свободные функции для атомарного доступа к экземплярам типа std::shared_ptr<>. Это отход от принципа, согласно которому атомарные операции поддерживаются только для атомарных типов, поскольку тип std::shared_ptr<> заведомо *не* атомарный. Однако комитет по стандартизации C++ счел этот случай достаточно важным, чтобы предоставить дополнительные функции. К числу определенных для него атомарных операций относятся *загрузка*, *сохранение*, *обмен* и *сравнение с обменом*, и реализованы они в виде перегрузок тех же операций над стандартными атомарными типами, в которых первым аргументом является указатель std::shared_ptr<>*:

```
std::shared_ptr<my_data> p;

void process_global_data() {
 std::shared_ptr<my_data> local = std::atomic_load(&p);
 process_data(local);
}

void update_global_data() {
 std::shared_ptr<my_data> local(new my_data);
 std::atomic_store(&p, local);
}
```

Как и для атомарных операций над другими типами, предоставляются _explicit-варианты, позволяющие задать необходимое упорядочение, а для проверки того, используется ли в реализации внутренняя блокировка, имеется функция

`std::atomic_is_lock_free()`.

Как отмечалось во введении, стандартные атомарные типы позволяют не только избежать неопределённого поведения, связанного с гонкой за данные; они еще дают возможность задать порядок операций в потоках. Принудительное упорядочение лежит в основе таких средств защиты данных и синхронизации операций, как `std::mutex` и `std::future<>`. Помня об этом, перейдём к материалу, составляющему главное содержание этой главы: аспектам модели памяти, относящимся к параллелизму, и тому, как с помощью атомарных операций можно синхронизировать данные и навязать порядок доступа к памяти.

5.3. Синхронизация операций и принудительное упорядочение

Пусть имеются два потока, один из которых заполняет структуру данных, а другой читает ее. Чтобы избежать проблематичного состояния гонки, первый поток устанавливает флаг, означающий, что данные готовы, а второй не приступает к чтению данных, пока этот флаг не установлен. Описанный сценарий демонстрируется в листинге ниже.

Листинг 5.2. Запись и чтение переменной в разных потоках

```
#include <vector>
#include <atomic>
#include <iostream>

std::vector<int> data;
std::atomic<bool> data_ready(false);

void reader_thread() {
 while (!data_ready.load()) {                ← (1)
  std::this_thread::sleep(std::milliseconds(1));
 }
 std::cout << "Ответ=" << data[0] << "\n";← (2)
}

void writer_thread() {
 data.push_back(42); ← (3)
 data_ready = true;  ← (4)
}
```

Оставим пока в стороне вопрос о неэффективности цикла ожидания готовности данных **(1)**. Для работы этой программы он действительно необходим, потому что в противном случае разделение данных между потоками становится практически бесполезным: каждый элемент данных должен быть атомарным. Вы уже знаете, что неатомарные операции чтения **(2)** и записи **(3)** одних и тех же данных без принудительного упорядочения приводят к неопределённому поведению, поэтому где-то упорядочение должно производиться, иначе ничего работать не будет.

Требуемое упорядочение обеспечивают операции с переменной `data_ready` типа `std::atomic<bool>` и делается это благодаря отношениям *происходит-раньше* и *синхронизируется-с*, заложенным в модель памяти. Запись данных **(3)** происходит-раньше записи флага `data_ready` **(4)**, а чтение флага **(1)** происходит-раньше чтения данных **(2)**. Когда прочитанное значение `data_ready` **(1)** равно `true`, операция записи синхронизируется-с этой операцией чтения, что приводит к порождению отношения происходит-раньше. Поскольку отношение происходит-раньше транзитивно, то запись данных **(3)** происходит-раньше записи флага **(4)**, которая происходит-раньше чтения значения `true` из этого флага **(1)**, которое в свою очередь происходит-раньше чтения данных **(2)**. И таким образом мы получаем принудительное упорядочение: запись данных происходит-раньше чтения данных, и программа работает правильно. На рис. 5.2 изображены важные отношения происходит-раньше в обоих потоках. Я включил две итерации цикла `while` в потоке-читателе.

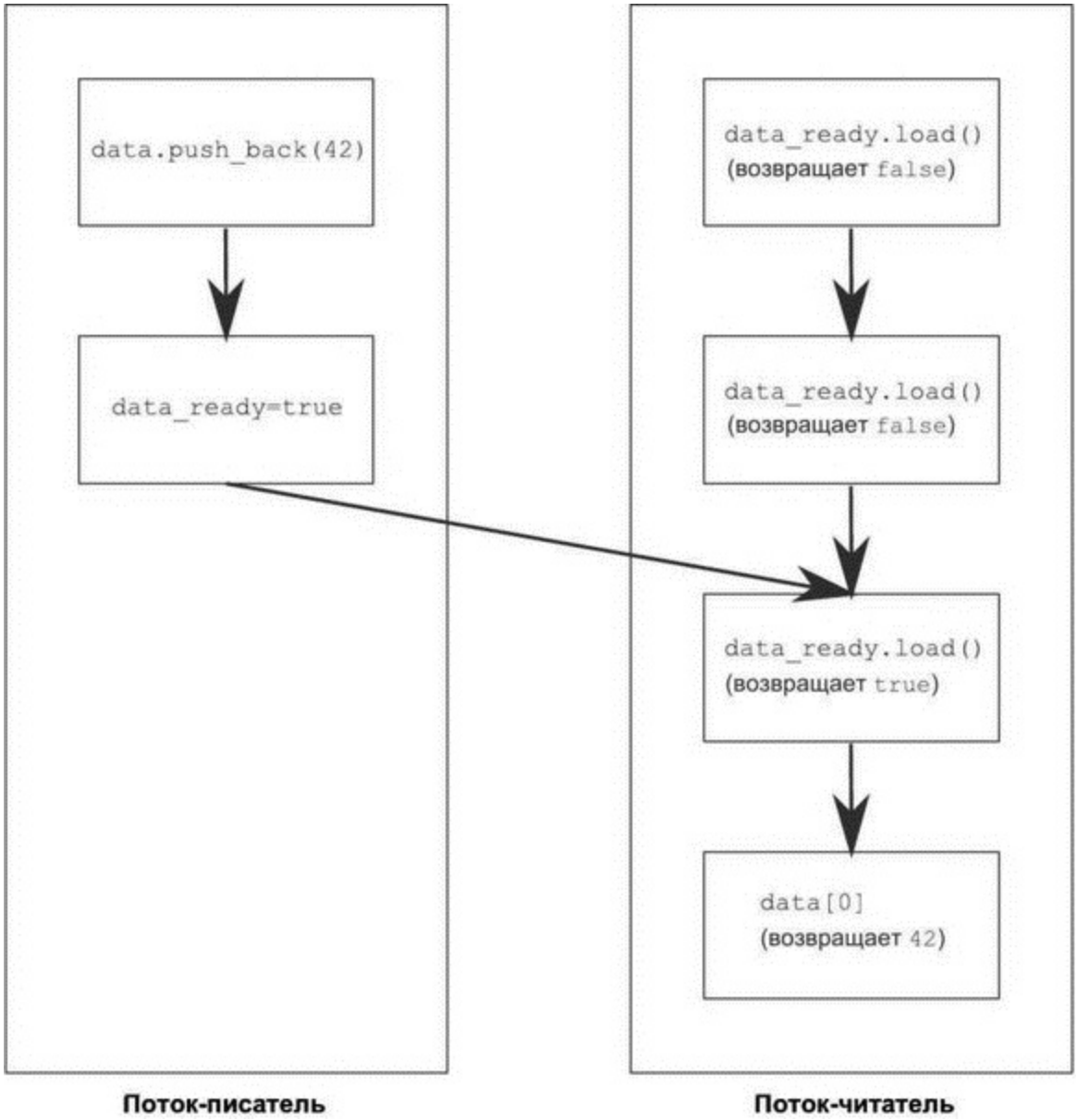

Рис. 5.2. Принудительное задание упорядочения неатомарных операций с помощью атомарных

Все это может показаться интуитивно очевидным — разумеется, операция записи значения происходит раньше операции его чтения! В случае атомарных операций по умолчанию это действительно так (на то и умолчания), однако подчеркну: у атомарных операций есть и другие возможности для задания требований к упорядочению, и скоро я о них расскажу.

Теперь, когда вы видели, как отношения происходит-раньше и синхронизируется-с работают на практике, имеет смысл поговорить о том, что же за ними стоит. Начнем с отношения синхронизируется-с.

5.3.1. Отношение синхронизируется-с

Отношение синхронизируется-с возможно только между операциями над атомарными типами. Операции над структурой данных (например, захват мьютекса) могут обеспечить это отношение, если в структуре имеются атомарные типы и определенные в ней операции выполняют необходимые атомарные операции. Однако реальным источником синхронизации всегда являются операции над атомарными типами.

Идея такова: подходящим образом помеченная атомарная операция записи `W` над переменной `x` синхронизируется-с подходящим образом помеченной атомарной операцией чтения над переменной `x`, которая читает значение, сохраненное либо данной операцией записи (`W`), либо следующей за ней атомарной операцией записи над `x` в том же потоке, который выполнил первоначальную операцию `W`, либо последовательностью атомарных операций чтения-модификации-записи над `x` (например, `fetch_add()` или `compare_exchange_weak()`) в любом потоке, при условии, что значение, прочитанное первым потоком в этой последовательности, является значением, записанным операцией `W` (см. раздел 5.3.4).

Пока оставим в стороне слова «подходящим образом помеченная», потому что по умолчанию все операции над атомарными типами помечены подходящим образом. По существу сказанное выше означает ровно то, что вы ожидаете: если поток A сохраняет значение, а поток B читает это значение, то существует отношение синхронизируется-с между сохранением в потоке A и загрузкой в потоке B — как в листинге 5.2.

Уверен, вы догадались, что нюансы как раз и скрываются за словами «подходящим образом помеченная». Модель памяти в C++ допускает применение различных ограничений на упорядочение к операциям над атомарными типами, и именно это и называется пометкой. Варианты упорядочения доступа к памяти и их связь с отношением синхронизируется-с рассматриваются в разделе 5.3.3. А пока отступим на один шаг и поговорим об отношении происходит-раньше.

5.3.2. Отношение происходит-раньше

Отношение происходит-раньше — основной строительный блок механизма упорядочения операций в программе. Оно определяет, какие операции видят последствия других операций и каких именно. В однопоточной программе всё просто: если в последовательности выполняемых операций одна стоит раньше другой, то она и происходит-раньше. Иначе говоря, если операция A в исходном коде предшествует операции B, то A происходит-раньше B. Это мы видели в листинге 5.2: запись в переменную `data` **(3)** происходит-раньше записи в переменную `data_ready` **(4)**. В общем случае между операциями, которые входят в состав одного предложения языка, нет отношения происходит-раньше, поскольку они не упорядочены. По-другому то же самое можно выразить, сказав, что порядок не определён. Мы знаем, что программа, приведённая в следующем листинге, напечатает "`1,2`" или "`2,1`", но что именно, неизвестно, потому что порядок двух обращений к `get_num()` не определён.

Листинг 5.3. Порядок определения аргументов функции не определён

```
#include <iostream>

void foo(int a, int b) {
 std::cout << a << "," << b << std::endl;
}

int get_num() {
 static int i = 0;
 return ++i;
```

```
}

int main() {
 foo(get_num(), get_num());←┐ Порядок обращений
}                           | к get_num() не определен
```

Существуют случаи, когда порядок операций внутри одного предложения точно известен, например, если используется встроенный оператор «занятая» или результат одного выражения является аргументом другого выражения. Но в общем случае никакого отношения расположено-перед (а, значит, и отношения происходит-раньше) между ними не существует. Разумеется, все операции в одном предложении происходят раньше всех операций в следующем за ним предложении.

Но это просто пересказ другими словами давно известных вам правил упорядочения в однопоточной программе. А где же новое? Новым является взаимодействие между потоками: если операция A в одном потоке межпоточно происходит-раньше операции B в другом потоке, то A происходит-раньше B. Вроде бы толку немного, мы просто добавили новое отношение (межпоточно происходит-раньше), но при написании многопоточной программы это отношение оказывается очень важным.

На понятийном уровне отношение межпоточно происходит-раньше довольно простое, оно опирается на отношение синхронизируется-с, введенное в разделе 5.3.1: если операция A в одном потоке синхронизируется-с операцией B в другом потоке, то A межпоточно происходит-раньше B. Это отношение также транзитивно: если A межпоточно происходит-раньше B, а B межпоточно происходит-раньше C, то A межпоточно происходит-раньше C. Это мы тоже видели в листинге 5.2.

Отношение межпоточно происходит-раньше также комбинируется с отношением расположено-перед: если операция A расположена перед операцией B и операция B межпоточно происходит-раньше операции C, то A межпоточно происходит-раньше C. Аналогично, если A синхронизируется-с B и B расположена-перед C, то A межпоточно происходит-раньше C. В совокупности эти два утверждения означают, что если произведена серия изменений данных в одном потоке, то нужно только одно отношение синхронизируется-с, чтобы данные стали видимы последующим операциям в потоке, где выполнена C.

Именно эти критически важные правила и обеспечивают упорядоченность операций между потоками, благодаря чему программа в листинге 5.2 работает правильно. Как мы скоро увидим, существуют дополнительные нюансы, связанные с зависимостями между данными. Чтобы разобраться в них, мне нужно будет рассмотреть признаки упорядочения доступа к памяти, используемые в атомарных операциях, и рассказать, как они связаны с отношением синхронизируется-с.

5.3.3. Упорядочение доступа к памяти для атомарных операций

Существует шесть вариантов упорядочения доступа к памяти, которые можно задавать в операциях над атомарными типами: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel` и `memory_order_seq_cst`. Если не указано противное, то для любой операции над атомарными типами подразумевается упорядочение `memory_order_seq_cst` — самое ограничительное из

всех. Хотя вариантов шесть, представляют они всего три модели: *последовательно согласованное* упорядочение (`memory_order_seq_cst`), упорядочение *захват-освобождение* (`memory_order_consume`, `memory_order_acquire`, `memory_order_release` и `memory_order_acq_rel`) и *ослабленное* упорядочение (`memory_order_relaxed`).

Эти три модели упорядочения доступа к памяти влекут за собой различные издержки для процессоров с разной архитектурой. Например, в системах с точным контролем над видимостью операций процессорами, отличными от произведшего изменения, могут потребоваться дополнительные команды синхронизации для обеспечения последовательно согласованного упорядочения по сравнению с ослабленным или упорядочением захват-освобождение, а также для обеспечения упорядочения захват-освобождение по сравнению с ослабленным. Если в такой системе много процессоров, то на выполнение дополнительных команд синхронизации может уходить заметное время, что приведет к снижению общей производительности системы. С другой стороны, процессоры с архитектурой x86 или x86-64 (в частности, Intel и AMD, столь распространенные в настольных ПК) не требуют никаких дополнительных команд для обеспечения упорядочения захват-освобождение, помимо необходимых для гарантий атомарности, и даже последовательно согласованное упорядочение не нуждается в каких-то специальных действиях на операциях загрузки, хотя операции сохранения все же требуют некоторых добавочных затрат.

Наличие различных моделей упорядочения доступа к памяти позволяет эксперту добиться повышения производительности за счет более точного управления отношениями упорядочения там, где это имеет смысл, и в то же время использовать последовательно согласованное упорядочение (которое гораздо проще для понимания) в случаях, когда такой выигрыш не критичен.

Чтобы выбрать подходящую модель, нужно понимать, каковы последствия того или иного решения для поведения программы. Поэтому рассмотрим, какое влияние оказывают различные модели на упорядочение операций и отношение синхронизируется-с.

Последовательно согласованное упорядочение

Упорядочение по умолчанию называется *последовательно согласованным*, потому что оно предполагает, что поведение программы согласовано с простым последовательным взглядом на мир. Если все операции над экземплярами атомарных типов последовательно согласованы, то поведение многопоточной программы такое же, как если бы эти операции выполнялись в какой-то определенной последовательности в одном потоке. Это самое простое для понимания упорядочение доступа к памяти, потому оно и подразумевается по умолчанию: все потоки должны видеть один и тот же порядок операций. Таким образом, становится достаточно легко рассуждать о поведении программы, написанной с использованием атомарных переменных. Можно выписать все возможные последовательности операций, выполняемых разными потоками, отбросить несогласованные и проверить, что в остальных случаях программа ведет себя, как и ожидалось. Это также означает, что порядок операций нельзя изменять; если в каком-то потоке одна операция предшествует другой, то этот порядок должен быть виден всем остальным потокам.

С точки зрения синхронизации, последовательно согласованное сохранение

синхронизируется-с последовательно согласованной операцией загрузки той же переменной, в которой читается сохраненное значение. Тем самым мы получаем одно ограничение на упорядочение операций в двух или более потоках. Однако этим последовательная согласованность не исчерпывается. Любая последовательно согласованная операция, выполненная после этой загрузки, должна быть видна всякому другому потоку в системе с последовательно согласованными атомарными операциями именно как следующая за загрузкой. Пример в листинге 5.4 демонстрирует это ограничение на упорядочение в действии. Однако это ограничение не распространяется на потоки, в которых для атомарных операций задано ослабленное упорядочение — они по-прежнему могут видеть операции в другом порядке. Поэтому, чтобы получить пользу от последовательного согласования операций, его надо использовать во всех потоках.

Но за простоту понимания приходится платить. На машине со слабым упорядочением и большим количеством процессоров может наблюдаться заметное снижение производительности, потому что для поддержания согласованной последовательности операций, возможно, придётся часто выполнять дорогостоящие операции синхронизации процессоров. Вместе с тем следует отметить, что некоторые архитектуры процессоров (в частности, такие распространенные, как x86 и x86-64) обеспечивают последовательную согласованность с относительно низкими издержками, так что если вас волнует влияние последовательно согласованного упорядочения на производительность, ознакомьтесь с документацией но конкретному процессору.

В следующем листинге последовательная согласованность демонстрируется на примере. Операции загрузки и сохранения переменных `x` и `y` явно помечены признаком `memory_order_seq_cst`, хотя его можно было бы и опустить, так как он подразумевается по умолчанию.

Листинг 5.4. Из последовательной согласованности вытекает полная упорядоченность

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x, y;
std::atomic<int> z;

void write_x() {
 x.store(true, std::memory_order_seq_cst); ← (1)
}

void write_y() {
 y.store(true, std::memory_order_seq_cst); ← (2)
}

void read_x_then_y() {
 while (!x.load(std::memory_order_seq_cst));← (3)
 if (y.load(std::memory_order_seq_cst))
  ++z;
}

void read_y_then_x() {
```

```
 while (!y.load(std::memory_order_seq_cst));← (4)
 if (x.load(std::memory_order_seq_cst))
  ++z;
}

int main() {
 x = false;
 y = false;
 z = 0;
 std::thread a(write_x);
 std::thread b(write_y);
 std::thread c(read_x_then_y);
 std::thread d(read_y_then_x);
 a.join();
 b.join();
 c.join();
 d.join();
 assert(z.load() != 0); ← (5)
}
```

Утверждение `assert` **(5)** не может сработать, потому что первым должно произойти сохранение `x` **(1)** или сохранение `y` **(2)**, пусть даже точно не сказано, какое именно. Если загрузка `y` в функции `read_x_then_y` **(3)** возвращает `false`, то сохранение `x` должно было произойти раньше сохранения `y`, и в таком случае загрузка `x` в `read_y_then_x` **(4)** должна вернуть `true`, потому что наличие цикла `while` гарантирует, что в этой точке `y` равно `true`. Поскольку семантика `memory_order_seq_cst` требует полного упорядочения всех операций, помеченных признаком `memory_order_seq_cst`, то существует подразумеваемое отношение порядка между операцией загрузки `y`, которая возвращает `false` **(3)**, и операцией сохранения `y` **(1)**. Чтобы имело место единственное полное упорядочение в случае, когда некоторый поток сначала видит `x==true`, затем `y==false`, необходимо, чтобы при таком упорядочении сохранение `x` происходило раньше сохранения `y`.

Разумеется, поскольку всё симметрично, могло бы произойти и ровно наоборот: загрузка `x` **(4)** возвращает `false`, и тогда загрузка `y` **(3)** обязана вернуть `true`. В обоих случаях `z` равно 1. Может быть и так, что обе операции вернут `true`, и тогда `z` будет равно 2. Но ни в каком случае `z` не может оказаться равным нулю.

Операции и отношения происходит-раньше для случая, когда `read_x_then_y` видит, что `x` равно `true`, а `y` равно `false`, изображены на рис. 5.3. Пунктирная линия от операции загрузки `y` в `read_x_then_y` к операции сохранения `y` в `write_y` показывает наличие неявного отношения порядка, необходимого для поддержания последовательной согласованности: загрузка должна произойти раньше сохранения в глобальном порядке операций, помеченных признаком `memory_order_seq_cst`, — только тогда получится показанный на рисунке результат.

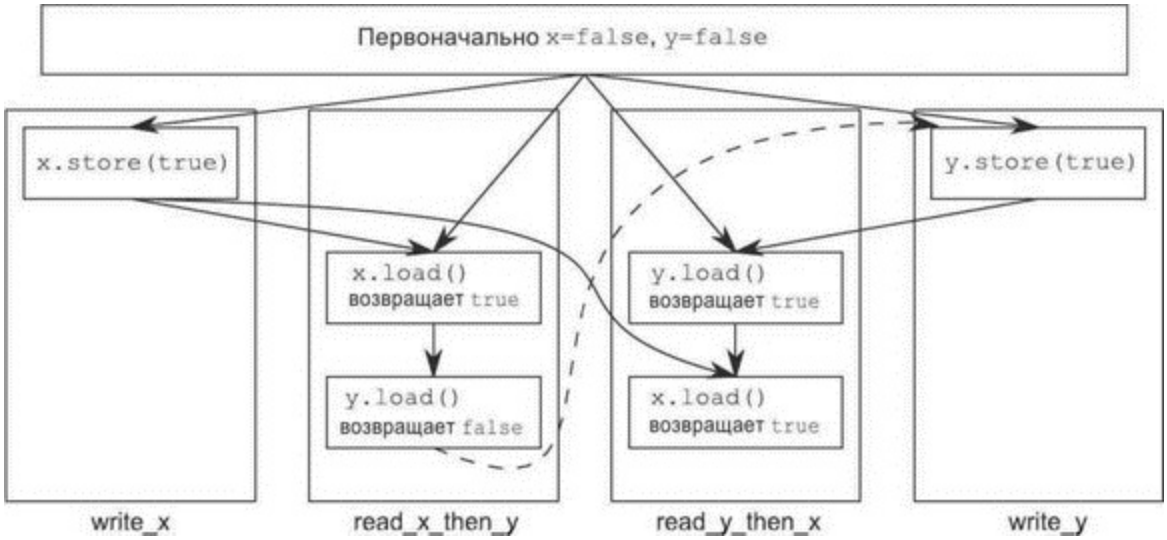

Рис. 5.3. Последовательная согласованность и отношения происходит-раньше

Последовательная согласованность — самое простое и интуитивно понятное упорядочение, но оно же является и самым накладным из- за необходимости глобальной синхронизации между всеми потоками. В многопроцессорной системе это потребовало бы многочисленных и затратных по времени взаимодействий между процессорами. Чтобы избежать затрат на синхронизацию, необходимо выйти за пределы мира последовательной согласованности и рассмотреть другие модели упорядочения доступа к памяти.

Не последовательно согласованное упорядочение доступа к памяти

За пределами уютного последовательно согласованного мирка нас встречает более сложная реальность. И, пожалуй, самое трудное — смириться с тем фактом, что *единого глобального порядка событий больше не существует*. Это означает, что разные потоки могут по-разному видеть одни и те же операции, и с любой умозрительной моделью, предполагающей, что операции, выполняемые в разных потоках, строго перемежаются, следует распрощаться. Вы должны учитывать не только то, что события могут происходить по-настоящему одновременно, но и то, что *потоки не обязаны согласовывать порядок событий между собой*. Чтобы написать (или хотя бы понять) код, в котором используется упорядочение, отличное от `memory_order_seq_cst`, абсолютно необходимо уложить этот факт в мозгу. Мало того что компилятор вправе изменять порядок команд. Даже если потоки исполняют один и тот же код, они могут видеть события в разном порядке, потому что в отсутствие явных ограничений на упорядочение кэши различных процессоров и внутренние буферы могут содержать различные значения для одной и той же ячейки памяти. Это настолько важно, что я еще раз повторю: потоки *не обязаны согласовывать порядок событий между собой*.

Вы должны отбросить мысленные модели, основанные не только на идее чередования операций, но и на представлении о том, что компилятор или процессор изменяет порядок команд. *В отсутствие иных ограничений на упорядочение, единственное требование заключается в том, что все потоки согласны относительно порядка модификации каждой отдельной переменной.* Операции над различными переменными могут быть видны разным потокам в разном порядке при условии, что видимые значения согласуются с наложенными дополнительными ограничениями на упорядочение.

Проще всего это продемонстрировать, перейдя от последовательной согласованности к ее полной противоположности — упорядочению `memory_order_relaxed` для всех операций. Освоив этот случай, мы сможем вернуться к упорядочению захват-освобождение, которое позволяет избирательно вводить некоторые отношения порядка между операциями. Это хоть как-то поможет собрать разлетевшиеся мозги в кучку.

Ослабленное упорядочение

Операции над атомарными типами, выполняемые в режиме ослабленного упорядочения, не участвуют в отношениях синхронизируется-с. Операции над одной и той же переменной в одном потоке по-прежнему связаны отношением происходит-раньше, но на относительный порядок операций в разных потоках не накладывается почти никаких ограничений. Есть лишь одно требование: операции доступа к одной атомарной переменной в одном и том же потоке нельзя переупорядочивать — если данный поток видел определенное значение атомарной переменной, то последующая операция чтения не может извлечь предыдущее значение этой переменной. В отсутствие дополнительной синхронизации порядок модификации отдельных переменных — это единственное, что объединяет потоки, использующие модель `memory_order_relaxed`.

Чтобы продемонстрировать, до какой степени могут быть «ослаблены» операции в этой модели, достаточно всего двух потоков (см. листинг 5.5).

Листинг 5.5. К ослабленным операциям предъявляются очень слабые требования

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y() {
 x.store(true, std::memory_order_relaxed); ← (1)
 y.store(true, std::memory_order_relaxed); ← (2)
}

void read_y_then_x() {
 while (!y.load(std::memory_order_relaxed));← (3)
 if (x.load(std::memory_order_relaxed))     ← (4)
  ++z;
}

int main() {
 x = false;
 y = false;
 z = 0;
 std::thread a(write_x_then_y);
 std::thread b(read_y_then_x);
 a.join();
 b.join();
```

```
 assert (z.load() != 0); ← (5)
}
```

На этот раз утверждение **(5)** *может* сработать, потому что операция загрузки `x` **(4)** может прочитать `false`, даже если загрузка `y` **(3)** прочитает `true`, а сохранение `x` **(1)** происходит-раньше сохранения `y` **(2)**. `x` и `y` — разные переменные, поэтому нет никаких гарантий относительно порядка видимости результатов операций над каждой из них.

Ослабленные операции над разными переменными можно как угодно переупорядочивать при условии, что они подчиняются ограничивающим отношениям происходит-раньше (например, действующим внутри одного потока). Никаких отношений синхронизируется-с не возникает. Отношения происходит-раньше, имеющиеся в листинге 5.5, изображены на рис. 5.4, вместе с возможным результатом. Несмотря на то, что существует отношение происходит-раньше отдельно между операциями сохранения и операциями загрузки, не существует ни одного такого отношения между любым сохранением и любой загрузкой, поэтому операция загрузки может увидеть операции сохранения не в том порядке, в котором они происходили.

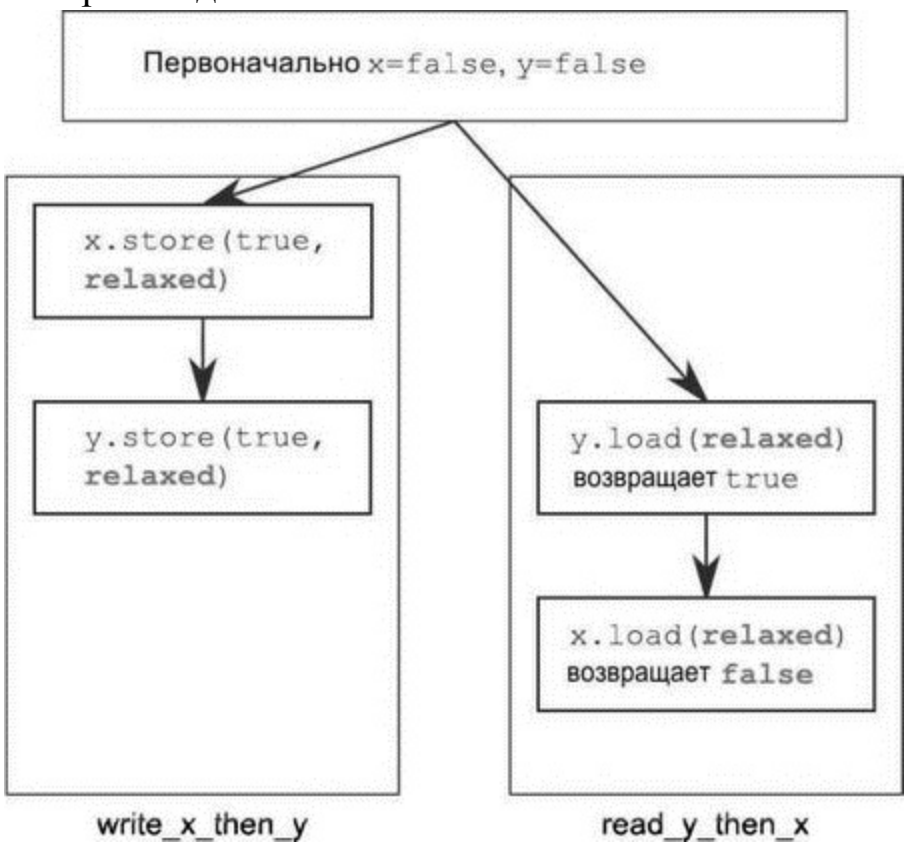

Рис. 5.4. Ослабленные атомарные операции и отношения происходит-раньше

Рассмотрим чуть более сложный пример с тремя переменными и пятью потоками.

Листинг 5.6. Ослабленные операции в нескольких потоках

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic<int> x(0), y(0), z(0);← (1)
std::atomic<bool> go(false);      ← (2)

unsigned const loop_count = 10;
```

```
struct read_values {
 int x, y, z;
};

read_values values1[loop_count];
read_values values2[loop_count];
read_values values3[loop_count];
read_values values4[loop_count];
read_values values5[loop_count];

void increment(
 std::atomic<int>* var_to_inc, read_values* values) {
 while (!go) ←(3) В цикле ждем сигнала
  std::this_thread::yield();
 for (unsigned i = 0; i < loop_count; ++i) {
  values[i].x = x.load(std::memory_order_relaxed);
  values[i].y = y.load(std::memory_order_relaxed);
  values[i].z = z.load(std::memory_order_relaxed);
  var_to_inc->store(i + 1, std::memory_order_relaxed);←(4)
  std::this_thread::yield();
 }
}

void read_vals(read_values* values) {
 while (!go) ←(5) В цикле ждем сигнала
 std::this_thread::yield();
 for (unsigned i = 0; i < loop_count; ++i) {
  values[i].x = x.load(std::memory_order_relaxed);
  values[i].y = y.load(std::memory_order_relaxed);
  values[i].z = z.load(std::memory_order_relaxed);
  std::this_thread::yield();
 }
}

void print(read_values* v) {
 for (unsigned i = 0; i < loop_count; ++i) {
  if (i)
   std::cout << ",";
  std::cout <<
   "(" << v [i] .x << "," << v[i].y << "," << v[i].z << ")";
 }
 std::cout << std::endl;
}

int main() {
 std::thread t1(increment, &x, values1);
 std::thread t2(increment, &y, values2);
 std::thread t3(increment, &z, values3);
 std::thread t4(read_vals, values4);
 std::thread t5(read_vals, values5);

 go = true; ←┐ Сигнал к началу выполнения
            |  (6) главного цикла
```

```
    t5.join();
    t4.join();
    t3.join();
    t2.join();
    t1.join();

    print(values1);←┐
    print(values2); | Печатаем получившиеся
    print(values3); (7) значения
    print(values4);
    print(values5);
  }
```

По существу, это очень простая программа. У нас есть три разделяемых глобальных атомарных переменных **(1)** и пять потоков. Каждый поток выполняет 10 итераций цикла, читая значения трех атомарных переменных в режиме `memory_order_relaxed` и сохраняя их в массиве. Три из пяти потоков обновляют одну из атомарных переменных при каждом проходе по циклу **(4)**, а остальные два только читают ее. После присоединения всех потоков мы распечатываем массивы, заполненные каждым из них **(7)**.

Атомарная переменная `go` **(2)** служит для того, чтобы все потоки начали работу по возможности одновременно. Запуск потока — накладная операция и, не будь явной задержки, первый поток мог бы завершиться еще до того, как последний зачал работать. Каждый поток ждет, пока переменная `go` станет равна `true`, и только потом входит в главный цикл **(3)**, **(5)**, а переменная `go` устанавливается в `true` только после запуска всех потоков **(6)**.

Ниже показан один из возможных результатов прогона этой прогона:

```
    (0,0,0),(1,0,0),(2,0,0),(3,0,0),(4,0,0),(5,7,0),(6,7,8),(7,9,8),(8,9,8),
(9,9,10)
    (0,0,0),(0,1,0),(0,2,0),(1,3,5),(8,4,5),(8,5,5),(8,6,6),(8,7,9),(10,8,9),
(10,9,10)
    (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6),(0,0,7),(0,0,8),
(0,0,9)
    (1,3,0),(2,3,0),(2,4,1),(3,6,4),(3,9,5),(5,10,6),(5,10,8),(5,10,10),
(9,10,10),(10,10,10)
    (0,0,0),(0,0,0),(0,0,0),(6,3,7),(6,5,7),(7,7,7),(7,8,7),(8,8,7),(8,8,9),
(8,8,9)
```

Первые три строки относятся к потокам, выполнявшим обновление, последние две — к потокам, которые занимались только чтением. Каждая тройка — это значения переменных `x`, `y`, `z` в порядке итераций цикла. Следует отметить несколько моментов.

• В первом наборе значения `x` увеличиваются на 1 в каждой тройке, во втором наборе на 1 увеличиваются значения `y`, а в третьем — значения `z`.

• Значения `x` (а равно `y` и `z`) увеличиваются только в пределах данного набора, но приращения неравномерны и относительный порядок в разных наборах различен.

• Поток 3 не видит обновлений `x` и `y`, ему видны только обновления `z`. Но это не мешает другим потокам видеть обновления `z` наряду с обновлениями `x` и `y`.

Это всего лишь один из возможных результатов выполнения ослабленных операций. Вообще говоря, возможен любой результат, в котором каждая из трех переменных принимает значения от 0 до 10, и в каждом потоке, обновляющем некоторую переменную, ее значения монотонно изменяются от 0 до 9.

Чтобы попять, как всё это работает, представьте, что каждая переменная — человек с блокнотом, сидящий в отдельном боксе. В блокноте записана последовательность значений. Вы можете позвонить сидельцу и попросить либо прочитать вслух какое-нибудь значение, либо записать новое. Новое значение он записывает в конец последовательности.

При первой просьбе дать значение человек может прочитать *любое* значение из списка, имеющегося в данный момент. В ответ на следующую просьбу он может прочитать либо то же самое значение, либо значение, расположенное позже него в списке, но никогда — значение, расположенное раньше уже прочитанного. Если вы просили записать значение, а потом прочитать, то он может сообщить либо значение, записанное в ответ на вашу просьбу, либо расположенное позже него в списке.

Теперь представьте, что в начале списка находятся значения 5, 10, 23, 3, 1, 2. Человек может прочитать любое из них. Если он скажет 10, то в следующий раз он может прочитать также 10 или любое последующее число, но не 5. Если вы позвоните пять раз, то может услышать, например, последовательность «10, 10, 1, 2, 2». Если вы попросите записать 42, он добавит это число в конец списка. Если вы затем будете просить прочитать число, то он будет повторять «42», пока в списке не появится новое число и он не захочет назвать его.

Предположим далее, что у Карла тоже есть телефон этого человека. Карл тоже может позволить ему с просьбой либо прочитать, либо записать число. При этом к Карлу применяются те же правила, что и к вам. Телефон только один, поэтому в каждый момент времени человек общается только с одним из вас, так что список в его блокноте растет строго последовательно. Но из того, что вы попросили записать его новое число, вовсе не следует, что он должен сообщить его Карлу. и наоборот. Если Карл попросил назвать число и услышал в ответ «23», то из того, что вы попросили записать число 42, не вытекает, что в следующий раз Карл услышит его. Человек может назвать Карлу любое из чисел 23, 3, 1, 2, 42 или даже 67, если после вас позвонил Фред и попросил записать это число. Он даже может назвать Карлу последовательность «23, 3, 3, 1, 67», и это не будет противоречить тому, что услышали вы. Можно представить себе, что человек запоминает, какое число кому назвал, сдвигая указатели, на которых написано имя спрашивающего, как показано на рис. 5.5.

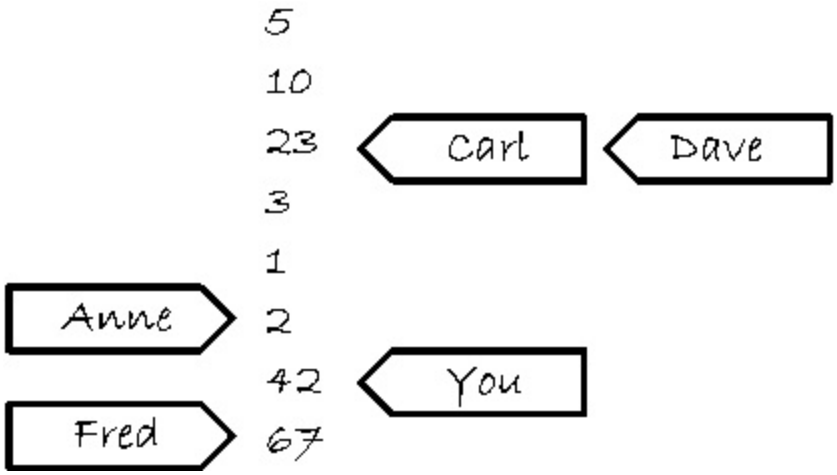

Рис. 5.5. Блокнот человека, сидящего в боксе

Теперь представьте, что имеется целый ряд боксов, в каждом из которых сидит по человеку с блокнотом и телефоном. Это всё наши атомарные переменные. У каждой переменной свой порядок модификации (список значений в блокноте), по между ними нет

никакой связи. Если каждый звонящий (вы, Карл, Анна, Дэйв и Фред) представляет поток, то именно такая картина наблюдается, когда все операции работают в режиме `memory_order_relaxed`. К человеку, сидящему в боксе, можно обращаться и с другими просьбами, например: «запиши это число и скажи мне, что находится в конце списка» (`exchange`) или «запиши *это* число, если число в конце списка равно *тому*, в противном случае скажи мне, что я должен был бы предположить» (`compare_exchange_strong`), но общий принцип при этом не изменяется.

Применив эту метафору к программе в листинге 5.5, можно сказать, что `write_x_then_y` означает, что некто позвонил человеку в боксе `x`, попросил его записать `true`, а потом позвонил человеку в боксе `y` и попросил *его* записать `true`. Поток, выполняющий функцию `read_y_then_x`, раз за разом звонит человеку в боксе `y` и спрашивает значение, пока не услышит `true`, после чего звонит человеку в боксе `x` и спрашивает значение у него. Человек в боксе `x` не обязан сообщать вам какое-то конкретное значение из своего списка и с полным правом может назвать `false`.

Из-за этого с ослабленными атомарными операциями трудно иметь дело. Чтобы они были полезны для межпоточной синхронизации, их нужно сочетать с атомарными операциями, работающими в режиме с более строгой семантикой упорядочения. Я настоятельно рекомендую вообще избегать ослабленных атомарных операций, если без них можно обойтись, а, если никак нельзя, то использовать крайне осторожно. Учитывая, насколько интуитивно неочевидные результаты получились в листинге 5.5 при наличии всего двух потоков и двух переменных, нетрудно представить себе сложности, с которыми придется столкнуться, когда потоков и переменных станет больше.

Один из способов организовать дополнительную синхронизацию, не прибегая к последовательной согласованности, — воспользоваться упорядочением захват-освобождение.

Упорядочение захват-освобождение

Упорядочение захват-освобождение — шаг от ослабленного упорядочения в сторону большего порядка; полной упорядоченности операций еще нет, но какая-то синхронизация уже возможна. При такой модели атомарные операции загрузки являются операциями *захвата* (`memory_order_acquire`), атомарные операции сохранения — операциями *освобождения* (`memory_order_release`), а атомарные операции чтения-модификации-записи (например, `fetch_add()` или `exchange()`) — операциями *захвата*, *освобождения* или того и другого (`memory_order_acq_rel`). Синхронизация попарная — между потоком, выполнившим захват, и потоком, выполнившим освобождение. *Операция освобождения синхронизируется-с операцией захвата, которая читает записанное значение.* Это означает, что различные потоки *могут* видеть операции в разном порядке, но возможны все-таки не любые порядки. В следующем листинге показала программа из листинга 5.4, переработанная под семантику захвата-освобождения вместо семантики последовательной согласованности.

Листинг 5.7. Из семантики захвата-освобождения не вытекает полная упорядоченность

```
#include <atomic>
```

```
#include <thread>
#include <assert.h>

std::atomic<bool> x, y;
std::atomic<int> z;

void write_x() {
 x.store(true, std::memory_order_release);
}

void write_y() {
 y.store(true, std::memory_order_release);
}

void read_x_then_y() {
 while (!x.load(std::memory_order_acquire));
 if (y.load(std::memory_order_acquire)) ← (1)
  ++z;
}

void read_y_then_x() {
 while (!y.load(std::memory_order_acquire));
 if (x.load(std::memory_order_acquire)) ← (2)
  ++z;
}

int main() {
 x = false;
 y = false;
 z = 0;
 std::thread a(write_x);
 std::thread b(write_y);
 std::thread c(read_x_then_y);
 std::thread d(read_y_then_x);
 a.join();
 b.join();
 c.join();
 d.join();
 assert(z.load() != 0); ← (3)
}
```

В данном случае утверждение **(3)** *может* сработать (как и в случае ослабленного упорядочения), потому что обе операции загрузки — `x` **(2)** и `y` **(1)** могут прочитать значение `false`. Запись в переменные `x` и `y` производится из разных потоков, но упорядоченность между освобождением и захватом в одном потоке никак не отражается на операциях в других потоках.

На рис. 5.6 показаны отношения происходит-раньше, имеющие место в программе из листинга 5.7, а также возможный исход, когда два потока-читателя имеют разное представление о мире. Это возможно, потому что, как уже было сказано, не существует отношения происходит-раньше, которое вводило бы упорядочение.

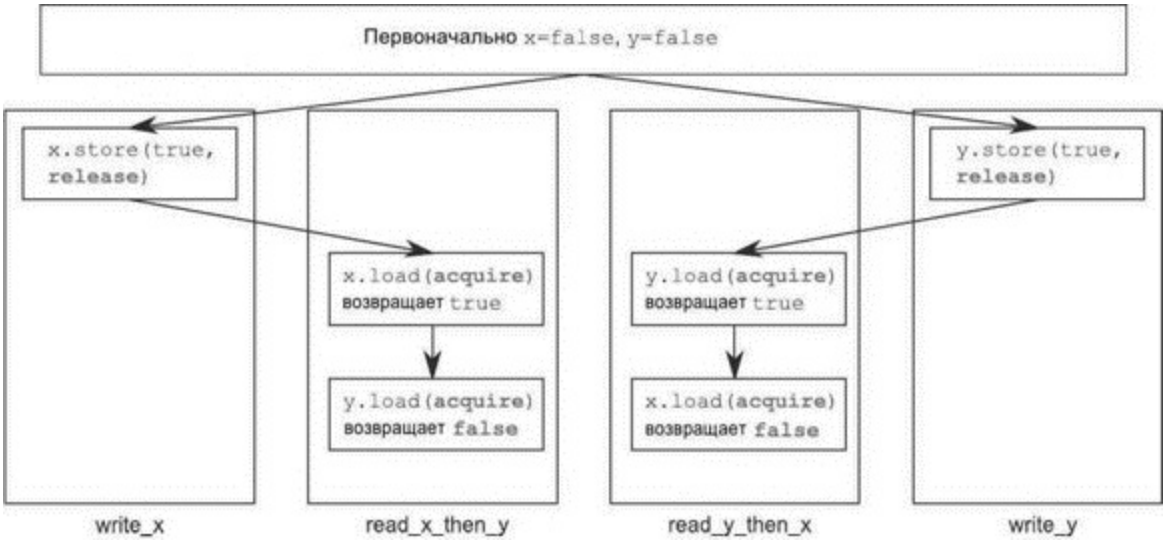

Рис. 5.6. Захват-освобождение и отношения происходит-раньше

Чтобы осознать преимущества упорядочения захват-освобождение, нужно рассмотреть две операции сохранения в одном потоке, как в листинге 5.5. Если при сохранении `y` задать семантику `memory_order_release`, а при загрузке `y` — семантику `memory_order_acquire`, как в листинге ниже, то операции над `x` станут упорядоченными.

Листинг 5.8. Операции с семантикой захвата-освобождения могут упорядочить ослабленные операции

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x, y;
std::atomic<int> z;

void write_x_then_y() {
 x.store(true,std::memory_order_relaxed);    ← (1)
 y.store(true,std::memory_order_release);    ← (2)
}

void read_y_then_x() {
 while (!y.load(std::memory_order_acquire));← (3)
 if (x.load(std::memory_order_relaxed))      ← (4)
  ++z;
}

int main() {
 x = false;
 y = false;
 z = 0;
 std::thread a(write_x_then_y);
 std::thread b(read_y_then_x);
 a.join();
 b.join();
 assert(z.load() != 0); ← (5)
}
```

В конечном итоге операция загрузки `y` **(3)** увидит значение `true`, записанное операцией

сохранения **(2)**. Поскольку сохранение производится в режиме `memory_order_release`, а загрузка — в режиме `memory_order_acquire`, то сохранение синхронизируется-с загрузкой. Сохранение `x` **(1)** происходит-раньше сохранения `y` **(2)**, потому что обе операции выполняются в одном потоке. Поскольку сохранение `y` синхронизируется-с загрузкой `y`, то сохранение `x` также происходит-раньше загрузки `y`, и, следовательно, происходит-раньше загрузки `x` **(4)**. Таким образом, операция загрузки `x` *должна* прочитать `true`, и, значит, утверждение **(5)** *не может* сработать. Если бы загрузка `y` не повторялась в цикле `while`, то высказанное утверждение могло бы оказаться неверным; операция загрузки `y` могла бы прочитать `false`, и тогда не было бы никаких ограничений на значение, прочитанное из `x`. Для обеспечения синхронизации операции захвата и освобождения должны употребляться парами. Значение, сохраненное операций восстановления, должно быть видно операции захвата, иначе ни та, ни другая не возымеют эффекта. Если бы сохранение в предложении **(2)** или загрузка в предложении **(3)** выполнялись в ослабленной операции, то обращения к `x` не были бы упорядочены, и, значит, нельзя было бы гарантировать, что операция загрузки в предложении **(4)** прочитает значение `true`, поэтому утверждение `assert` могло бы сработать.

К упорядочению захват-освобождение можно применить метафору человека с блокнотом в боксе, если ее немного дополнить. Во-первых, допустим, что каждое сохранение является частью некоторого пакета обновлений, поэтому, обращаясь к человеку с просьбой записать число, вы заодно сообщается ему идентификатор пакета, например: «Запиши 99 как часть пакета 423». Если речь идет о последнем сохранении в пакете, то мы сообщаем об этом: «Запиши 147, отметив, что это последнее сохранение в пакете 423». Человек в боксе честно записывает эту информацию вместе с указанным вами значением. Так моделируется операция сохранения с освобождением. Когда вы в следующий раз попросите записать значение, помер пакета нужно будет увеличить: «Запиши 41 как часть пакета 424».

Теперь, когда вы просите сообщить значение, у вас есть выбор: узнать только значение (это аналог ослабленной загрузки) или значение и сведения о том, является ли оно последним в пакете (это аналог загрузки с захватом). Если информация о пакете запрашивается, по значение не последнее в пакете, то человек ответит: «Число равно 987, и это 'обычное' значение»; если же значение последнее, то ответ прозвучит так: «Число 987, последнее в пакете 956 от Анны». Тут-то и проявляется семантика захвата-освобождения: если, запрашивая значение, вы сообщите человеку номера всех пакетов, о которых знаете, то он найдёт в своем списке последнее значение из всех известных вам пакетов и назовёт либо его, либо какое-нибудь следующее за ним в списке.

Как эта метафора моделирует семантику захвата-освобождения? Взгляните на наш пример — и поймете. В самом начале поток `a` вызывает функцию `write_x_then_y` и говорит человеку в боксе `x`: «Запиши `true`, как часть пакета 1 от потока `a`». Затем поток `a` говорит человеку в боксе `y`: «Запиши `true`, как последнюю операцию записи в пакете 1 от потока `a`». Тем временем поток `b` выполняет функцию `read_y_then_x`. Он раз за разом просит человека в боксе `y` сообщить значение вместе с информацией о пакете, пока не услышит в ответ «`true`». Возможно, спросить придется много раз, но в конце концов человек обязательно ответит «`true`». Однако человек в боксе `y` говорит не просто «`true`», а еще добавляет: «Это последняя операция записи в пакете 1 от потока `a`».

Далее поток `b` просит человека в боксе `x` назвать значение, но на это раз говорит: «Сообщи мне значение и, кстати, я знаю о пакете 1 от потока `a`». Человек в боксе x ищет в

своем списке последнее упоминание о пакете 1 от потока `a`. Он находит единственное значение `true`, которое стоит последним в списке, поэтому он *обязан* сообщить именно это значение, иначе нарушит правила игры.

Вспомнив определение отношения *межпоточно происходит раньше* в разделе 5.3.2, вы обнаружите, что одно из его существенных свойств — транзитивность: *если A межпоточно происходит-раньше B* и *B межпоточно происходит-раньше C*, то *A межпоточно происходит-раньше C*. Это означает, что упорядочение захват-освобождение можно использовать для синхронизации данных между несколькими потоками, даже если «промежуточные» потоки на самом деле не обращались к данным.

Транзитивная синхронизация с помощью упорядочения захват-освобождение

Для рассуждений о транзитивном упорядочении нужны по меньшей мере три потока. Первый модифицирует какие-то разделяемые переменные и выполняет операцию сохранения с освобождением в одну из них. Второй читает переменную, записанную операцией сохранения с освобождением, с помощью операции загрузки с захватом и выполняет сохранение с освобождением во вторую разделяемую переменную. Наконец, третий поток выполняет операцию загрузки с захватом для второй разделяемой переменной. При условии, что операции загрузки с захватом видят значения, записанные операциями сохранения с освобождением, и тем самым поддерживают отношения синхронизируется-с, третий поток может прочитать значения других переменных, сохраненные первым потоком, даже если промежуточный поток к ним не обращался. Этот сценарий иллюстрируется в следующем листинге.

Листинг 5.9. Транзитивная синхронизация с помощью упорядочения захват-освобождение

```
std::atomic<int> data[5];
std::atomic<bool> sync1(false), sync2(false);

void thread_1() {
 data[0].store(42, std::memory_order_relaxed);
 data[1].store(97, std::memory_order_relaxed);
 data[2].store(17, std::memory_order_relaxed);
 data[3].store(-141, std::memory_order_relaxed);
 data[4].store(2003, std::memory_order_relaxed);←┐ Установить
 sync1.store(true, std::memory_order_release);   (1)sync1
}

void thread_2()                                    (2)Цикл до
{                                                   | установки
 while (!sync1.load(std::memory_order_acquire));←┘ sync1
 sync2.store(true, std::memory_order_release); ←┐ Установить
}                                                  (3) sync2

void thread_3()                                    (4)Цикл до
```

```
{                                                          | установки
 while (!sync2.load(std::memory_order_acquire));←┘ sync2
 assert(data[0].load(std::memory_order_relaxed) == 42);
 assert(data[1].load(std::memory_order_relaxed) == 97);
 assert(data[2].load(std::memory_order_relaxed) == 17);
 assert(data[3].load(std::memory_order_relaxed) == -141);
 assert(data[4].load(std::memory_order_relaxed) == 2003);
}
```

Хотя поток `thread_2` обращается только к переменным `sync1` **(2)** и `sync2` **(3)**, этого достаточно для синхронизации между `thread_1` и `thread_3` и, стало быть, гарантии несрабатывания утверждений `assert`. Прежде всего, операции сохранения в элементы массива `data` в потоке `thread_1` происходят-раньше сохранения `sync1` **(1)**, потому что они связаны отношением расположено-перед в одном потоке. Поскольку операция загрузки `sync1` **(2)** находится внутри цикла `while`, она в конце концов увидит значение, сохраненное в `thread_1` и, значит, образует вторую половину пары освобождение-захват. Поэтому сохранение `sync1` происходит-раньше последней загрузки `sync1` в цикле `while`. Эта операция загрузки расположена-перед (и, значит, происходит-раньше) операцией сохранения `sync2` **(3)**, которая образует пару освобождение-захват вместе с последней операцией загрузки в цикле `while` в потоке `thread_3` **(4)**. Таким образом, сохранение `sync2` **(3)** происходит-раньше загрузки **(4)**, которая происходит-раньше загрузок `data`. В силу транзитивности отношения происходит-раньше всю эту цепочку можно соединить: операции сохранения `data` происходят-раньше операций сохранения `sync1` **(1)**, которые происходят-раньше загрузки `sync1` **(2)**, которая происходит-раньше сохранения `sync2` **(3)**, которая происходит-раньше загрузки `sync2` **(4)**, которая происходит-раньше загрузок `data`. Следовательно, операции сохранения `data` в потоке `thread_1` происходят-раньше операций загрузки `data` в потоке `thread_3`, и утверждения `assert` сработать не могут.

В этом случае можно было бы объединить `sync1` и `sync2` в одну переменную, воспользовавшись операцией чтения-модификации-записи с семантикой `memory_order_acq_rel` в потоке `thread_2`. Один из вариантов — использовать функцию `compare_exchange_strong()`, гарантирующую, что значение будет обновлено только после того, как поток `thread_2` увидит результат сохранения в потоке `thread_1`:

```
std::atomic<int> sync(0);

void thread_1() {
 // ...
 sync.store(1, std::memory_order_release);
}

void thread_2() {
 int expected = 1;
 while (!sync.compare_exchange_strong(expected, 2,
         std::memory_order_acq_rel))
  expected = 1;
}

void thread_3() {
 while(sync.load(std::memory_order_acquire) < 2);
 // ...
}
```

При использовании операций чтения-модификации-записи важно выбрать нужную семантику. В данном случае нам нужна одновременно семантика захвата и освобождения, поэтому подойдет `memory_order_acq_rel`, но можно было бы применить другие виды упорядочения. Операция `fetch_sub` с семантикой `memory_order_acquire` не синхронизируется ни с чем, хотя и сохраняет значение, потому что это не операция освобождения. Аналогично сохранение не может синхронизироваться-с операцией `fetch_or` с семантикой `memory_order_release`, потому что часть «чтение» `fetch_or` не является операцией захвата. Операции чтения-модификации-записи с семантикой `memory_order_acq_rel` ведут себя как операции захвата и освобождения одновременно, поэтому предшествующее сохранение может синхронизироваться-с такой операцией и с последующей загрузкой, как и обстоит дело в примере выше.

Если вы сочетаете операции захвата-освобождения с последовательно согласованными операциями, то последовательно согласованные операции загрузки ведут себя, как загрузки с семантикой захвата, а последовательно согласованные операции сохранения — как сохранения с семантикой освобождения. Последовательно согласованные операции чтения-модификации-записи ведут себя как операции, наделенные одновременно семантикой захвата и освобождения. Ослабленные операции так и остаются ослабленными, но связаны дополнительными отношениями синхронизируется-с и последующими отношениями происходит-раньше, наличие которых обусловлено семантикой захвата-освобождения.

Несмотря на интуитивно неочевидные результаты, всякий, кто использовал блокировки, вынужденно имел дело с вопросами упорядочения: блокировка мьютекса — это операция захвата, а его разблокировка — операция освобождения. Работая с мьютексами, вы на опыте узнаете, что при чтении значения необходимо захватывать тот же мьютекс, который захватывался при его записи. Точно так же обстоит дело и здесь — для обеспечения упорядочения операции захвата и освобождения должны применяться к одной и той же переменной. Если данные защищены мьютексом, то взаимно исключающая природа блокировки означает, что результат неотличим от того, который получился бы, если бы операции блокировки и разблокировки были последовательно согласованы. Аналогично, если для построения простой блокировки к атомарным переменным применяется упорядочение захват-освобождение, то с точки зрения программы, *использующей* такую блокировку, поведение кажется последовательно согласованным, хотя внутренние операции таковыми не являются.

Если для выполняемых в вашей программе атомарных операций не нужна строгость последовательно согласованного упорядочения, то попарная синхронизация с помощью упорядочения захват-освобождение может обеспечить синхронизацию со значительно меньшими издержками, чем необходимое для последовательно согласованных операций глобальное упорядочение. Ценой компромисса являются мысленные усилия, необходимые для того, чтобы удостовериться в том, что упорядочение работает правильно, а интуитивно неочевидное поведение нескольких потоков не вызывает проблем.

Зависимости по данным, упорядочение захват-освобождение и семантика ***memory_order_consume***

Во введении к этому разделу я говорил, что семантика `memory_order_consume` является

частью модели упорядочения захват-освобождение, но из предшествующего описания она полностью выпала. Дело в том, что семантика `memory_order_consume` особая: она связана с зависимостями по данным и позволяет учесть соответствующие нюансы в отношении *межпоточно происходит-раньше*, о котором шла речь в разделе 5.3.2.

С зависимостями по данным связаны два новых отношения: *предшествует-по-зависимости* (dependency-ordered-before) и *переносит-зависимость-в* (carries-a-dependency-to). Как и отношение расположено-перед, отношение переносит-зависимость-в применяется строго внутри одного потока и моделирует зависимость по данным между операциями — если результат операции A используется в качестве операнда операции B, то A переносит-зависимость-в B. Если результатом операции A является значение скалярного типа, например `int`, то отношение применяется и тогда, когда результат A сохраняется в переменной, которая затем используется в качестве операнда B. Эта операция также транзитивна, то есть если A переносит-зависимость-в B и B переносит-зависимость-в C, то A переносит-зависимость-в C.

С другой стороны, отношение предшествует-по-зависимости может применяться к разным потокам. Оно вводится с помощью атомарных операций загрузки, помеченных признаком `memory_order_consume`. Это частный случай семантики `memory_order_acquire`, в котором синхронизированные данные ограничиваются прямыми зависимостями; операция сохранения A, помеченная признаком `memory_order_release`, `memory_order_acq_rel` или `memory_order_seq_cst`, предшествует-по-зависимости операции загрузки B, помеченной признаком `memory_order_consume`, если потребитель читает сохраненное значение. Это противоположность отношению синхронизируется-с, которое образуется, если операция загрузки помечена признаком `memory_order_acquire`. Если такая операция B затем переносит-зависимость-в некоторую операцию C, то A также предшествует-по-зависимости C.

Это не дало бы ничего полезного для целей синхронизации, если бы не было связано с отношением межпоточно происходит-раньше. Однако же справедливо следующее утверждение: если A предшествует-по-зависимости B, то A межпоточно происходит-раньше B.

Одно из важных применений такого упорядочения доступа к памяти связано с атомарной операцией загрузки указателя на данные. Пометив операцию загрузки признаком `memory_order_consume`, а предшествующую ей операцию сохранения — признаком `memory_order_release`, можно гарантировать, что данные, адресуемые указателем, правильно синхронизированы, даже не накладывая никаких требований к синхронизации с другими независимыми данными. Этот сценарий иллюстрируется в следующем листинге.

Листинг 5.10. Использование `std::memory_order_consume` для синхронизации данных

```
struct X {
 int i;
 std::string s;
};

std::atomic<X*> p;
std::atomic<int> a;

void create_x() {
 X* x = new X;
```

```
 x->i = 42;
 x->s = "hello";
 a.store(99, std::memory_order_relaxed);← (1)
 p.store(x, std::memory_order_release); ← (2)
}

void use_x() {
 X* x;
 while (!(x = p.load(std::memory_order_consume)))← (3)
 std::this_thread::sleep(std::chrono::microseconds(1));
 assert(x->i == 42);                                ← (4)
 assert(x->s =="hello");                            ← (5)
 assert(a.load(std::memory_order_relaxed) == 99);← (6)
}

int main() {
 std::thread t1(create_x);
 std::thread t2(use_x);
 t1.join();
 t2.join();
}
```

Хотя сохранение `a` **(1)** расположено перед сохранением `p` **(2)** и сохранение `p` помечено признаком `memory_order_release`, но загрузка `p` **(3)** помечена признаком `memory_order_consume`. Это означает, что сохранение `p` происходит-раньше только тех выражений, которые зависят от значения, загруженного из `p`. Поэтому утверждения о членах-данных структуры `x` **(4)**, **(5)** гарантированно не сработают, так как загрузка `p` переносит-зависимость-в эти выражения посредством переменной `x`. С другой стороны, утверждение о значении `a` **(6)** может как сработать, так и не сработать; эта операция не зависит от значения, загруженного из `p`, поэтому нет никаких гарантий о прочитанном значении. Это ясно следует из того, что она помечена признаком `memory_order_relaxed`.

Иногда нам не нужны издержки, которыми сопровождается перенос зависимости. Мы хотим, чтобы компилятор мог кэшировать значения в регистрах и изменять порядок операций во имя оптимизации кода, а не волновался по поводу зависимостей. В таких случаях можно воспользоваться шаблоном функции `std::kill_dependency()` для явного разрыва цепочки зависимостей. Эта функция просто копирует переданный ей аргумент в возвращаемое значение, но попутно разрывает цепочку зависимостей. Например, если имеется глобальный массив с доступом только для чтения, и вы используете семантику `std::memory_order_consume` при чтении какого-то элемента этого массива из другого потока, то с помощью `std::kill_dependency()` можно сообщить компилятору, что ему необязательно заново считывать содержимое элемента массива (см. пример ниже).

```
int global_data[] = { ... };
std::atomic<int> index;

void f() {
 int i = index.load(std::memory_order_consume);
 do_something_with(global_data[std::kill_dependency(i)]);
}
```

Разумеется, в таком простом случае вы вряд ли вообще будете пользоваться семантикой `std::memory_order_consume`, но в аналогичной ситуации функцией

`std::kill_dependency()` можно воспользоваться и в более сложной программе. Только не забывайте, что это оптимизация, поэтому прибегать к ней следует с осторожностью и только тогда, когда профилирование ясно продемонстрировало необходимость.

Теперь, рассмотрев основы упорядочения доступа к памяти, мы можем перейти к более сложным аспектам отношения синхронизируется-с, которые проявляются в форме *последовательностей освобождений* (release sequences).

5.3.4. Последовательности освобождений и отношение синхронизируется-с

В разделе 5.3.1 я упоминал, что можно получить отношение синхронизируется-с между операцией сохранения атомарной переменной и операцией загрузки той же атомарной переменной в другом потоке, даже если между ними выполняется последовательность операций чтения-модификации-записи, — при условии, что все операции помечены надлежащим признаками. Теперь, когда мы знаем обо всех возможных «признаках» упорядочения, я могу подробнее осветить этот вопрос. Если операция сохранения помечена одним из признаков `memory_order_release`, `memory_order_acq_rel` или `memory_order_seq_cst`, а операция загрузки — одним из признаков `memory_order_consume`, `memory_order_acquire` или `memory_order_seq_cst`, и каждая операция в цепочке загружает значение, записанное предыдущей операцией, то такая цепочка операций составляет *последовательность освобождений*, и первая в ней операция сохранения синхронизируется-с (в случае `memory_order_acquire` или `memory_order_seq_cst`) или предшествует-по-зависимости (в случае `memory_order_consume`) последней операции загрузки. Любая атомарная операция чтения-модификации-записи в цепочке может быть помечена *произвольным* признаком упорядочения (даже `memory_order_relaxed`).

Чтобы попять, что это означает и почему так важно, рассмотрим значение типа `atomic<int>`, которое используется как счетчик `count` элементов в разделяемой очереди (см. листинг ниже).

Листинг 5.11. Чтение из очереди с применением атомарных операций

```
#include <atomic>
#include <thread>

std::vector<int> queue_data; std::atomic<int> count;

void populate_queue() {
 unsigned const number_of_items = 20;
 queue_data.clear();
 for (unsigned i = 0; i < number_of_items; ++i) {
  queue_data.push_back(i);
 } ← (1) Начальное сохранение
 count.store(number_of_items, std::memory_order_release);
}

void consume_queue_items() {
 while (true) { ← (2) Операция ЧМЗ
 int item_index;
 if (
  (item_index =
```

```
    count.fetch_sub(1, std::memory_order_acquire)) <= 0) {
    wait_for_more_items();←┐ Ждем дополнительных
    continue;              (3) элементов
  }
  process(queue_data[item_index-1]);←┐ Чтение из queue_data
}                                    (4) безопасно

int main() {
  std::thread a(populate_queue);
  std::thread b(consume_queue_items);
  std::thread c(consume_queue_items);
  a.join();
  b.join();
  c.join();
}
```

Можно, например, написать программу так, что поток, производящий данные, сохраняет их в разделяемом буфере, а затем вызывает функцию `count.store(number_of_items, memory_order_release)` **(1)**, чтобы другие потоки узнали о готовности данных. Потоки-потребители, читающие данные из очереди, могли бы затем вызвать `count.fetch_sub(1, memory_order_acquire)` **(2)**, чтобы проверить, есть ли элементы в очереди перед тем, как фактически читать из разделяемого буфера **(4)**. Если счетчик `count` стал равен 0, то больше элементов нет, и поток должен ждать **(3)**.

Если поток-потребитель всего один, то всё хорошо; `fetch_sub()` — это операция чтения с семантикой `memory_order_acquire`, а операция сохранения была помечена признаком `memory_order_release`, поэтому сохранение синхронизируется-с загрузкой, и поток может читать данные из буфера. Но если читают два потока, то второй вызов `fetch_sub()` увидит значение, записанное при первом вызове, а не то, которое было записано операцией `store`. Без правила о последовательности освобождений между вторым и первым потоком не было бы отношения происходит-раньше, поэтому было бы небезопасно читать из разделяемого буфера, если только и для первого вызова `fetch_sub()` тоже не задана семантика `memory_order_release`; однако, задав ее, мы ввели бы излишнюю синхронизацию между двумя потоками-потребителями. Без правила о последовательности освобождений или задания семантики `memory_order_release` для всех операций `fetch_sub` не было бы никакого механизма, гарантирующего, что операции сохранения в queue_data видны второму потребителю, следовательно, мы имели бы гонку за данными. К счастью, первый вызов `fetch_sub()` на самом деле *участвует* в последовательности освобождений, и вызов `store()` синхронизируется-с вторым вызовом `fetch_sub()`. Однако отношения синхронизируется-с между двумя потоками-потребителями все еще не существует. Это изображено на рис. 5.7, где пунктирные линии показывают последовательность освобождений, а сплошные — отношения происходит-раньше.

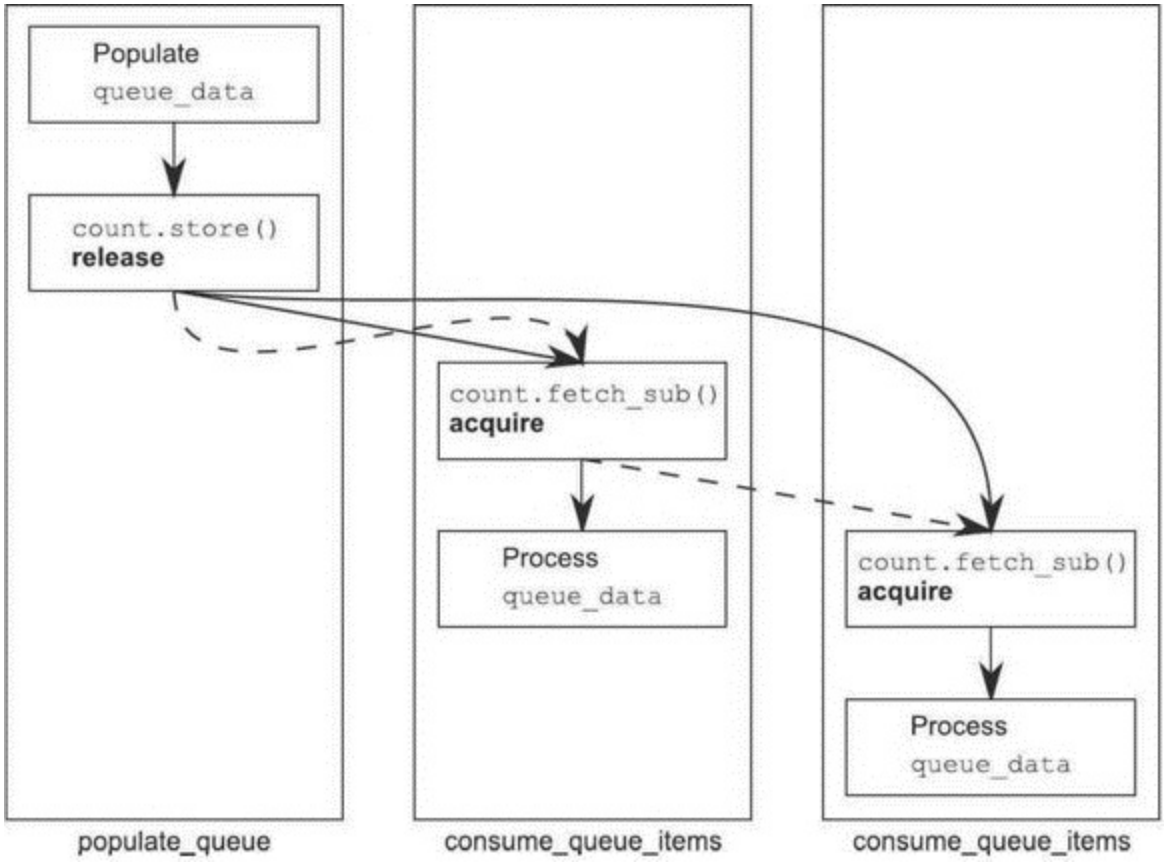

Рис. 5.7. Последовательность освобождений для операций с очередью из листинга 5.11

В цепочке может быть сколько угодно звеньев, но при условии, что все они являются операциями чтения-модификации-записи, как `fetch_sub()`, операция `store()` синхронизируется-с каждым звеном, помеченным признаком `memory_order_acquire`. В данном примере все звенья одинаковы и являются операциями захвата, но это вполне могли бы быть разные операции с разной семантикой упорядочения доступа к памяти.

Хотя большая часть отношений синхронизации проистекает из семантики упорядочения доступа к памяти, применённой к операциям над атомарными переменными, существует возможность задать дополнительные ограничения на упорядочение с помощью *барьеров* (fence).

5.3.5. Барьеры

Библиотека атомарных операций была бы неполна без набора барьеров. Это операции, которые налагают ограничения на порядок доступа к памяти без модификации данных. Обычно они используются в сочетании с атомарными операциями, помеченными признаком `memory_order_relaxed`. Барьеры — это глобальные операции, они влияют на упорядочение других атомарных операций в том потоке, где устанавливается барьер. Своим названием барьеры обязаны тому, что устанавливают в коде границу, которую некоторые операции не могут пересечь. В разделе 5.3.3 мы говорили, что компилятор или сам процессор вправе изменять порядок ослабленных операций над различными переменными. Барьеры ограничивают эту свободу и вводят отношения происходит-раньше и синхронизируется-с, которых до этого не было.

В следующем листинге демонстрируется добавление барьера между двумя атомарными операциями в каждом потоке из листинга 5.5.

Листинг 5.12. Ослабленные операции можно упорядочить с помощью барьеров

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x, y;
std::atomic<int> z;

void write_x_then_y() {
 x.store(true, std::memory_order_relaxed);                    ← (1)
 std::atomic_thread_fence(std::memory_order_release);         ← (2)
 y.store(true, std::memory_order_relaxed);                    ← (3)
}

void read_y_then_x() {
 while (!y.load(std::memory_order_relaxed));                  ← (4)
 std::atomic_thread_fence(std::memory_order_acquire);         ← (5)
 if (x.load(std::memory_order_relaxed))                       ← (6)
  ++z;
}

int main() {
 x = false;
 y = false;
 z = 0;
 std::thread a(write_x_then_y);
 std::thread b(read_y_then_x);
 a.join();
 b.join();
 assert(z.load() != 0); ← (7)
}
```

Барьер освобождения **(2)** синхронизируется-с барьером захвата **(5)**, потому что операция загрузки `y` в точке **(4)** читает значение, сохраненное в точке **(3)**. Это означает, что сохранение `x` **(1)** происходит-раньше загрузки `x` **(6)**, поэтому прочитанное значение должно быть равно `true`, и утверждение **(7)** не сработает. Здесь мы наблюдаем разительное отличие от исходного случая без барьеров, когда сохранение и загрузка `x` не были упорядочены, и утверждение могло сработать. Отметим, что оба барьера обязательны: чтобы получить отношение синхронизируется-с необходимо освобождение в одном потоке и захват в другом.

В данном случае барьер освобождения **(2)** оказывает такой же эффект, как если бы операция сохранения `y` **(3)** была помечена признаком `memory_order_release`, а не `memory_order_relaxed`. Аналогично эффект от барьера захвата **(5)** такой же, как если бы операция загрузки `y` **(4)** была помечена признаком `memory_order_acquire`. Это общее свойство всех барьеров: если операция захвата видит результат сохранения, имевшего место после барьера освобождения, то барьер синхронизируется-с этой операцией захвата. Если же операция загрузки, имевшая место до барьера захвата, видит результат операции освобождения, то операция освобождения синхронизируется-с барьером захвата. Разумеется, можно поставить барьеры по обе стороны, как в примере выше, и в таком случае если загрузка, которая имела место до барьера захвата, видит значение, записанное операцией

сохранения, имевшей место после барьера освобождения, то барьер освобождения синхронизируется-с барьером захвата.

Хотя барьерная синхронизация зависит от значений, прочитанных или записанных операциями до и после барьеров, важно отметить, что точкой синхронизации является сам барьер. Если взять функцию `write_x_then_y` из листинга 5.12 и перенести запись в `x` после барьера, как показано ниже, то уже не гарантируется, что условие в утверждение будет истинным, несмотря на то что запись в x предшествует записи в `y`:

```
void write_x_then_y() {
 std::atomic_thread_fence(std::memory_order_release);
 x.store(true, std::memory_order_relaxed);
 y.store(true, std::memory_order_relaxed);
}
```

Эти две операции больше не разделены барьером и потому не упорядочены. Барьер обеспечивает упорядочение только тогда, когда находится *между* сохранением `x` и сохранением `y`. Конечно, наличие или отсутствие барьера не влияет на упорядочения, обусловленные отношениями происходит-раньше, которые существуют благодаря другим атомарным операциям.

Данный пример, как и почти все остальные в этой главе, целиком построен на переменных атомарных типов. Однако реальная польза от применения атомарных операций для навязывания упорядочения проистекает из того, что они могут упорядочивать неатомарные операции и тем самым предотвращать неопределенное поведение из-за гонок за данными, как мы видели в листинге 5.2.

5.3.6. Упорядочение неатомарных операций с помощью атомарных

Если заменить тип переменной `x` в листинге 5.12 обычным неатомарным типом `bool` (как в листинге ниже), то гарантируется точно такое же поведение, как и раньше.

Листинг 5.13. Принудительное упорядочение неатомарных операций

```
#include <atomic>
#include <thread>
#include <assert.h>

bool x = false;     ←┐ Теперь x — простая
std::atomic<bool> y;| неатомарная
std::atomic<int> z; | переменная
void write_x_then_y() { (1) Сохранение x
 x = true;              ←┘ перед барьером
 std::atomic_thread_fence(std::memory_order_release);
 y.store(true, std::memory_order_relaxed);←┐ Сохранение y
}                                          (2) после барьера

void read_y_then_x()                        (3) Ждем, пока не
{                                             | увидим значение,
 while (!y.load(std::memory_order_relaxed));←┘ записанное в 2
 std::atomic_thread_fence(std::memory_order_acquire);
```

```
  if (x) ←┐ Здесь будет прочитано
    ++z;     (4) значение, записанное в 1
}

int main() {
  x = false;
  y = false;
  z = 0;
  std::thread a(write_x_then_y);
  std::thread b(read_y_then_x);
  a.join();
  b.join();                    (5) Это утверждение
  assert(z.load() != 0);←┘ не сработает
}
```

Барьеры по-прежнему обеспечивают упорядочение сохранения `x` **(1)** и `y` **(2)** и загрузки `y` **(3)** и `x` **(4)**, и, как и раньше, существует отношение происходит-раньше между сохранением `x` и загрузкой `x`, поэтому утверждение **(5)** не сработает. Сохранение `y` **(2)** и загрузка `y` **(3)** тем не менее должны быть атомарными, иначе возникла бы гонка за `y`, но барьеры упорядочивают операции над `x` после того, как поток-читатель увидел сохраненное значение `y`. Такое принудительное упорядочение означает, что гонки за `x` нет, хотя ее значение модифицируется в одном потоке, а читается в другом.

Но не только с помощью барьеров можно упорядочить неатомарные операции. Эффект упорядочения мы наблюдали также в листинге 5.10, где пара `memory_order_release` / `memory_order_consume` упорядочивала неатомарные операции доступа к динамически выделенному объекту. Многие примеры из этой главы можно было бы переписать, заменив некоторые операции с семантикой `memory_order_relaxed` простыми неатомарными операциями.

Упорядочение неатомарных операций с помощью атомарных — это та область, где особую важность приобретает аспект расположено-перед отношения происходит-раньше. Если неатомарная операция расположено-перед атомарной, и эта атомарная операция происходит-раньше какой-либо операции в другом потоке, то и неатомарная операция также происходит-раньше этой операции в другом потоке. Именно из этого вытекает упорядочение операций над `x` в листинге 5.13, и именно поэтому работает пример из листинга 5.2. Этот факт также лежит в основе таких высокоуровневых средств синхронизации в стандартной библиотеке C++, как мьютексы и условные переменные. Чтобы понять, как это работает, рассмотрим простой мьютекс-спинлок из листинга 5.1.

В функции `lock()` выполняется цикл по `flag.test_and_set()` с упорядочением `std::memory_order_acquire`, а функция `unlock()` вызывает операцию `flag.clear()` с признаком упорядочения `std::memory_order_release`. В момент, когда первый поток вызывает `lock()`, флаг еще сброшен, поэтому первое обращение к `test_and_set()` установит его и вернет `false`. Это означает, что поток завладел блокировкой, и цикл завершается. Теперь этот поток вправе модифицировать любые данные, защищенные мьютексом. Всякий другой поток, который вызовет `lock()` в этот момент, обнаружит, что флаг уже поднят, и потому будет заблокирован в цикле `test_and_set()`. Когда поток, владеющий блокировкой, закончит модифицировать защищенные данные, он вызовет функцию `unlock()`, которая вызовет `flag.clear()` с семантикой `std::memory_order_release`.

Это приводит к синхронизации-с (см. раздел 5.3.1) последующим обращением к

`flag.test_and_set()` из функции `lock()` в другом потоке, потому что в этом обращении задана семантика `std::memory_order_acquire`. Так как модификация защищенных данных обязательно расположена-перед вызовом `unlock()`, то эта модификация происходит-раньше вызова `unlock()` и, следовательно, происходит-раньше последующего обращения к `lock()` из другого потока (благодаря наличию отношения синхронизируется-с между `unlock()` и `lock()`) и происходит-раньше любой операции доступа к данным из второго потока после того, как он захватит блокировку.

В других реализациях мьютексов используются иные внутренние операции, но принцип остается неизменным: `lock()` — это операция захвата над некоторой внутренней ячейкой памяти, а `unlock()` — операция освобождения над той же ячейкой памяти.

5.4. Резюме

В этой главе мы рассмотрели низкоуровневые детали модели памяти в C++11 и атомарные операции, лежащие в основе синхронизации потоков. Были также рассмотрены простые атомарные типы, предоставляемые специализациями шаблона класса `std::atomic<>`, и обобщенный интерфейс в виде основного шаблона `std::atomic<>`, операции над этими типами и непростые детали, связанные с различными вариантами упорядочения доступа к памяти.

Мы также рассмотрели барьеры и их использование в сочетании с операциями над атомарными типами для обеспечения принудительного упорядочения. Наконец, мы вернулись к началу и показали, как можно использовать атомарные операции для упорядочения неатомарных операций, выполняемых в разных потоках.

В следующей главе мы увидим, как высокоуровневые средства синхронизации вкупе с атомарными операциями применяются для проектирования эффективных контейнеров, допускающих параллельный доступ, а также напишем алгоритмы для параллельной обработки данных.

Глава 6
Проектирование параллельных структур данных с блокировками

В этой главе:

■ Что понимается под проектированием структур данных, рассчитанных на параллельный доступ?

■ Рекомендации по проектированию таких структур.

■ Примеры реализации параллельных структур данных.

В предыдущей главе мы рассмотрели низкоуровневые детали атомарных операций и модели памяти. В этой главе мы на время отойдем от низкоуровневых деталей (чтобы вернуться к ним в главе 7) и поразмыслим о структурах данных.

От выбора структуры данных может в значительной степени зависеть всё решение поставленной задачи, и параллельное программирование — не исключение. Если к структуре данных планируется обращаться из разных потоков, то возможно два варианта: либо структура вообще неизменяемая, и тогда никакой синхронизации не требуется, либо программа спроектирована так, что любые изменения корректно синхронизированы. Одна из возможностей — завести отдельный мьютекс и пользоваться для защиты данных внешней по отношению к структуре блокировкой, применяя технику, рассмотренную в главах 3 и 4. Другая — спроектировать саму структуру данных, так чтобы к ней был возможен параллельный доступ.

При проектировании структуры данных, допускающей параллельный доступ, мы можем использовать основные строительные блоки многопоточных приложений, описанные в предыдущих главах, например мьютексы и условные переменные. На самом деле, вы уже видели примеры структур, в которых сочетание этих блоков гарантирует безопасный доступ из нескольких потоков.

Мы начнем эту главу с нескольких общих рекомендаций по проектированию параллельных структур данных. Затем мы еще раз вернемся к использованию блокировок и условных переменных в простых структурах, после чего перейдём к более сложным. В главе 7 я покажу, как с помощью атомарных операций можно строить структуры данных без блокировок.

Но довольно предисловий — посмотрим, что входит в проектирование структуры данных для параллельного программирования.

6.1. Что понимается под проектированием структур данных, рассчитанных на параллельный доступ?

На простейшем уровне это означает, что нужно спроектировать структуру данных, к которой смогут одновременно обращаться несколько потоков для выполнения одних и тех же или разных операций, причём каждый поток должен видеть согласованное состояние структуры. Данные не должны теряться или искажаться, все инварианты должны быть соблюдены, и никаких проблематичных состояний гонки не должно быть. Такая структура данных называется *потокобезопасной*. В общем случае структура данных безопасна только относительно определенных типов параллельного доступа. Не исключено, что несколько потоков могут одновременно выполнять какую-то одну операцию над структурой, а для выполнения другой необходим монопольный доступ. Наоборот, бывает, что несколько потоков могут одновременно и безопасно выполнять *различные* операции, но при выполнении *одной и той же* операции в разных потоках возникают проблемы.

Но проектирование с учетом параллелизма этим не исчерпывается: задача заключается в том, чтобы предоставить *возможность распараллеливания* потокам, обращающимся к структуре данных. По природе своей, мьютекс означает *взаимное исключение*: в каждый момент времени только один поток может захватить мьютекс. Следовательно, мьютекс защищает структуру данных, явным образом *предотвращая* истинно параллельный доступ к ней.

Это называется *сериализацией*: потоки обращаются к защищенным мьютексом данным по очереди, то есть последовательно, а не параллельно. Чтобы обеспечить истинно параллельный доступ, нужно тщательно продумывать внутреннее устройство структуры данных. Одни структуры данных оставляют больший простор для распараллеливания, чем другие, но идея остается неизменной: чем меньше защищаемая область, тем меньше операций приходится сериализовывать и тем больше потенциал для распараллеливания.

Прежде чем знакомиться с конкретными структурами данных, приведём несколько простых рекомендаций, полезных при проектировании с учетом параллелизма.

6.1.1. Рекомендации по проектированию структур данных для параллельного доступа

Как я уже отмечал, при проектировании структур данных для параллельного доступа нужно учитывать два аспекта: обеспечить *безопасность* доступа и *разрешить* истинно параллельный доступ. Как сделать структуру потокобезопасной, я уже рассказывал в главе 3.

• Гарантировать, что ни один поток не может увидеть состояние, в котором инварианты структуры данных нарушены действиями со стороны других потоков.

• Позаботиться о предотвращении состояний гонки, внутренне присущих структуре данных, предоставив такие функции, которые выполняли бы операции целиком, а не частями.

• Обращать внимание на том, как ведет себя структура данных при наличии исключений, — не допускать нарушения инвариантов и в этом случае.

• Минимизировать шансы возникновения взаимоблокировки, ограничивая область действия блокировок и избегая но возможности вложенных блокировок.

Прежде чем задумываться об этих деталях, важно решить, какие ограничения вы собираетесь наложить на использование структуры данных: если некоторый поток обращается к структуре с помощью некоторой функции, то какие функции можно в этот момент безопасно вызывать из других потоков?

Это на самом деле весьма важный вопрос. Обычно конструкторы и деструкторы нуждаются в монопольном доступе к структуре данных, но обязанность не обращаться к структуре до завершения конструирования или после начала уничтожения возлагается на пользователя. Если структура поддерживает присваивание, функцию `swap()` или копирующий конструктор, то проектировщик должен решить, безопасно ли вызывать эти операции одновременно с другими или пользователь должен обеспечить на время их выполнения монопольный доступ, хотя большинство других операций можно без опаски выполнять параллельно из разных потоков.

Второй аспект, нуждающийся в рассмотрении, — обеспечение истинно параллельного доступа. Тут я не могу предложить конкретных рекомендаций, а вместо этого перечислю несколько вопросов, которые должен задать себе проектировщик структуры данных.

• Можно ли ограничить область действия блокировок, так чтобы некоторые части операции выполнялись не под защитой блокировки?

• Можно ли защитить разные части структуры данных разными мьютексами?

• Все ли операции нуждаются в одинаковом уровне защиты?

• Можно ли с помощью простого изменения структуры данных расширить возможности распараллеливания, не затрагивая семантику операций?

В основе всех этих вопросов лежит одна и та же мысль: как свести к минимуму необходимую сериализацию и обеспечить максимально возможную степень истинного параллелизма? Часто бывает так, что структура данных допускает одновременный доступ из нескольких потоков для чтения, но поток, желающий модифицировать данные, должен получать монопольный доступ. Такое требование поддерживает класс `boost::shared_mutex` и ему подобные. Как мы скоро увидим, встречается и другой случай: поддерживается одновременный доступ из потоков, выполняющих различные операции над структурой, но потоки, выполняющие одну и ту же операцию, сериализуются.

В простейших потокобезопасных структурах данных обычно для защиты используются мьютексы и блокировки. Хотя, как мы видели в главе 3, им свойственны некоторые проблемы, но гарантировать с их помощью, что в каждый момент времени доступ к данным будет иметь только один поток, сравнительно легко. Мы будем знакомиться с проектированием потокобезопасных структур данных постепенно, и в этой главе рассмотрим только структуры на основе блокировок. А разговор о параллельных структурах данных без блокировок отложим до главы 7.

6.2. Параллельные структуры данных с блокировками

Проектирование параллельных структур данных с блокировками сводится к тому, чтобы захватить нужный мьютекс при доступе к данным и удерживать его минимально возможное время. Это довольно сложно, даже когда имеется только один мьютекс, защищающий всю структуру. Как мы видели в главе 3, требуется гарантировать, что к данным невозможно обратиться без защиты со стороны мьютекса и что интерфейс свободен от внутренне присущих состояний гонки. Если для защиты отдельных частей структуры применяются разные мьютексы, то проблема еще усложняется, поскольку в случае, когда некоторые операции требуют захвата нескольких мьютексов, появляется возможность взаимоблокировки. Поэтому к проектированию структуры данных с несколькими мьютексами следует подходить еще более внимательно, чем при наличии единственного мьютекса. В этом разделе мы применим рекомендации из раздела 6.1.1 к проектированию нескольких простых структур данных, защищаемых мьютексами. В каждом случае мы будем искать возможности повысить уровень параллелизма, обеспечивая в то же время потокобезопасность.

Начнем с реализации стека, приведённой в главе 3; это одна из самых простых структур данных, к тому же в ней используется всего один мьютекс. Но является ли она потокобезопасной? И насколько она хороша с точки зрения достижения истинного распараллеливания?

6.2.1. Потокобезопасный стек с блокировками

В следующем листинге воспроизведен код потокобезопасного стека из главы 3. Задача состояла в том, чтобы реализовать потокобезопасную структуру данных наподобие `std::stack<>`, которая поддерживала бы операции заталкивания и выталкивания.

Листинг 6.1. Определение класса потокобезопасного стека

```
#include <exception>

struct empty_stack: std::exception {
 const char* what() const throw();
};

template<typename T>
class threadsafe_stack {
private:
 std::stack<T> data;
 mutable std::mutex m;
public:
 threadsafe_stack(){}
 threadsafe_stack(const threadsafe_stack& other) {
  std::lock_guard<std::mutex> lock(other.m);
  data = other.data;
 }
 threadsafe_stack& operator=(const threadsafe_stack&) = delete;

 void push(T new_value) {
```

```
  std::lock_guard<std::mutex> lock(m);
  data.push(std::move(new_value)); ← (1)
 }

 std::shared_ptr<T> pop() {
  std::lock_guard<std::mutex> lock(m);
  if (data.empty()) throw empty_stack(); ← (2)
  std::shared_ptr<T> const res(
   std::make_shared<T>(std::move(data.top())));← (3)
  data.pop(); ← (4)
  return res;
 }

 void pop(T& value) {
  std::lock_guard<std::mutex> lock(m);
  if (data.empty()) throw empty_stack();
  value = std::move(data.top()); ← (5)
  data.pop(); ← (6)
 }

 bool empty() const {
  std::lock_guard<std::mutex> lock(m);
  return data.empty();
 }
};
```

Посмотрим, как в этом случае применяются сформулированные выше рекомендации. Во-первых, легко видеть, что базовую потокобезопасность обеспечивает защита каждой функции-члена с помощью мьютекса `m`. Он гарантирует, что в каждый момент времени к данным может обращаться только один поток, поэтому если функции-члены поддерживают какие-то инварианты, то ни один поток не увидит их нарушения.

Во-вторых, существует потенциальная гонка между `empty()` и любой из функций `pop()`, но поскольку мы явно проверяем, что стек пуст, удерживая блокировку в `pop()`, эта гонка не проблематична. Возвращая извлеченные данные прямо в `pop()`, мы избегаем потенциальной гонки, которая могла бы случиться, если бы `top()` и `pop()` были отдельными функциями-членами, как в `std::stack<>`.

Далее, существует несколько возможных источников исключений. Операция захвата мьютекса может возбудить исключение, но, во-первых, это крайне редкий случай (свидетельствующий о проблемах в реализации мьютекса или о нехватке системных ресурсов), а, во-вторых, эта операция всегда выполняется в самом начале любой функции-члена. Поскольку в этот момент никакие данные еще не изменены, опасности нет. Операция освобождения мьютекса не может завершиться ошибкой, она всегда безопасна, а использование `std::lock_guard<>` гарантирует, что мьютекс не останется захваченным.

Вызов `data.push()` **(1)** может возбудить исключение, если его возбуждает копирование или перемещение данных либо если памяти недостаточно для увеличения размера структуры, в которой хранятся сами данные. В любом случае `std::stack<>` гарантирует безопасность, поэтому здесь проблемы тоже нет.

В первом перегруженном варианте `pop()` наш код может возбудить исключение `empty_stack` **(2)**, но в этот момент еще ничего не изменено, так что мы в безопасности.

Создание объекта `res` (3) может возбудить исключение по двум причинам: при обращении к `std::make_shared` может не хватить памяти для нового объекта и внутренних данных, необходимых для подсчёта ссылок, или копирующий либо перемещающий конструктор возбуждает исключение при копировании или перемещении данных в только что выделенную область памяти. В обоих случаях исполняющая среда C++ и стандартная библиотека гарантируют отсутствие утечек памяти и корректное уничтожение нового объекта (если он был создан). Поскольку мы все еще не модифицировали данные стека, все хорошо. Вызов `data.pop()` **(4)** гарантированно не возбуждает исключений, равно как и возврат результата, так что этот вариант `pop()` безопасен относительно исключений.

Второй перегруженный вариант `pop()` аналогичен, только на этот раз исключение может возбудить оператор копирующего или перемещающего присваивания **(5)**, а не конструктор нового объекта или экземпляра `std::shared_ptr`. Но и теперь мы ничего не изменяли до вызова функции `data.pop()` **(6)**, которая гарантированно не возбуждает исключений, так что и этот вариант безопасен относительно исключений.

Наконец, функция `empty()` вообще не изменяет данные, так что она точно безопасна относительно исключений

В этом коде есть две возможности для взаимоблокировки из-за того, что пользовательский код вызывается, когда удерживается блокировка: копирующий или перемещающий конструктор **(1)**, **(3)** и копирующий или перемещающий оператор присваивания **(5)** хранимых в стеке данных. И еще — `operator new`, который также мог бы быть определён пользователем. Если любая из этих функций вызовет функции-члены стека, в который вставляется или из которого удаляется элемент, либо затребует какую-либо блокировку в момент, когда удерживается блокировка, захваченная при вызове функции-члена стека, то может возникнуть взаимоблокировка. Однако было бы разумно возложить ответственность за это на пользователей стека; невозможно представить себе разумную реализацию операций добавления в стек и удаления из стека, которые не копировали бы данные и не выделяли память.

Поскольку все функции-члены используют для защиты данных класс `std::lock_guard<>`, их можно безопасно вызывать из любого количества потоков. Единственные небезопасные функции-члены — конструкторы и деструкторы, но эта проблема не особенно серьезна; объект можно сконструировать и уничтожить только один раз. Вызов функций-членов не полностью сконструированного или частично уничтоженного объекта — это всегда плохо, и к одновременности доступа отношения не имеет. Таким образом, пользователь должен гарантировать, что никакой другой поток не может обратиться к стеку, пока он не будет сконструирован полностью, и что любая операция доступа завершается до начала его уничтожения.

Хотя благодаря блокировке несколько потоков могут одновременно вызывать функции-члены стека, в каждый момент времени с ним реально работает не более одного потока. Такая *сериализация* потоков может снизить производительность приложения в случае, когда имеется значительная конкуренция за стек, — пока поток ожидает освобождения блокировки, он не выполняет никакой полезной работы. Кроме того, стек не предоставляет средств, позволяющих ожидать добавления элемента. Следовательно, если потоку нужно получить из стека элемент, то он должен периодически опрашивать его с помощью `empty()` или просто вызывать `pop()` и обрабатывать исключение `empty_stack`.

Поэтому для такого сценария данная реализация стека неудачна, так как ожидающий

поток должен либо впустую растрачивать драгоценные ресурсы, ожидая данных, либо пользователь должен писать внешний код ожидания и извещения (например, с помощью условных переменных), который сделает внутренний механизм блокировки избыточным и, стало быть, расточительным. Приведенная в главе 4 реализация очереди демонстрирует, как можно включить такое ожидание в саму структуру данных с помощью условной переменной. Это и станет нашим следующим примером.

6.2.2. Потокобезопасная очередь с блокировками и условными переменными

В листинге 6.2 воспроизведен код потокобезопасной очереди из главы 4. Если стек построен по образцу `std::stack<>`, то очередь — по образцу `std::queue<>`. Но ее интерфейс также отличается от стандартного адаптера контейнера, потому что запись в структуру данных должна быть безопасной относительно одновременного доступа из нескольких потоков.

Листинг 6.2. Потокобезопасная очередь с блокировками и условными переменными

```
template<typename T>
class threadsafe_queue {
private:
 mutable std::mutex mut;
 std::queue<T> data_queue;
 std::condition_variable data_cond;
public:
 threadsafe_queue() {}

 void push(T new_value) {
  std::lock_guard<std::mutex> lk(mut);
  data_queue.push(std::move(data));
  data_cond.notify_one(); ← (1)
 }

 void wait_and_pop(T& value) { ← (2)
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{return !data_queue.empty();});
  value = std::move(data_queue.front());
  data_queue.pop();
 }

 std::shared_ptr<T> wait_and_pop() ← (3)
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this] {return !data_queue.empty();}); ← (4)
  std::shared_ptr<T> res(
   std::make_shared<T>(std::move(data_queue.front())));
  data_queue.pop();
  return res;
 }

 bool try_pop(T& value) {
  std::lock_guard<std::mutex> lk(mut);
```

```
    if (data_queue.empty())
     return false;
    value = std::move(data_queue.front());
    data_queue.pop();
    return true;
   }

   std::shared_ptr<T> try_pop() {
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
     return std::shared_ptr<T>(); ← (5)
    std::shared_ptr<T> res(
     std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
   }

   bool empty() const {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
   }
  };
```

Структурно очередь в листинге 6.2 реализована аналогично стеку в листинге 6.1, отличие только в обращениях к функции `data_cond.notify_one()` в `push()` **(1)** и в наличии двух вариантов функции `wait_and_pop()` **(2)**, **(3)**. Оба перегруженных варианта `try_pop()` почти идентичны функциям `pop()` в листинге 6.1 с тем отличием, что не возбуждают исключение, если очередь пуста. Вместо этого одна функция возвращает булевское значение, показывающее, были ли извлечены данные, а вторая — возвращающая указатель на данные **(5)** — указатель `NULL`. Точно так же можно было бы поступить и в случае стека. Таким образом, если оставить в стороне функции `wait_and_pop()`, то применим тот же анализ, который мы провели для стека.

Новые функции `wait_and_pop()` решают проблему ожидания значения в очереди, с которой мы столкнулись при обсуждении стека; вместо того чтобы раз за разом вызывать `empty()`, ожидающий поток может просто вызвать `wait_and_pop()`, а структура данных обслужит этот вызов с помощью условной переменной. Обращение к `data_cond.wait()` не вернет управление, пока во внутренней очереди не появится хотя бы один элемент, так что мы можем не беспокоиться но поводу того, что в этом месте кода возможна пустая очередь. При этом данные по-прежнему защищаются мьютексом. Таким образом, функции `wait_and_pop()` не вводят новых состояний гонки, не создают возможности взаимоблокировок и не нарушают никаких инвариантов.

В части безопасности относительно исключений есть мелкая неприятность — если помещения данных в очередь ожидают несколько потоков, то лишь один из них будет разбужен в результате вызова `data_cond.notify_one()`. Однако если этот поток возбудит исключение в `wait_and_pop()`, например при конструировании `std::shared_ptr<>` **(4)**, то ни один из оставшихся потоков разбужен не будет. Если это неприемлемо, то можно заменить `notify_one()` на `data_cond.notify_all()`, тогда будут разбужены все потоки, но за это придётся заплатить — большая часть из них сразу же уснет снова, увидев, что очередь по-прежнему пуста. Другой вариант — включить в `wait_and_pop()` обращение к `notify_one()` в случае исключения, тогда другой поток сможет попытаться извлечь

находящееся в очереди значение. Третий вариант — перенести инициализацию `std::shared_ptr<>` в `push()` и сохранять экземпляры `std::shared_ptr<>`, а не сами значения данных. Тогда при копировании `std::shared_ptr<>` из внутренней очереди `std::queue<>` никаких исключений возникнуть не может, и `wait_and_pop()` становится безопасной. В следующем листинге приведена реализация очереди, переработанная с учетом высказанных соображений.

Листинг 6.3. Потокобезопасная очередь, в которой хранятся объекты `std::shared_ptr`

```
template<typename T>
class threadsafe_queue {
private:
 mutable std::mutex mut;
 std::queue<std::shared_ptr<T> > data_queue;
 std::condition_variable data_cond;
public:
 threadsafe_queue() {}

 void wait_and_pop(T& value) {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{return !data_queue.empty();});
  value = std::move(*data_queue.front()); ← (1)
  data_queue.pop();
 }

 bool try_pop(T& value) {
  std::lock_guard<std::mutex> lk(mut);
  if (data_queue.empty())
   return false;
  value = std::move(*data_queue.front()); ← (2)
  data_queue.pop();
  return true;
 }

 std::shared_ptr<T> wait_and_pop() {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{return !data_queue.empty();});
  std::shared_ptr<T> res = data_queue.front(); ← (3)
  data_queue.pop();
  return res;
 }

 std::shared_ptr<T> try_pop() {
  std::lock_guard<std::mutex> lk(mut);
  if (data_queue.empty())
   return std::shared_ptr<T>();
  std::shared_ptr<T> res = data_queue.front(); ← (4)
  data_queue.pop();
  return res;
 }

 void push(T new_value) {
  std::shared_ptr<T> data(
```

```
    std::make_shared<T>(std::move(new_value))); ← (5)
   std::lock_guard<std::mutex> lk(mut);
   data_queue.push(data);
   data_cond.notify_one();
  }

  bool empty() const {
   std::lock_guard<std::mutex> lk(mut);
   return data_queue.empty();
  }
};
```

Последствия хранения данных, обернутых в `std::shared_ptr<>`, понятны: функции pop, которые получают значение из очереди в виде ссылки на переменную, теперь должны разыменовывать указатель **(1)**, **(2)**, а функции `pop`, которые возвращают `std::shared_ptr<>`, теперь могут напрямую извлекать его из очереди **(3)**, **(4)** без дальнейших манипуляций.

У хранения данных в виде `std::shared_ptr<>` есть и еще одно преимущество: выделение памяти для нового объекта можно производить не под защитой блокировки в `push()` **(5)**, тогда как в листинге 6.2 это приходилось делать в защищенном участке кода внутри `pop()`. Поскольку выделение памяти, вообще говоря, дорогая операция, это изменение весьма благотворно скажется на общей производительности очереди, так как уменьшается время удержания мьютекса, а, значит, у остальных потоков остается больше времени на полезную работу.

Как и в примере стека, применение мьютекса для защиты всей структуры данных ограничивает возможности распараллеливания работы с очередью; хотя ожидать доступа к очереди могут несколько потоков, выполняющих разные функции, в каждый момент лишь один совершает какие-то действия. Однако это ограничение отчасти проистекает из того, что мы пользуемся классом `std::queue<>`, — стандартный контейнер составляет единый элемент данных, который либо защищен, либо нет. Полностью взяв на себя управление деталями реализации структуры данных, мы сможем обеспечить мелкогранулярные блокировки и повысить уровень параллелизма.

6.2.3. Потокобезопасная очередь с мелкогранулярными блокировками и условными переменными

В листингах 6.2 и 6.3 имеется только один защищаемый элемент данных (`data_queue`) и, следовательно, только один мьютекс. Чтобы воспользоваться мелкогранулярными блокировками, мы должны заглянуть внутрь очереди и связать мьютекс с каждым хранящимся в ней элементом данных.

Проще всего реализовать очередь в виде односвязного списка, как показано на рис. 6.1. Указатель *head* направлен на первый элемент списка, и каждый элемент указывает на следующий. Когда данные извлекаются из очереди, в *head* записывается указатель на следующий элемент, после чего возвращается элемент, который до этого был в начале.

Добавление данных производится с другого конца. Для этого нам необходим указатель *tail*, направленный на последний элемент списка. Чтобы добавить узел, мы записываем в поле *next* в последнем элементе указатель на новый узел, после чего изменяем указатель *tail*, так чтобы он адресовал новый элемент. Если список пуст, то оба указателя *head* и *tail* равны

NULL.

В следующем листинге показана простая реализация такой очереди с урезанным по сравнению с листингом 6.2 интерфейсом; мы оставили только функцию `try_pop()` и убрали функцию `wait_and_pop()`, потому что эта очередь поддерживает только однопоточную работу.

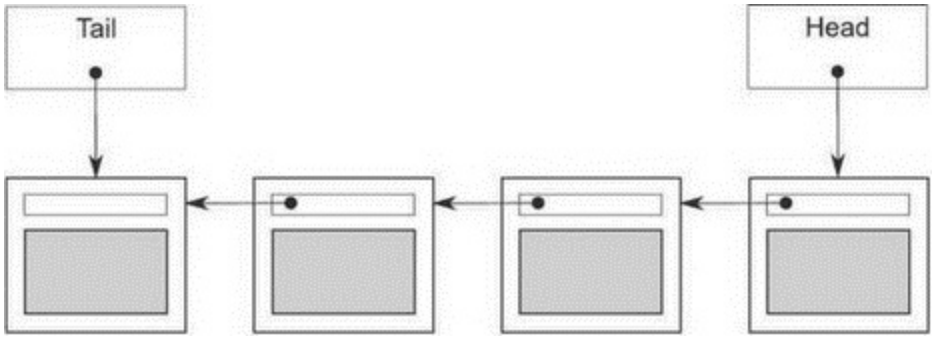

Рис. 6.1. Очередь, представленная в виде односвязного списка

Листинг 6.4. Простая реализация однопоточной очереди

```
template<typename T>
class queue {
private:
 struct node {
  T data;
  std::unique_ptr<node> next;
  node(T data_):
   data(std::move(data_)) {}
 };
 std::unique_ptr<node> head;← (1)
 node* tail;                ← (2)

public:
 queue() {}
 queue(const queue& other) = delete;
 queue& operator=(const queue& other) = delete;

 std::shared_ptr<T> try_pop() {
  if (!head) {
   return std::shared_ptr<T>();
  }
  std::shared_ptr<T> const res(
   std::make_shared<T>(std::move(head->data)));
  std::unique_ptr<node> const old_head = std::move(head);
  head = std::move(old_head->next);← (3)
  return res;
 }

 void push(T new_value) {
  std::unique_ptr<node> p(new node(std::move(new_value)));
  node* const new_tail = p.get();
  if (tail) {
   tail->next = std::move(p);← (4)
  } else {
   head = std::move(p);      ← (5)
```

```
    }
    tail = new_tail;                ← (6)
  }
};
```

Прежде всего отметим, что в листинге 6.4 для управления узлами используется класс `std::unique_ptr<node>`, потому что он гарантирует удаление потерявших актуальность узлов (и содержащихся в них данных) без явного использования `delete`. За передачу владения отвечает `head`, тогда как `tail` является простым указателем на последний узел.

В однопоточном контексте эта реализация прекрасно работает, но при попытке ввести мелкогранулярные блокировки в многопоточном контексте возникают две проблемы. Учитывая наличие двух элементов данных (`head` **(1)** и `tail` **(2)**), мы в принципе могли бы использовать два мьютекса — для защиты `head` и `tail` соответственно. Но не всё так просто.

Самая очевидная проблема заключается в том, что `push()` может изменять как `head` **(5)**, так и `tail` **(6)**, поэтому придётся захватывать оба мьютекса. Это не очень хорошо, но не трагедия, потому что захватить два мьютекса, конечно, можно. Настоящая проблема возникает из-за того, что и `push()`, и `pop()` обращаются к указателю `next` в узле: `push()` обновляет `tail->next` **(4)**, а `try_pop()` читает `head->next` **(3)**. Если в очереди всего один элемент, то `head==tail`, и, значит, `head->next` и `tail->next` — один и тот же объект, который, следовательно, нуждается в защите. Поскольку нельзя сказать, один это объект или нет, не прочитав и `head`, и `tail`, нам приходится захватывать один и тот же мьютекс в `push()` и в `try_pop()`, и получается, что мы ничего не выиграли по сравнению с предыдущей реализацией. Есть ли выход из этого тупика?

Обеспечение параллелизма за счет отделения данных

Решить проблему можно, заранее выделив фиктивный узел, не содержащий данных, и тем самым гарантировать, что в очереди всегда есть хотя бы один узел, отделяющий голову от хвоста. В случае пустой очереди `head` и `tail` теперь указывают на фиктивный узел, а не равны `NULL`. Это хорошо, потому что `try_pop()` не обращается к `head->next`, если очередь пуста. После добавления в очередь узла (в результате чего в ней находится один реальный узел) `head` и `tail` указывают на разные узлы, так что гонки за `head->next` и `tail->next` не возникает. Недостаток этого решения в том, что нам пришлось добавить лишний уровень косвенности для хранения указателя на данные, чтобы поддержать фиктивный узел. В следующем листинге показано, как теперь выглядит реализация.

Листинг 6.5. Простая очередь с фиктивным узлом

```
template<typename T>
class queue {
private:
 struct node {
  std::shared_ptr<T> data;← (1)
  std::unique_ptr<node> next;
 };

 std::unique_ptr<node> head;
 node* tail;
```

```
public:
 queue():
 head(new node), tail(head.get())← (2)
 {}

 queue(const queue& other) = delete;
 queue& operator=(const queue& other) = delete;
 std::shared_ptr<T> try_pop() {
  if (head.get() ==tail) ← (3)
  {
   return std::shared_ptr<T>();
  }
  std::shared_ptr<T> const res(head->data);← (4)
  std::unique_ptr<node> old_head = std::move(head);
  head = std::move(old_head->next); ← (5)
  return res; ← (6)
 }

 void push(T new_value) {
  std::shared_ptr<T> new_data(
   std::make_shared<T>(std::move(new_value)));← (7)
  std::unique_ptr<node> p(new node); ← (8)
  tail->data = new_data; ← (9)
  node* const new_tail = p.get();
  tail->next = std::move(p);
  tail = new_tail;
 }
};
```

Изменения в `try_pop()` минимальны. Во-первых, мы сравниваем `head` с `tail` **(3)**, а не с `NULL`, потому что благодаря наличию фиктивного узла `head` никогда не может обратиться в `NULL`. Поскольку `head` имеет тип `std::unique_ptr<node>`, для сравнения необходимо вызывать `head.get()`. Во-вторых, так как в `node` теперь хранится указатель на данные **(1)**, то можно извлекать указатель непосредственно **(4)** без конструирования нового экземпляра `T`. Наиболее серьезные изменения претерпела функция `push()`: мы должны сначала создать новый экземпляр `T` в куче и передать владение им `std::shared_ptr<>` **(7)** (обратите внимание на использование функции `std::make_shared`, чтобы избежать накладных расходов на второе выделение памяти под счетчик ссылок). Вновь созданный узел станет новым фиктивным узлом, поэтому передавать конструктору значение `new_value` необязательно **(8)**. Вместо этого мы записываем в старый фиктивный узел значение только что созданной копии — `new_value` **(9)**. Наконец, первоначальный фиктивный узел следует создать в конструкторе **(2)**.

Уверен, теперь вы задаетесь вопросом, что мы выиграли от всех этих изменений и как они помогут сделать код потокобезопасным. Разберемся. Функция `push()` теперь обращается только к `tail`, но не к `head`, и это, безусловно, улучшение. `try_pop()` обращается и к `head`, и к `tail`, но `tail` нужен только для начального сравнения, так что блокировка удерживается очень недолго. Основной выигрыш мы получили за счет того, что из-за наличия фиктивного узла `try_pop()` и `push()` никогда не оперируют одним и тем же узлом, так что нам больше

не нужен всеохватывающий мьютекс. Стало быть, мы можем завести по одному мьютексу для `head` и `tail`. Но где расставить блокировки?

Мы хотим обеспечить максимум возможностей для распараллеливания, поэтому блокировки должны освобождаться как можно быстрее. С функцией `push()` всё просто: мьютекс должен быть заблокирован на протяжении всех обращений к `tail`, а это означает, что мы захватываем его после выделения памяти для нового узла **(8)** и перед тем, как записать данные в текущий последний узел **(9)**. Затем блокировку следует удерживать до конца функции.

С `try_pop()` сложнее. Прежде всего, нам нужно захватить мьютекс для `head` и удерживать его, пока мы не закончим работать с `head`. По сути дела, этот мьютекс определяет, какой поток производит извлечение из очереди, поэтому захватить его надо в самом начале. После того как значение `head` изменено **(5)**, мьютекс можно освободить; в момент, когда возвращается результат **(6)**, он уже не нужен. Остается разобраться с защитой доступа к `tail`. Поскольку мы обращаемся к `tail` только один раз, то можно захватить мьютекс на время, требуемое для чтения. Проще всего сделать это, поместив операцию доступа в отдельную функцию. На самом деле, поскольку участок кода, в котором мьютекс для `head` должен быть заблокирован, является частью одной функции-члена, то будет правильнее завести отдельную функцию и для него тоже. Окончательный код приведён в листинге 6.6.

Листинг 6.6. Потокобезопасная очередь с мелкогранулярными блокировками

```
template<typename T>
class threadsafe_queue {
private:
 struct node {
  std::shared_ptr<T> data;
  std::unique_ptr<node> next;
 };

 std::mutex head_mutex;
 std::unique_ptr<node> head;
 std::mutex tail_mutex;
 node* tail;

 node* get_tail() {
  std::lock_guard<std::mutex> tail_lock(tail_mutex);
  return tail;
 }

 std::unique_ptr<node> pop_head() {
  std::lock_guard<std::mutex> head_lock(head_mutex);
  if (head.get() == get_tail()) {
   return nullptr;
  }
  std::unique_ptr<node> old_head = std::move(head);
  head = std::move(old_head->next);
  return old_head;
 }

public:
```

```
    threadsafe_queue():
     head(new node), tail(head.get()) {}
    threadsafe_queue(const threadsafe_queue& other) = delete;
    threadsafe_queue& operator=(
     const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
     std::unique_ptr<node> old_head = pop_head();
     return old_head ? old_head->data : std::shared_ptr<T>();
    }

    void push(T new_value) {
     std::shared_ptr<T> new_data(
      std::make_shared<T>(std::move(new_value)));
     std::unique_ptr<node> p(new node);
     node* const new_tail = p.get();
     std::lock_guard<std::mutex> tail_lock(tail_mutex);
     tail->data = new_data;
     tail->next = std::move(p);
     tail = new_tail;
    }
};
```

Давайте взглянем на этот код критически, памятуя о рекомендациях из раздела 6.1.1. Прежде чем искать, где нарушены инварианты, надо бы их точно сформулировать:

- `tail->next == nullptr`.
- `tail->data == nullptr`.
- `head == tail` означает, что список пуст.
- Для списка с одним элементом `head->next==tail`.
- Для каждого узла `x` списка, для которого `x!=tail`, `x->data` указывает на экземпляр `T`, а `x->next` — на следующий узел списка. Если `x->next==tail`, то `x` — последний узел списка.
- Если проследовать по указателям `next`, начиная с головы списка, то рано или поздно мы достигнем его хвоста.

Сама по себе, функция `push()` очень проста: все модификации данных защищены мьютексом `tail_mutex`, и инвариант при этом сохраняется, потому что новый хвостовой узел пуст и правильно установлены указатели `data` и `next` для старого хвостового узла, который теперь стал настоящим последним узлом списка.

Самое интересное происходит в функции `try_pop()`. Как выясняется, мьютекс `tail_mutex` нужен не только для защиты чтения самого указателя `tail`, но и чтобы предотвратить гонку при чтении данных из головного узла. Не будь этого мьютекса, могло бы получиться, что один поток вызывает `try_pop()`, а другой одновременно вызывает `push()`, и эти операции никак не упорядочиваются. Хотя каждая функция-член удерживает мьютекс, но это разные *мьютексы*, а функции могут обращаться к одним и тем же данным — ведь все данные появляются в очереди только благодаря `push()`. Раз потоки потенциально могут обращаться к одним и тем же данным без какого бы то ни было упорядочения, то возможна гонка за данными и, как следствие (см. главу 5), неопределенное поведение. К счастью, блокировка мьютекса `tail_mutex` в `get_tail()` решает все проблемы. Поскольку внутри `get_tail()` захватывается тот же мьютекс, что в `push()`, то оба вызова оказываются упорядоченными. Либо обращение к функции `get_tail()` происходит раньше обращения к `push()` — тогда `get_tail()` увидит старое значение `tail` — либо после обращения к `push()`

— и тогда она увидит новое значение `tail` *и новые данные, присоединенные к прежнему значению* `tail`.

Важно также, что обращение к `get_tail()` производится под защитой захваченного мьютекса `head_mutex`. Если бы это было не так, то между вызовом `get_tail()` и захватом `head_mutex` мог бы вклиниться вызов `pop_head()`, потому что другой поток вызвал `try_pop()` (и, следовательно, `pop_head()`) и захватил мьютекс первым, не давая первому потоку продолжить исполнение:

```
                                          | Эта реализация
std: :unique_ptr<node> pop_head()↵        | некорректна
{                                         |  (1) Старое значение tail
                                          | получено не
 node* const old_tail = get_tail();↵      | под защитой head_mutex
 std::lock_guard<std::mutex> head_lock(head_mutex);
 if (head.get() == old_tail)       ← (2)
  return nullptr;
 }
 std::unique_ptr<node> old_head = std::move(head);
 head = std::move(old_head->next);← (3)
 return old_head;
}
```

При такой — *некорректной* — реализации в случае, когда `get_tail()` **(1)** вызывается вне области действия блокировки, может оказаться, что и `head`, и `tail` изменились к моменту, когда первому потоку удалось захватить `head_mutex`, и теперь возвращенный хвост мало того что больше не является хвостом, но и вообще не принадлежит списку. Тогда сравнение `head` с `old_tail` **(2)** не прошло бы, хотя `head` в действительности является последним узлом. Следовательно, после обновления **(3)** узел `head` мог бы оказаться в списке дальше `tail`, то есть за концом списка, что полностью разрушило бы структуру данных. В *корректной* реализации, показанной в листинге 6.6, вызов `get_tail()` производится под защитой `head_mutex`. Это означает, что больше никакой поток не сможет изменить `head`, а `tail` будет только отодвигаться от начала списка (по мере добавления новых узлов с помощью `push()`) — это вполне безопасно. Указатель `head` никогда не сможет оказаться дальше значения, возвращенного `get_tail()`, так что инварианты соблюдаются.

После того как `pop_head()` удалит узел из очереди, обновив `head`, мьютекс освобождается, и `try_pop()` может извлечь данные и удалить узел, если таковой был (или вернуть `NULL`-экземпляр класса `std::shared_ptr<>`, если узла не было), твердо зная, что она работает в единственном потоке, который имеет доступ к этому узлу.

Далее, внешний интерфейс является подмножеством интерфейса из листинга 6.2, поэтому ранее выполненный анализ остается в силе: в интерфейсе нет внутренне присущих состояний гонки.

Вопрос об исключениях более интересен. Поскольку мы изменили порядок выделения памяти, исключения могут возникать в других местах. Единственные операции в `try_pop()`, способные возбудить исключение, — это захваты мьютексов, но пока мьютексы не захвачены, данные не модифицируются. Поэтому `try_pop()` безопасна относительно исключений. С другой стороны, `push()` выделяет из кучи память для объектов `T` и `node`, и каждая такая операция может возбудить исключение. Однако оба вновь созданных объекта присваиваются интеллектуальным указателям, поэтому в случае исключения память

корректно освобождается. После того как мьютекс захвачен, ни одна из последующих операций внутри `push()` не может возбудить исключение, так что мы снова в безопасности.

Поскольку мы не изменяли интерфейс, то новых внешних возможностей для взаимоблокировки не возникло. Внутренних возможностей также нет; единственное место, где захватываются два мьютекса, — это функция `pop_head()`, но она всегда захватывает сначала `head_mutex`, а потом `tail_mutex`, так что взаимоблокировки не случится.

Осталось рассмотреть только один вопрос — в какой мере возможно распараллеливание. Эта структура данных предоставляет куда больше таких возможностей, чем приведенная в листинге 6.2, потому что гранулярность блокировок мельче, и *больше работы выполняется не под защитой блокировок*. Например, в `push()` память для нового узла и нового элемента данных выделяется, когда ни одна блокировка не удерживается. Это означает, что несколько потоков могут спокойно выделять новые узлы и элементы данных в одно и то же время. В каждый момент времени только один поток может добавлять новый узел в список, но выполняющий это действие код сводится к нескольким простым присваиваниям указателей, так что блокировка удерживается совсем недолго по сравнению с реализацией на основе `std::queue<>`, где мьютекс остается захваченным в течение всего времени, пока выполняются операции выделения памяти внутри `std::queue<>`.

Кроме того, `try_pop()` удерживает `tail_mutex` лишь на очень короткое время, необходимое для защиты чтения `tail`. Следовательно, почти все действия внутри `try_pop()` могут производиться одновременно с вызовом `push()`. Объем операций, выполняемых под защитой мьютекса `head_mutex` также совсем невелик; дорогостоящая операция `delete` (в деструкторе указателя на узел) производится вне блокировки. Это увеличивает потенциальное число одновременных обращений к `try_pop()`; в каждый момент времени только один поток может вызывать `pop_head()`, зато несколько потоков могут удалять старые узлы и безопасно возвращать данные.

Ожидание поступления элемента

Ну хорошо, код в листинге 6.6 дает в наше распоряжение потокобезопасную очередь с мелкогранулярными блокировками, но он поддерживает только функцию `try_pop()` (и к тому же всего в одном варианте). А как насчет таких удобных функций `wait_and_pop()`, которые мы написали в листинге 6.2? Сможем ли мы реализовать идентичный интерфейс, сохранив мелкогранулярные блокировки?

Ответ, разумеется, — да, только вот как это сделать? Модифицировать `push()` несложно: нужно лишь добавить вызов `data_cond.notify_one()` в конец функции, как и было в листинге 6.2. Но на самом деле не всё так просто; мы же связались с мелкогранулярными блокировками для того, чтобы увеличить уровень параллелизма. Если оставить мьютекс захваченным на все время вызова `notify_one()` (как в листинге 6.2), то поток, разбуженный до того, как мьютекс освобожден, должен будет ждать мьютекса. С другой стороны, если освободить мьютекс *до* обращения к notify_one(), то ожидающий поток сможет захватить его сразу, как проснётся (если, конечно, какой-то другой поток не успеет раньше). Это небольшое улучшение, но в некоторых случаях оно бывает полезно.

Функция `wait_and_pop()` сложнее, потому что мы должны решить, где поместить ожидание, какой задать предикат и какой мьютекс захватить. Мы ждем условия «очередь не

пуста», оно представляется выражением `head != tail`. Если записать его в таком виде, то придется захватывать и `head_mutex`, и `tail_mutex`, но, разбирая код в листинге 6.6, мы уже поняли, что захватывать `tail_mutex` нужно только для чтения `tail`, а не для самого сравнения, та же логика применима и здесь. Если записать предикат в виде `head != get_tail()`, то нужно будет захватить только `head_mutex` и использовать уже полученную блокировку для защиты `data_cond.wait()`. Прочий код такой же, как в `try_pop()`.

Второй перегруженный вариант `try_pop()` и соответствующий ему вариант `wait_and_pop()` нуждаются в тщательном осмыслении. Если просто заменить возврат указателя `std::shared_ptr<>`, полученного из `old_head`, копирующим присваиванием параметру `value`, то функция перестанет быть безопасной относительно исключений. В этот момент элемент данных уже удален из очереди и мьютекс освобожден, осталось только вернуть данные вызывающей программе. Однако, если копирующее присваивание возбудит исключение (а почему бы и нет?), то элемент данных будет потерян, потому что вернуть его в то же место очереди, где он был, уже невозможно.

Если фактический тип `T`, которым конкретизируется шаблон, обладает не возбуждающими исключений оператором перемещающего присваивания или операцией обмена (`swap`), то так поступить можно, но ведь мы ищем общее решение, применимое к любому типу `T`. В таком случае следует поместить операции, способные возбудить исключения, в защищенную область перед тем, как удалять узел из списка. Это означает, что нам необходим еще один перегруженный вариант `pop_head()`, который извлекает сохраненное значение до модификации списка.

Напротив, модификация функции empty() тривиальна: нужно просто захватить `head_mutex` и выполнить проверку `head == get_tail()` (см. листинг 6.10). Окончательный код очереди приведён в листингах 6.7, 6.8, 6.9 и 6.10.

Листинг 6.7. Потокобезопасная очередь с блокировкой и ожиданием: внутренние данные и интерфейс

```
template<typename T>
class threadsafe_queue {
private:
 struct node {
  std::shared_ptr<T> data;
  std::unique_ptr<node> next;
 };
 std::mutex head_mutex;
 std::unique_ptr<node> head;
 std::mutex tail_mutex;
 node* tail;
 std::condition_variable data_cond;


public:
 threadsafe_queue():
  head(new node), tail(head.get()) {}
 threadsafe_queue(const threadsafe_queue& other) = delete;
 threadsafe_queue& operator=(
  const threadsafe_queue& other) = delete;
 std::shared_ptr<T> try_pop();
 bool try_pop(T& value);
 std::shared_ptr<T> wait_and_pop();
 void wait_and_pop(T& value);
```

```
 void push(T new_value);
 void empty();
};
```

Код, помещающий новые узлы в очередь, прост — его реализация (показанная в листинге ниже) близка к той, что мы видели раньше.

Листинг 6.8. Потокобезопасная очередь с блокировкой и ожиданием: добавление новых значений

```
template<typename T>
void threadsafe_queue<T>::push(T new_value) {
 std::shared_ptr<T> new_data(
  std::make_shared<T>(std::move(new_value)));

 std::unique_ptr<node> p(new node);
 {
  std::lock_guard<std::mutex> tail_lock(tail_mutex);
  tail->data = new_data;
  node* const new_tail = p.get();
  tail->next = std::move(p);
  tail = new_tail;
 }
 data_cond.notify_one();
}
```

Как уже отмечалось, вся сложность сосредоточена в части *pop*. В листинге ниже показана реализация функции-члена `wait_and_pop()` и относящихся к ней вспомогательных функций.

Листинг 6.9. Потокобезопасная очередь с блокировкой и ожиданием: `wait_and_pop`

```
template<typename T>
class threadsafe_queue {
private:
 node* get_tail() {
  std::lock_guard<std::mutex> tail_lock(tail_mutex);
  return tail;
 }

 std::unique_ptr<node> pop_head() {← (1)
  std::unique_ptr<node> old_head = std::move(head);
  head = std::move(old_head->next);
  return old_head;
 }

 std::unique_lock<std::mutex> wait_for_data() {← (2)
  std::unique_lock<std::mutex> head_lock(head_mutex);
  data_cond.wait(
   head_lock, [&]{return head.get() != get_tail();});
  return std::move(head_lock);                    ← (3)
 }

 std::unique_ptr<node> wait_pop_head() {
  std::unique_lock<std::mutex> head_lock(wait_for_data());← (4)
  return pop_head();
```

```
    }

    std::unique_ptr<node> wait_pop_head(T& value) {
     std::unique_lock<std::mutex> head_lock(wait_for_data());← (5)
     value = std::move(*head->data);
     return pop_head();
    }
   public:
    std::shared_ptr<T> wait_and_pop() {
     std::unique_ptr<node> const old_head = wait_pop_head();
     return old_head->data;
    }

    void wait_and_pop(T& value) {
     std::unique_ptr<node> const old_head = wait_pop_head(value);
    }
   };
```

В реализации извлечения из очереди используется несколько небольших вспомогательных функций, которые упрощают код и позволяют устранить дублирование, например: `pop_head()` **(1)** (модификация списка в результате удаления головного элемента) и `wait_for_data()` **(2)** (ожидание появления данных в очереди). Особенно стоит отметить функцию `wait_for_data()`, потому что она не только ждет условную переменную, используя лямбда-функцию в качестве предиката, но и возвращает объект блокировки вызывающей программе **(3)**. Тем самым мы гарантируем, что та же самая блокировка удерживается, пока данные модифицируются в соответствующем перегруженном варианте `wait_pop_head()` **(4)**, **(5)**. Функция `pop_head()` используется также в функции `try_pop()`, показанной ниже.

Листинг 6.10. Потокобозопасная очередь с блокировкой и ожиданием: `try_pop()` и `empty()`

```
   template<typename T>
   class threadsafe_queue {
   private:
    std::unique_ptr<node> try_pop_head() {
     std::lock_guard<std::mutex> head_lock(head_mutex);
     if (head.get() == get_tail()) {
      return std::unique_ptr<node>();
     }
     return pop_head();
    }

    std::unique_ptr<node> try_pop_head(T& value) {
     std::lock_guard<std::mutex> head_lock(head_mutex);
     if (head.get() == get_tail()) {
      return std::unique_ptr<node>();
     }
     value = std::move(*head->data);
     return pop_head();
    }

   public:
    std::shared_ptr<T> try_pop() {
     std::unique_ptr<node> old_head = try_pop_head();
     return old_head ? old_head->data : std::shared_ptr<T>();
```

```
  }

  bool try_pop(T& value) {
   std::unique_ptr<node> const old_head = try_pop_head(value);
   return old_head;
  }

  void empty() {
   std::lock_guard<std::mutex> head_lock(head_mutex);
   return (head.get() == get_tail());
  }
};
```

Эта реализация очереди ляжет в основу очереди без блокировок, которую мы будем рассматривать в главе 7. Данная очередь *неограниченная*, то есть в нее можно помещать и помещать данные, ничего не удаляя, пока не кончится память. Альтернативой является *ограниченная* очередь, максимальная длина которой задается при создании. Попытка поместить элемент в заполненную очередь либо завершается ошибкой, либо приводит к приостановке потока до тех пор, пока из очереди не будет удален хотя бы один элемент. Ограниченные очереди бывают полезны для равномерного распределения задач между потоками (см. главу 8). Такая дисциплина не дает потоку (или потокам), заполняющему очередь, намного обогнать потоки, читающие из очереди.

Показанную реализацию неограниченной очереди легко преобразовать в очередь с ограниченной длиной, введя ожидание условной переменной в функцию `push()`. Вместо того чтобы ждать, пока в очереди появятся элементы (как `pop()`), мы должны будем ждать, когда число элементов в ней окажется меньше максимума. Дальнейшее обсуждение ограниченных очередей выходит за рамки этой книги, мы же перейдём от очередей к более сложным структурам данных.

6.3. Проектирование более сложных структур данных с блокировками

Стеки и очереди — простые структуры данных, их интерфейс крайне ограничен, и используются они в очень узких целях. Но не все структуры данных так просты, как правило они поддерживают более широкий набор операций. В принципе, это может открыть больше возможностей для распараллеливания, но и проблема защиты данных сильно усложняется, так как приходится учитывать много разных способов доступа. При проектировании структур данных, предназначенных для параллельного доступа, важно принимать во внимание характер выполняемых операций.

Чтобы понять, какие трудности возникают на этом пути, рассмотрим справочную таблицу (lookup table).

6.3.1. Разработка потокобезопасной справочной таблицы с блокировками

Справочная таблица, или словарь позволяет ассоциировать значения одного типа (типа ключа) со значениями того же или другого типа (ассоциированного типа). В общем случае задача такой структуры — запрашивать данные, ассоциированные с данным ключом. В стандартной библиотеке C++ для этой цели предназначены ассоциативные контейнеры: `std::map<>`, `std::multimap<>`, `std::unordered_map<>`, и `std::unordered_multimap<>`.

Справочная таблица используется иначе, чем стек или очередь. Если большинство операций со стеком и очередью каким-то образом модифицируют структуру данных, то есть либо добавляют, либо удаляют элементы, то справочная таблица изменяется сравнительно редко. Примером может служить простой DNS-кэш из листинга 3.13, предоставляющий интерфейс, сильно урезанный по сравнению с `std::map<>`. При рассмотрении стека и очереди мы видели, что интерфейсы стандартных контейнеров не годятся для случая, когда к структуре данных одновременно обращаются несколько потоков, так как им внутренне присущи состояния гонки. Поэтому интерфейсы приходится пересматривать и сокращать.

Самая серьезная с точки зрения распараллеливания проблема в интерфейсе контейнера `std::map<>` — его итераторы. Можно написать итератор так, что доступ к контейнеру с его помощью для чтения и модификации будет безопасен, но это амбициозная задача. Для корректной работы нужно учесть, что другой поток может удалить элемент, на который итератор сейчас ссылается, а это совсем непросто. Поэтому при первом подходе к проектированию интерфейса потокобезопасной справочной таблицы итераторы мы опустим. Памятуя о том, насколько сильно интерфейс `std::map<>` (и прочих стандартных ассоциативных контейнеров) зависит от итераторов, пожалуй, будет разумнее сразу отказаться от попыток смоделировать их и вместо этого придумать новый интерфейс с нуля.

Справочной таблице нужно всего несколько основных операций:

- добавить новую пару ключ/значение;
- изменить значение, ассоциированное с данным ключом;
- удалить ключ и ассоциированное с ним значение;
- получить значение, ассоциированное с данным ключом, если такой ключ существует.

Есть также несколько полезных операций, относящихся к контейнеру в целом, например: проверить, пуст ли контейнер, получить полный текущий список ключей или

полное множество пар ключ/ значение.

Если придерживаться базовых рекомендаций, касающихся потокобезопасности, например, не возвращать ссылки и защищать мьютексом все тело каждой функции-члена, то все эти операции будут безопасны. Угроза возникновения гонки возникает прежде всего при добавлении новой пары ключ/значение; если два потока одновременно добавляют новое значение, то лишь один из них будет первым, а второй, следовательно, получит ошибку. Один из способов решить эту проблему состоит в том, чтобы объединить операции добавления и изменения в одной функции-члене, как мы проделали в DNS-кэше в листинге 3.13.

С точки зрения интерфейса, остается еще лишь один интересный момент: фраза «если такой ключ существует» в описании операции получения ассоциированного значения. Можно, например, предоставить пользователю возможность задать некий результат «по умолчанию», который будет возвращен, если указанного ключа нет.

```
mapped_type get_value(
 key_type const& key, mapped_type default_value);
```

В таком случае можно использовать сконструированный по умолчанию экземпляр типа `mapped_type`, если `default_value` не было задано явно. Эту идею можно развить и возвращать не просто экземпляр `mapped_type`, а объект типа `std::pair<mapped_type, bool>`, где `bool` указывает, было ли найдено значение. Другой вариант — использовать интеллектуальный указатель, ссылающийся на значение; если он равен `NULL`, то значение не было найдено.

Как уже отмечалось, после определения интерфейса (в предположении, что состояний гонки в нем нет), обеспечить потокобезопасность можно, заведя единственный мьютекс, который дет захватываться в начале каждой функции-члена для защиты внутренней структуры данных. Однако тем самым мы сведем на нет все возможности распараллеливания, которые могли бы открыться в результате наличия отдельных функций для чтения и модификации структуры данных. Отчасти исправить ситуацию можно было бы, воспользовавшись мьютексом, который разрешает нескольким потокам читать данные, но только одному — изменять их, например, классом `boost::shared_mutex` из листинга 3.13. Это, конечно, несколько улучшило бы общую картину, но при таком решении модифицировать структуру данных по-прежнему может только один поток. В идеале хотелось бы добиться большего.

Проектирование структуры данных для справочной таблицы с мелкогранулярными блокировками

Как и в случае очереди (см. раздел 6.2.3), чтобы ввести мелкогранулярные блокировки, нужно внимательно изучить особенности структуры данных, а не пытаться обернуть уже имеющийся контейнер, например `std::map<>`. Есть три общепринятых подхода к реализации ассоциативного контейнера, каковым является, в частности, справочная таблица:

- двоичное, например красно-черное, дерево;
- отсортированный массив;
- хеш-таблица.

Двоичное дерево плохо приспособлено для распараллеливания — каждый просмотр или модификация должны начинается с доступа к корневому узлу, который, следовательно,

нужно блокировать. Блокировку, конечно, можно освобождать по мере спуска вниз по дереву, но в целом это немногим лучше блокирования всей структуры целиком.

Отсортированный массив еще хуже, потому что заранее невозможно сказать, в каком месте массива может оказаться требуемое значение, поэтому придется заводить одну блокировку на весь массив. Остается только хеш-таблица. В предположении, что число кластеров фиксировано, вопрос о том, в каком кластере находится ключ, является свойством только лишь самого ключа и хеш-функции. А это значит, что мы сможем завести по одной блокировке на каждый кластер. Если еще и использовать мьютекс, который поддерживает несколько читателей и одного писателя, то коэффициент распараллеливания увеличится в *N* раз, где *N* — число кластеров. Недостаток в том, что нужна хорошая хеш-функция для ключа. В стандартной библиотеке C++ имеется шаблон `std::hash<>`, которым можно воспользоваться для этой цели. Он уже специализирован для таких фундаментальных типов, как `int`, и некоторых библиотечных типов, например `std::string`, а пользователь может без труда написать специализации и для других типов ключа. Если, следуя примеру стандартных неупорядоченных контейнеров, указать в качестве параметра шаблона тип объекта-функции, которая выполняет хеширование, то пользователь сможет сам решить, специализировать ли ему шаблон `std::hash<>` для типа своего ключа или предоставить отдельную хеш-функцию.

Теперь обратимся собственно к коду. Как могла бы выглядеть реализация потокобезопасной справочной таблицы? Один из вариантов показан ниже.

Листинг 6.11. Потокобезопасная справочная таблица

```
template<typename Key, typename Value,
         typename Hash = std::hash<Key> >
class threadsafe_lookup_table {
private:
 class bucket_type {
 private:
  typedef std::pair<Key, Value> bucket_value;
  typedef std::list<bucket_value> bucket_data;
  typedef typename bucket_data::iterator bucket_iterator;

  bucket_data data;
  mutable boost::shared_mutex mutex;← (1)

  bucket_iterator find_entry_for(Key const& key) const {← (2)
   return std::find_if(data.begin(), data.end(),
                       [&](bucket_value const& item)
   { return item.first == key; });
  }

 public:
  Value value_for(
   Key const& key, Value const& default_value) const {
   boost::shared_lock<boost::shared_mutex> lock(mutex);← (3)
   bucket_iterator const found_entry = find_entry_for(key);
   return (found_entry==data.end()) ?
          default_value : found_entry->second;
  }

  void add_or_update_mapping(
```

```
    Key const& key, Value const& value) {
    std::unique_lock<boost::shared_mutex> lock(mutex);← (4)
    bucket_iterator const found_entry = find_entry_for(key);
    if (found_entry == data.end()) {
     data.push_back(bucket_value(key, value));
    } else {
     found_entry->second = value;
    }
   }

   void remove_mapping(Key const& key) {
    std::unique_lock<boost::shared_mutex> lock(mutex);← (5)
    bucket_iterator const found_entry = find_entry_for(key);
    if (found_entry != data.end()) {
     data.erase(found_entry);
    }
   }
  };

  std::vector<std::unique_ptr<bucket_type>> buckets;← (6)
  Hash hasher;

  bucket_type& get_bucket(Key const& key) const {← (7)
   std::size_t const bucket_index = hasher(key)%buckets.size();
   return *buckets[bucket_index];
  }
 public:
  typedef Key key_type;
  typedef Value mapped_type;
  typedef Hash hash_type;

  threadsafe_lookup_table(
   unsigned num_buckets = 19,
   Hash const& hasher_ = Hash()):
   buckets(num_buckets), hasher(hasher_) {
   for (unsigned i = 0; i < num_buckets; ++i) {
    buckets[i].reset(new bucket_type);
   }
  }

  threadsafe_lookup_table(
   threadsafe_lookup_table const& other) = delete;
  threadsafe_lookup_table& operator=(
   threadsafe_lookup_table const& other) = delete;

  Value value_for(Key const& key,
   Value const& default_value = Value()) const {
   return get_bucket(key).value_for(key, default_value);← (8)
  }

  void add_or_update_mapping(Key const& key,Value const& value) {
   get_bucket(key).add_or_update_mapping(key, value);← (9)
  }
```

```
  void remove_mapping(Key const& key) {
   get_bucket(key).remove_mapping(key);← (10)
  }
};
```

В этой реализации для хранения кластеров используется вектор `std::vector<std::unique_ptr<bucket_type>>` **(6)**, это позволяет задавать число кластеров в конструкторе. По умолчанию оно равно 19 (произвольно выбранное простое число); оптимальные показатели работы хеш-таблиц достигаются, когда имеется простое число кластеров. Каждый кластер защищен мьютексом типа `boost::shared_mutex` **(1)**, который позволяет нескольким потокам одновременно читать, но только одному обращаться к любой из функций модификации *данного кластера*.

Поскольку количество кластеров фиксировано, функцию `get_bucket()` **(7)** можно вызывать вообще без блокировки **(8)**, **(9)**, **(10)**, а затем захватить мьютекс кластера для совместного (только для чтения) **(3)** или монопольного (чтение/запись) **(4)**, **(5)** владения — в зависимости от вызывающей функции.

Во всех трех функциях используется функция-член кластера `find_entry_for()` **(2)**, которая определяет, есть ли в данном кластере указанный ключ. Каждый кластер содержит всего лишь список `std::list<>` пар ключ/значение, поэтому добавлять и удалять элементы легко.

Я уже рассматривал это решение с точки зрения распараллеливания, и мы убедились, что все надежно защищено мьютексами. Но как обстоит дело с безопасностью относительно исключений? Функция `value_for` ничего не изменяет, поэтому с ней проблем нет: если она и возбудит исключение, то на структуру данных это не повлияет.

Функция `remove_mapping` модифицирует список, обращаясь к его функции-члену `erase`, которая гарантированно не возбуждает исключений, так что здесь тоже всё безопасно. Остается функция `add_or_update_mapping`, которая может возбуждать исключения в обеих ветвях `if`. Функция `push_back` безопасна относительно исключений, то есть в случае исключения оставляет список в исходном состоянии, так что с этой ветвью всё хорошо. Единственная проблема — присваивание в случае замены существующего значения; если оператор присваивания возбуждает исключение, то мы полагаемся на то, что он оставляет исходный объект неизмененным. Однако это не влияет на структуру данных в целом и является свойством пользовательского типа, так что пускай пользователь и решает эту проблему.

В начале этого раздела я упоминал, что было бы хорошо, если бы можно было получить текущее состояние справочной таблицы, например, в виде объекта `std::map<>`. Чтобы копия состояния была согласована, потребуется заблокировать контейнер целиком, то есть все кластеры сразу. Поскольку для «обычных» операций захватывается мьютекс только одного кластера, то получение копии состояния будет единственной операцией, блокирующей все кластеры. При условии, что мьютексы всегда захватываются в одном и том же порядке, взаимоблокировка не возникнет. Такая реализация приведена в следующем листинге.

Листинг 6.12. Получение содержимого таблицы `threadsafe_lookup_table` в виде `std::map<>`

```
std::map<Key, Value> threadsafe_lookup_table::get_map() const {
 std::vector<std::unique_lock<boost::shared_mutex> > locks;
```

```
  for (unsigned i = 0; i < buckets.size(); ++i) {
   locks.push_back(
    std::unique_lock<boost::shared_mutex>(buckets[i].mutex));
  }

  std::map<Key, Value> res;
  for (unsigned i = 0; i < buckets.size(); ++i) {
   for (bucket_iterator it = buckets[i].data.begin();
   it != buckets[i].data.end(); ++it) {
    res.insert(*it);
   }
  }
  return res;
}
```

Реализация справочной таблицы, показанная в листинге 6.11, увеличивает уровень параллелизма таблицы в целом за счет того, что каждый кластер блокируется отдельно, и при этом используется `boost::shared_mutex`, который позволяет нескольким потокам одновременно читать кластер. Но нельзя ли добиться большего уровня параллелизма, еще уменьшив гранулярность блокировок? В следующем разделе мы покажем, как это сделать, воспользовавшись потокобезопасным списковым контейнером с поддержкой итераторов.

6.3.2. Потокобезопасный список с блокировками

Список — одна из самых простых структур данных, и, наверное, написать его потокобезопасную версию будет несложно, правда? Все зависит от того, какая вам нужна функциональность и требуется ли поддержка итераторов — та самая, которую я побоялся включать в справочную таблицу на том основании, что это слишком сложно. Основная идея итератора в духе STL состоит в том, что итератор содержит своего рода ссылку на элемент внутренней структуры данных, образующей контейнер. Если контейнер модифицируется из другого потока, то ссылка должна оставаться действительной, а это значит, что итератор должен удерживать блокировку на какую-то часть структуры. Учитывая, что контейнер никак не контролирует время жизни STL-итератора, такой подход абсолютно непродуктивен.

Альтернатива — включить функции итерирования, например `for_each`, в сам контейнер. Тогда контейнер сможет полностью управлять своим обходом и блокировкой, но зато перестаёт отвечать рекомендациям но предотвращению взаимоблокировок из главы 3. Чтобы функция `for_each` делала что-то полезное, она должна вызывать предоставленный пользователем код, удерживая блокировку. Хуже того, она должна передавать ссылку на каждый элемент контейнера пользовательскому коду, чтобы тот мог к этому элементу обратиться. Можно было бы вместо ссылки передавать копию элемента, но это обойдется слишком дорого, если размер элементов велик.

Таким образом, мы оставим на совести пользователя заботу о предотвращении взаимоблокировок, предупредив его, что в пользовательских функциях не следует ни захватывать блокировки, ни сохранять ссылки, которые могли бы использоваться для доступа к данным вне защиты блокировок и тем самым привести к гонке. В случае внутреннего списка, используемого в реализации справочной таблицы, это абсолютно безопасно, поскольку мы не собираемся делать ничего противозаконного.

Остается решить, какие операции должен поддерживать список. Вернувшись к листингам 6.11 и 6.12, вы увидите, что именно нам требуется:

• добавлять элемент в список;
• удалять элемент из списка, если он удовлетворяет некоторому условию;
• искать в списке элемент, удовлетворяющий некоторому условию;
• изменить элемент, удовлетворяющий некоторому условию;
• скопировать все элементы списка в другой контейнер.

Чтобы получился приличный списковый контейнер общего назначения, полезно было бы добавить еще кое-какие операции, например вставку в указанную позицию, но для нашей справочной таблицы это излишне, поэтому оставляю реализацию в качестве упражнения для читателя.

Основная идея связанного списка с мелкогранулярными блокировками — завести по одному мьютексу на каждый узел. Если список длинный, то получится целая куча мьютексов! Достоинство заключается в том, что операции над отдельными частями списка полностью распараллеливаются: каждая операция удерживает только блокировки на узлы, которыми манипулирует, и освобождает блокировку при переходе к следующему узлу. В листинге ниже приведена реализация такого списка.

Листинг 6.13. Потокобезопасный список с поддержкой итераторов

```
template<typename T>
class threadsafe_list {
 struct node { ← (1)
  std::mutex m;
  std::shared_ptr<T> data;
  std::unique_ptr<node> next;

  node() : ← (2)
   next() {}

 node(T const& value) : ← (3)
  data(std::make_shared<T>(value)) {}
 };

 node head;

public:
 threadsafe_list() {}
 ~threadsafe_list() {
  remove_if([](node const&){return true;});
 }

 threadsafe_list(threadsafe_list const& other) = delete;
 threadsafe_list& operator=(
  threadsafe_list const& other) = delete;

 void push_front(T const& value) {
  std::unique_ptr<node> new_node(new node(value));← (4)
  std::lock_guard<std::mutex> lk(head.m);
  new_node->next = std::move(head.next);          ← (5)
  head.next = std::move(new_node);                ← (6)
 }
```

```
 template<typename Function>
 void for_each(Function f) {                          ←(7)
  node* current = &head;
  std::unique_lock<std::mutex> lk(head.m);            ←(8)
  while(node* const next = current->next.get()) {     ←(9)
   std::unique_lock<std::mutex> next_lk(next->m);     ←(10)
   lk.unlock();                                       ←(11)
   f(*next->data);                                    ←(12)
   current = next;
   lk = std::move(next_lk);                           ←(13)
  }
 }

 template<typename Predicate>
 std::shared_ptr<T> find_first_if(Predicate p) {      ←(14)
  node* current = &head;
  std::unique_lock<std::mutex> lk(head.m);
  while (node* const next=current->next.get()) {
   std::unique_lock<std::mutex> next_lk(next->m);
   lk.unlock();
   if (p(*next->data)) {                              ←(15)
    return next->data;                                ←(16)
   }
   current = next;
   lk = std::move(next_lk);
  }
  return std::shared_ptr<T>();
 }

 template<typename Predicate>                         ←(17)
 void remove_if(Predicate p) {
  node* current = &head;
  std::unique_lock<std::mutex> lk(head.m);
  while(node* const next = current->next.get()) {
   std::unique_lock<std::mutex> next_lk(next->m);
   if (p(*next->data)) {                              ←(18)
    std::unique_ptr<node> old_next = std::move(current->next);
    current->next = std::move(next->next);            ←(19)
    next_lk.unlock();
   }                                                  ←(20)
   else {
    lk.unlock();                                      ←(21)
    current = next;
    lk = std::move(next_lk);
   }
  }
 }
};
```

Показанный в листинге 6.13 шаблон `threadsafe_list<>` — это реализация односвязного списка, в котором каждый элемент является структурой типа `node` **(1)**. В роли

головы head списка выступает сконструированный по умолчанию объект node, в котором указатель next равен NULL **(2)**. Новые узлы добавляются в список функцией push_front(); сначала новый узел конструируется **(4)**, при этом для хранимых в нем данных выделяется память из кучи **(3)**, но указатель next остается равным NULL. Затем мы должны захватить мьютекс для узла head, чтобы получить нужное значение указателя next **(5)**, после чего вставить узел в начало списка, записав в head.next указатель на новый узел **(6)**. Пока всё хорошо: для добавления элемента в список необходимо захватить только один мьютекс, поэтому никакого риска взаимоблокировки нет. Кроме того, медленная операция выделения памяти происходит вне блокировки, так что блокировка защищает только обновление двух указателей — действия, которые не могут привести к ошибке. Переходим к функциям итерирования.

Для начала рассмотрим функцию for_each() **(7)**. Она принимает объект Function, который применяется к каждому элементу списка; следуя примеру большинства библиотечных алгоритмов, этот объект передаётся по значению и может быть как настоящей функцией, так и объектом класса, в котором определена оператор вызова. В данном случае функция должна принимать в качестве единственного параметра значение типа T. Здесь мы производим передачу блокировки. Сначала захватывается мьютекс в головном узле head **(8)**. Теперь можно безопасно получить указатель на следующий узел next (с помощью get(), потому что мы не принимаем на себя владение указателем). Если этот указатель не равен NULL **(9)**, то захватываем мьютекс в соответствующем узле **(10)**, чтобы обработать данные. Получив блокировку для этого узла, мы можем освободить блокировку для предыдущего узла **(11)** и вызвать указанную функцию **(12)**. По выходе из этой функции мы можем обновить указатель current на только что обработанный узел и с помощью move передать владение блокировкой от next_lk в lk **(13)**. Поскольку for_each передаёт каждый элемент данных напрямую пользовательской функции Function, мы можем обновить данные, скопировать их в другой контейнер и вообще сделать всё, что угодно. Если функция не делает того, чего нельзя, то это безопасно, потому что на всем протяжении вызова удерживается мьютекс узла, содержащего элемент данных.

Функция find_first_if() **(14)** аналогична for_each(); существенное отличие заключается в том, что переданный предикат Predicate должен вернуть true, если нужный элемент найден, и false в противном случае **(15)**. Если элемент найден, то мы сразу возвращаем хранящиеся в нем данные **(16)**, прерывая поиск. Можно было бы добиться того же результата с помощью for_each(), но тогда мы продолжили бы просмотр списка до конца, хотя после обнаружения искомого элемента в этом уже нет необходимости.

Функция remove_if() **(17)** несколько отличается, потому что она должна изменить список; for_each() для этой цели непригодна. Если предикат Predicate возвращает true **(18)**, то мы удаляем узел из списка, изменяя значение current->next **(19)**. Покончив с этим, мы можем освободить удерживаемый мьютекс следующего узла. Узел удаляется, когда объект std::unique_ptr<node>, в который мы его переместили, покидает область видимости **(20)**. В данном случае мы не изменяем current, потому что необходимо проверить следующий узел next. Если Predicate возвращает false, то нужно просто продолжить обход списка, как и раньше **(21)**.

А могут ли при таком обилии мьютексов возникнуть взаимоблокировки или состояния гонки? Ответ — твердое *нет*, при условии, что полученные от пользователя предикаты и

функции ведут себя, как положено. Итерирование всегда производится в одном направлении, начиная с узла `head`, и следующий мьютекс неизменно блокируется до освобождения текущего, поэтому не может случиться так, что в разных потоках порядок захвата мьютексов будет различен. Единственный потенциальный кандидат на возникновение гонки — удаление исключенного из списка узла в функции `remove_if()` **(20)**, потому что это делается после освобождения мьютекса (уничтожение захваченного мьютекса приводит к неопределённому поведению). Однако, немного поразмыслив, мы придём к выводу, что это безопасно, так как в этот момент все еще удерживается мьютекс предыдущего узла (`current`), поэтому ни один другой поток не сможет попытаться захватить мьютекс удаляемого узла.

Что можно сказать по поводу распараллеливания? Вся эта возня с мелкогранулярными блокировками затевалась для того, чтобы увеличить уровень параллелизма по сравнению с единственным мьютексом. Так достигли мы своей цели или нет? Да, безусловно — теперь разные потоки могут одновременно работать с разными узлами списка, выполняя разные функции: `for_each()` для обработки каждого узла, `find_first_if()` для поиска или `remove_if()` для удаления элементов. Но, поскольку мьютексы узлов захватываются по порядку, потоки не могут обгонять друг друга. Если какой-то поток тратит много времени на обработку конкретного узла, то, дойдя до этого узла, остальные потоки вынуждены будут ждать.

6.4. Резюме

В начале этой главы мы обсудили, что понимается под проектированием структуры данных, допускающей распараллеливание, и сформулировали несколько рекомендаций. Затем на примере нескольких широко распространенных структур данных (стек, очередь, хеш-таблица и связанный список) мы видели, как эти рекомендации применяются на практике — обеспечивают параллельный доступ с применением блокировок и предотвращают гонки. Теперь вы можете проанализировать дизайн своих структур данных и подумать, где в нем есть возможности для распараллеливания, а где возможны состояния гонки.

В главе 7 мы узнаем, как можно, не отходя от сформулированных рекомендаций, вообще обойтись без блокировок, применяя для обеспечения необходимых ограничений на упорядочение низкоуровневые атомарные операции.

Глава 7
Проектирование параллельных структур данных без блокировок

В этой главе:

- Реализация параллельных структур данных без использования блокировок.
- Техника управления памятью в структурах данных без блокировок.
- Простые рекомендации по написанию структур данных без блокировок.

В предыдущей главе мы рассмотрели общие аспекты проектирования параллельных структур данных и сформулировали общие рекомендации, как удостовериться в том, что спроектированная структура безопасна. Затем мы изучили несколько распространенных структур данных и показали, как можно их реализовать с применением мьютексов и блокировок для защиты разделяемых данных. В первых двух примерах мы использовали один мьютекс для защиты всей структуры данных, а в последующих — несколько мьютексов, защищающих более мелкие части структуры, что обеспечило более высокий уровень параллелизма при доступе к данным.

Мьютексы — это мощный механизм, позволяющий нескольким потокам безопасно обращаться к структуре данных, не нарушая инвариантов и не порождая гонок. Рассуждать о поведении кода, в котором используются мьютексы, сравнительно просто: код либо захватывает защищающий данные мьютекс, либо нет. Но не всё коту масленица — в главе 3 мы видели, что некорректное использование мьютексов может стать причиной взаимоблокировок, а при рассмотрении очереди и справочной таблицы показали, как гранулярность блокировок может влиять на истинно параллельное выполнение программы. Если бы удалось разработать структуры данных, с которыми можно было бы безопасно работать, не прибегая к блокировкам, то эти проблемы вообще не возникали бы. Такие структуры называются структурами данных *без блокировок*, или *свободными от блокировок*.

В этой главе мы рассмотрим, как можно использовать свойства упорядочения доступа к памяти, присущие атомарным операциям (см. главу 5), для настроения структур данных без блокировок. При проектировании таких структур надо проявлять крайнюю осторожность, потому что реализовать их правильно трудно, а условия, при которых проявляются ошибки, могут возникать очень редко. Начнем с ответа на вопрос, что понимается под структурой данных без блокировок. Затем остановимся на причинах, но которым такие структуры полезны, и, наконец, проработаем ряд примеров и дадим некоторые общие рекомендации.

7.1. Определения и следствия из них

Алгоритмы и структуры данных, в которых для синхронизации доступа используются мьютексы, условные переменные и будущие результаты, называются *блокирующими*. Приложение вызывает библиотечные функции, которые приостанавливают выполнение потока до тех пор, пока другой поток не завершит некоторое действие. Такие библиотечные функции называются *блокирующими*, потому что поток не может продвинуться дальше некоторой точки, пока не будет снят блокировка. Обычно ОС полностью приостанавливает заблокированный поток (и отдает его временные кванты другому потоку) до тех пор, пока он не будет *разблокирован* в результате выполнения некоторого действия в другом потоке, будь то освобождение мьютекса, сигнал условной переменной или перевод будущего результата в состояние *готов*.

Структуры данных и алгоритмы, в которые блокирующие библиотечные функции не используются, называются *неблокирующими*. Но не все такие структуры данных *свободны от блокировок*, поэтому давайте сначала рассмотрим различные типы неблокирующих структур.

7.1.1. Типы неблокирующих структур данных

В главе 5 мы реализовали простой мьютекс-спинлок с помощью `std::atomic_flag`. Этот код воспроизведён в листинге ниже.

Листинг 7.1. Реализация мьютекса-спинлока с помощью `std::atomic_flag`

```
class spinlock_mutex {
 std::atomic_flag flag;
public:
 spinlock_mutex():
  flag(ATOMIC_FLAG_INIT) {}

 void lock() {
  while (flag.test_and_set(std::memory_order_acquire));
 }

 void unlock() {
  flag.clear(std::memory_order_release);
 }
};
```

Здесь не вызываются никакие блокирующие функции; `lock()` просто «крутится» в цикле, пока `test_and_set()` не вернет `false`. Отсюда и название *спинлок* (spin lock) — слово spin означает «крутиться». Как бы то ни было, блокирующих вызовов нет, и, значит, любая программа, в которой для защиты разделяемых данных используется такой мьютекс, будет *неблокирующей*. Однако она не *свободна от блокировок*. Это по-прежнему мьютекс, который в каждый момент времени может захватить только один поток. Теперь сформулируем определение *свободы от блокировок* и посмотрим, какие структуры данных под него подпадают.

7.1.2. Структуры данных, свободные от блокировок

Чтобы структура данных считалась свободной от блокировок, она должна быть открыта для одновременного доступа со стороны сразу нескольких потоков. Не требуется, чтобы потоки могли выполнять одну и ту же операцию; свободная от блокировок очередь может позволять одного потоку помещать, а другому — извлекать данные, но запрещать одновременное добавление данных со стороны двух потоков. Более того, если один из потоков, обращающихся к структуре данных, будет приостановлен планировщиком в середине операции, то остальные должны иметь возможность завершить операцию, не дожидаясь возобновления приостановленного потока.

Алгоритмы, в которых применяются операции сравнения с обменом, часто содержат циклы. Зачем вообще используются такие операции? Затем, что другой поток может в промежутке модифицировать данные, и тогда программа должна будет повторить часть операции, прежде чем попытается еще раз выполнить сравнение с обменом. Такой код может оставаться свободным от блокировок при условии, что сравнение с обменом в конце концов завершится успешно, если другие потоки будут приостановлены. Если это не так, то мы по существу получаем спинлок, то есть алгоритм неблокирующий, но не свободный от блокировок.

Свободные от блокировок алгоритмы с такими циклами могут приводить к *застреванию* (starvation) потоков, когда один поток, выполняющий операции с «неудачным» хронометражем, продвигается вперёд, а другой вынужден постоянно повторять одну и ту же операцию. Структуры данных, в которых такой проблемы не возникает, называются свободными от блокировок и ожидания.

7.1.3. Структуры данных, свободные от ожидания

Свободная от блокировок структура данных называется свободной от ожидания, если обладает дополнительным свойством: каждый обращающийся к ней поток может завершить свою работу за ограниченное количество шагов вне зависимости от поведения остальных потоков. Алгоритмы, в которых количество шагов может быть неограничено из-за конфликтов с другими потоками, не свободны от ожидания.

Корректно реализовать свободные от ожидания структуры данных чрезвычайно трудно. Чтобы гарантировать, что каждый поток сможет завершить свою работу за ограниченное количество шагов, необходимо убедиться, что каждая операция может быть выполнена за один проход и что шаги, выполняемые одним потоком, не приводят к ошибке в операциях, выполняемых другими потоками. В результате алгоритмы выполнения различных операций могут значительно усложниться. Учитывая, насколько трудно правильно реализовать структуру данных, свободную от блокировок и ожидания, нужно иметь весьма веские причины для того, чтобы взяться за это дело; требуется тщательно соотносить затраты с ожидаемым выигрышем. Поэтому обсудим, какие факторы влияют на это соотношение.

7.1.4. Плюсы и минусы структур данных, свободных от блокировок

Основная причина для использования структур данных, свободных от блокировок, — достижение максимального уровня параллелизма. В контейнерах с блокировками всегда есть возможность, что один поток будет приостановлен на время, пока другой не завершит

операцию, — в конце концов, основное назначение мьютекса в том и состоит, чтобы предотвратить одновременный доступ за счет взаимного исключения. В случае структуры данных, свободной от блокировок, *какой-то* поток продвигается вперёд на каждом шаге. Если же структура еще и свободна от ожидания, то вперёд продвигаются все потоки, вне зависимости от того, что в это время делают другие, — необходимости ждать не возникает. Это свойство весьма желательно, но труднодостижимо. На этом пути очень легко скатиться к спинлоку.

Вторая причина для использования структур данных, свободных от блокировок, — надежность. Если поток завершается, не освободив блокировку, то вся структура данных безвозвратно испорчена. Но если такое происходит с потоком во время операции над структурой данных, свободной от блокировок, то не теряется ничего, кроме данных самого потока; остальные потоки продолжают нормально работать.

Но у этой медали есть и оборотная сторона: если вы не можете запретить потокам одновременный доступ к структуре, то должны внимательно следить за соблюдением инвариантов или выбирать альтернативные инварианты, соблюдение которых можно гарантировать. Кроме того, следует обращать внимание на ограничения упорядочения, налагаемые на операции. Чтобы избежать неопределённого поведения вследствие гонки за данными, следует использовать для всех модификаций атомарные операции. Но и этого недостаточно — необходимо гарантировать, что изменения становятся видны другим потокам в правильном порядке. Все это означает, что написание потокобезопасных структур данных без использования блокировок гораздо сложнее, чем с блокировками.

Ввиду отсутствия блокировок невозможны и взаимоблокировки, однако вместо них появляется угроза активных блокировок. *Активная блокировка* (live lock) возникает, когда два потока одновременно пытаются изменить структуру данных, но каждый из них должен начинать свою операцию сначала из-за изменений, произведенных другим потоком. Таким образом, каждый поток беспрестанно повторяет попытки в цикле. Представьте себе двух людей, пытающихся разойтись в узком проходе. Войдя в него одновременно, они сталкиваются лбами, поэтому оба отступают назад и пробуют еще раз. И так будет повторяться до тех пор, пока кто-то не проскочит первым (по взаимному согласию, потому что оказался быстрее или просто благодаря удаче). Как и в этом простом примере, активные блокировки обычно существуют недолго, потому что зависят от случайных временных соотношений при планировании потоков. Поэтому они скорее «подъедают» производительность, чем вызывают долгосрочные проблемы, но остерегаться их все равно стоит. По определению программа, свободная от блокировок, не страдает от активных блокировок, потому что существует ограничение сверху на количество шагов, необходимых для выполнения операции. Зато и алгоритм, скорее всего, окажется сложнее альтернативного и может потребовать большего числа шагов даже в случае, когда никакой другой поток одновременно не обращается к структуре данных.

Это подводит нас к еще одному недостатку кода, свободного от блокировок и ожидания: хотя он позволяет лучше распараллелить операции над структурой данных и сократить время ожидания в каждом конкретном потоке, общая производительность программы вполне может *упасть*. Во-первых, атомарные операции, используемые в свободном от блокировок коде, часто выполняются гораздо медленнее, чем неатомарные, а в структуре данных без блокировок их, скорее всего, будет гораздо больше, чем в аналогичной структуре с блокировками на основе мьютексов. К тому же, оборудование должно как-то

синхронизировать данные между потоками, которые обращаются к одним и тем же атомарным переменным. В главе 8 мы увидим, что эффект перебрасывания кэша, возникающий из-за того, что несколько потоков обращаются к одним и тем же атомарным переменным, может привести к существенному падению производительности. Как обычно, необходимо тщательно анализировать аспекты, связанные с производительностью (время ожидания в худшем случае, среднее время ожидания, полное время выполнения и т.д.), для обоих решений — с блокировками и без, — прежде чем остановиться на каком-то одном.

А теперь перейдём к примерам.

7.2. Примеры структур данных, свободных от блокировок

Для демонстрации некоторых приёмов проектирования структур данных, свободных от блокировок, мы рассмотрим реализации ряда простых структур.

Как уже отмечалось, структуры данных, свободные от блокировок, опираются на использование атомарных операций и связанные с ними гарантии упорядочения доступа к памяти, благодаря которым можно быть уверенным, что изменения данных становятся видны потокам в правильном порядке. Сначала мы будем использовать во всех атомарных операциях принимаемое по умолчанию упорядочение `memory_order_seq_cst`, потому что оно проще всего для понимания (напомним, что семантика `memory_order_seq_cst` устанавливает полное упорядочение всех операций). Но позже мы посмотрим, как ослабить некоторые ограничения с помощью семантик `memory_order_acquire`, `memory_order_release` и даже `memory_order_relaxed`. Хотя ни в одном примере мьютексы не используются напрямую, не стоит забывать, что отсутствие блокировок гарантируется только для типа `std::atomic_flag`. На некоторых платформах в казалось бы свободном от блокировок коде могут использоваться внутренние блокировки, скрытые в реализации стандартной библиотеки C++ (детали см. в главе 5). В этом случае простая структура данных с блокировками может оказаться предпочтительнее, но дело не только в этом; прежде чем выбирать ту или иную реализацию, нужно четко сформулировать требования, а затем подвергнуть профилированию различные решения, удовлетворяющие этим требованиям.

Итак, снова начнем с простейшей структуры данных — стека.

7.2.1. Потокобезопасный стек без блокировок

Основное свойство стека понятно: элементы извлекаются в порядке, обратном тому, в котором помещались — последним пришёл, первым ушел (LIFO). Поэтому важно убедиться, что после добавления значения в стек оно может быть сразу же безопасно извлечено другим потоком и что только один поток получает данное значение. Простейшая реализация стека основана на связанном списке; указатель `head` направлен на первый узел (который будет извлечен следующим), и каждый узел указывает на следующий в списке. При такой схеме добавление узла реализуется просто.

1. Создать новый узел.
2. Записать в его указатель `next` текущее значение `head`.
3. Записать в `head` указатель на новый узел.

Все это прекрасно работает в однопоточной программе, но, когда стек могут модифицировать сразу несколько потоков, этого недостаточно. Существенно, что если узлы добавляют два потока, то между шагами 2 и 3 возможна гонка: второй поток может модифицировать значение `head` после того, как первый прочитает его на шаге 2, но до изменения на шаге 3. В таком случае изменения, произведенные вторым потоком, будут отброшены или случится еще что-нибудь похуже. Прежде чем решать эту проблему, следует отметить, что после того, как указатель `head` будет изменен и станет указывать на новый узел, этот узел может быть прочитан другим потоком. Поэтому крайне важно, чтобы новый узел был аккуратно подготовлен *до того*, как на него начнет указывать `head`; потом изменять узел уже нельзя.

Ну хорошо, а как все-таки быть с этим неприятным состоянием гонки? Ответ таков — использовать атомарную операцию сравнить-и-обменять на шаге 3, гарантирующую, что `head` не был модифицирован с момента чтения на шаге 2. Если был, то следует вернуться в начало цикла и повторить. В листинге ниже показано, как можно реализовать потокобезопасную функцию `push()` без блокировок.

Листинг 7.2. Реализация функции `push()` без блокировок

```
template<typename T>
class lock_free_stack {
private:
 struct node {
  T data;
  node* next;
  node(T const& data_) : ← (1)
   data(data_) {}
 };

 std::atomic<node*> head;

public:
 void push(T const& data) {
  node* const new_node = new node(data);← (2)
  new_node->next = head.load();          ← (3)
  while (!head.compare_exchange_weak(
          new_node->next, new_node));    ← (4)
 }
};
```

В этом коде дотошно реализованы все три пункта изложенного выше плана: создать новый узел **(2)**, записать в его поле `next` текущее значение `head` **(3)** и записать в `head` указатель на новый узел **(4)**. Заполнив данные самой структуры `node` в конструкторе **(1)**, мы гарантируем, что узел готов к использованию сразу после конструирования, так что легкая проблема решена. Затем мы вызываем функцию `compare_exchange_weak()`, которая проверяет, что указатель `head` по-прежнему содержит то значение, которое было сохранено в `new_node->next` **(3)**, и, если это так, то записывает его в `new_node`. В этой части программы используется также полезное свойство сравнения с обменом: если функция возвращает `false`, означающее, что сравнение не прошло (например, потому что значение `head` было изменено другим потоком), то в переменную, которая передана в первом параметре (`new_node->next`) записывается текущее значение `head`. Поэтому нам не нужно перезагружать `head` на каждой итерации цикла — это сделает за нас компилятор. Кроме того, поскольку мы сразу переходим в начало цикла в случае неудачного сравнения, можно использовать функцию `compare_exchange_weak`, которая в некоторых архитектурах дает более оптимальный код, чем `compare_exchange_strong` (см. главу 5).

Итак, операции `pop()` у нас пока еще нет, но уже можно сверить реализацию `push()` с рекомендациями. Единственное место, где возможны исключения, — конструирование нового узла **(1)**, но здесь все будет подчищено автоматически, и, поскольку список еще не модифицирован, то опасности нет. Поскольку мы сами строим данные, сохраняемые в узле `node`, и используем `compare_exchange_weak()` для обновления указателя `head`, то проблематичных состояний гонки здесь нет. Если операция сравнения с обменом

завершилась успешно, то узел находится в списке, и его можно извлекать. Так как нет никаких блокировок, то нет и возможности взаимоблокировки, и, стало быть, функция `push()` успешно сдала экзамен.

Теперь, когда у нас есть средства для добавления данных в стек, надо научиться их извлекать обратно. На первый взгляд, тут всё просто.

1. Прочитать текущее значение `head`.
2. Прочитать `head->next`.
3. Записать в `head` значение `head->next`.
4. Вернуть поле `data`, хранящееся в извлеченном узле `node`.
5. Удалить извлеченный узел.

Однако наличие нескольких потоков осложняет дело. Если два потока пытаются удалить элементы из стека, то оба могут прочитать одно и то же значение `head` на шаге 1. Если затем один поток успеет выполнить все операции вплоть до шага 5, прежде чем другой доберется до шага 2, то второй поток попробует разыменовать висячий указатель. Это одна из самых серьезных проблем при написании кода, свободного от блокировок, поэтому пока мы просто опустим шаг 5, смирившись с утечкой узлов.

Однако на этом трудности не кончаются. Есть еще одна проблема: если два потока прочитают одно и то же значение `head`, то они вернут один и тот же узел. Это вступает в противоречие с самой идеей стека, поэтому должно быть предотвращено любой ценой. Решить проблему можно так же, как мы устранили гонку в `push()`: использовать для обновления `head` операцию сравнения с обменом. Если она завершается с ошибкой, значит, либо в промежутке был добавлен новый узел, либо другой поток только что извлек узел, который собирались извлечь мы. В любом случае нужно вернуться на шаг 1 (хотя операция сравнения с обменом автоматически перечитывает `head`).

Если сравнение с обменом завершилось успешно, то мы точно знаем, что больше ни один поток не пытался удалить данный узел из стека, поэтому можем без опаски выполнить шаг 4. Вот первая попытка написать код `pop()`:

```
template<typename T>
class lock_free_stack {
public:
 void pop(T& result) {
  node* old_head = head.load();
  while (!head.compare_exchange_weak(old_head, old_head->next));
  result = old_head->data;
 }
};
```

Вроде бы всё красиво и лаконично, но, помимо утечки узлов, осталось еще две проблемы. Во-первых, этот код не работает для пустого списка: если указатель `head` нулевой, то при попытке прочитать `next` мы получим неопределённое поведение. Это легко исправить, сравнивая в цикле `while` значение `head` с `nullptr`: если стек оказался пуст, мы можем либо возбудить исключение, либо вернуть булевский индикатор успеха или ошибки.

Вторая проблема касается безопасности относительно исключений. Впервые подступаясь к потокобезопасному стеку в главе 3, мы видели, что простой возврат объекта по значению небезопасен относительно исключений: если исключение возникает во время копирования возвращаемого значения, то значение будет потеряно. Тогда передача ссылки на результат оказалась приемлемым решением, которое гарантировало неизменность стека в случае исключения. К сожалению, сейчас мы лишены такой роскоши; безопасно скопировать

данные можно только тогда, когда мы точно знаем, что больше никакой поток не пытается вернуть данный узел, а *это означает, что узел уже удален из стека*. Следовательно, передача возвращаемого значения по ссылке больше не является преимуществом, с тем же успехом можно было бы вернуть его и по значению. Чтобы безопасно вернуть значение, придется воспользоваться другим вариантом, описанным в главе 3: возвращать интеллектуальный указатель на данные.

Возврат `nullptr` в качестве значения интеллектуального указателя будет означать, что данных в стеке нет, но беда в том, что теперь приходится выделять память из кучи. Если делать это в `pop()`, то получится, что мы ровным счетом ничего не выиграли, потому что выделение памяти может возбудить исключение. Вместо этого мы будем выделять память в `push()`, при помещении данных в стек — память-то для структуры `node` выделять приходится в любом случае. Возврат `std::shared_ptr<>` не возбуждает исключений, поэтому `pop()` теперь безопасна. Собрав все вместе, мы получим код, показанный в следующем листинге.

Листинг 7.3. Свободный от блокировок стек с утечкой узлов

```
template<typename T>
class lock_free_stack {
private:
 struct node                          (1) Теперь данные
 {                                     | удерживаются
  std::shared_ptr<T> data;←┘           указателем
  node* next;

  node(T const& data_) :                    (2) Создаем std::shared_ptr
   data(std::make_shared<T>(data))←┤ Для только что выде-
   {}                               | ленного T
 };

 std::atomic<node*> head;

public:
 void push(T const& data) {
  node* const new_node = new node(data);
  new_node->next = head.load();
  while (!head.compare_exchange_weak(new_node->next, new_node));
 }

 std::shared_ptr<T> pop()
 {                                      (3) Перед разыменованием
  node* old_head = head.load();|проверяем, что old_head —
  while (old_head &&                ←┘ ненулевой указатель
   !head.compare_exchange_weak(old_head, old_head->next));
  return old_head ? old_head->data : std::shared_ptr<T>();← (4)
 }
};
```

Теперь данные удерживаются указателем **(1)**, поэтому мы должны выделять память для них из кучи в конструкторе узле **(2)**. Кроме того, перед тем как разыменовывать `old_head` в цикле `compare_exchange_weak()` **(3)**, следует проверять указатель на нуль. Наконец, мы либо

возвращаем ассоциированные с узлом данные, если узел имеется, либо нулевой указатель, если его нет **(4)**. Отметим, что этот алгоритм *свободен от блокировок*, но не *свободен от ожидания*, потому что цикл `while` в `push()` и `pop()` теоретически может выполняться бесконечно, если `compare_exchange_weak()` будет каждый раз возвращать `false`.

Если бы у нас был сборщик мусора (как в управляемых языках типа C# или Java), то на этом можно было бы ставить точку — старый узел был бы убран и повторно использован после того, как все потоки перестанут к нему обращаться. Но сегодня мало найдётся компиляторов C++ с встроенным сборщиком мусора, поэтому прибираться за собой нужно самостоятельно.

7.2.2. Устранение утечек: управление памятью в структурах данных без блокировок

При первом подходе к `pop()` мы решили смириться с утечкой узлов, чтобы избежать гонки в случае, когда один поток удаляет узел, а другой в это время хранит указатель на него, который собирается разыменовать. Однако в корректной программе на C++ утечка памяти, конечно, недопустима, так что с этим надо что-то делать. Сейчас мы покажем, как эта проблема решается.

Основная трудность состоит в том, что мы хотим освободить занятую узлом память, но не можем сделать это, пока нет уверенности, что никакой другой поток не хранит указателей на нее. Если бы в каждый момент времени только один поток вызывал `pop()` для данного экземпляра стека, то все было бы прекрасно. Функция `push()` не обращается к уже добавленному в стек узлу, так что кроме потока, вызвавшего `pop()`, этот узел больше никого не интересует, и его можно безопасно удалить.

С другой стороны, если мы все-таки хотим, чтобы несколько потоков могли одновременно вызывать `pop()`, то нужно каким-то образом отслеживать момент, когда удаление узла становится безопасным. По сути дела, это означает, что необходимо написать специализированный сборщик мусора для узлов `node`. Звучит пугающе, и эта задача действительно не самая простая, но на практике все не так плохо: мы проверяем только указатели на `node` и только те узлы, к которым обращается `pop()`. Нас не интересуют узлы внутри `push()`, потому что они доступны только одному потоку, пока не окажутся в стеке. А вот внутри `pop()` к одному и тому же узлу могут одновременно иметь доступ несколько потоков.

Если потоков, вызывающих `pop()`, нет вообще, то можно без опаски удалить все узлы, ожидающие удаления. Поэтому, если после извлечения данных помещать узлы в список «подлежат удалению», то их можно будет удалить одним махом в момент, когда не будет потоков, вызывающих `pop()`. Но как узнать, что потоков, вызывающих `pop()`, действительно нет? Очень просто — подсчитывать их. Если увеличивать счетчик при входе и уменьшать при выходе, то удалять узлы из списка «подлежащих удалению» можно будет, когда счетчик становится равным нулю. Разумеется, сам счетчик должен быть атомарным, чтобы к нему можно было безопасно обращаться из нескольких потоков. В листинге 7.4 показала исправленная функция `pop()`, а в листинге 7.5 — вспомогательные функции, используемые в ее реализации.

Листинг 7.4. Освобождение занятой узлами памяти в момент, когда нет потоков, вызывающих `pop()`

```
template<typename T>
class lock_free_stack {
private:
 std::atomic<unsigned> threads_in_pop;←┐ Атомарная
 void try_reclaim(node* old_head);     (1) переменная

public:
 std::shared_ptr<T> pop()
 {                                  (2) Увеличить счетчик
  ++threads_in_pop;                 ←┤ перед тем, как что-то
  node* old_head = head.load();     | делать
  while (old_head &&
   !head.compare_exchange_weak(old_head, old_head->next));
  std::shared_ptr<T> res;
  if (old_head)
  {                                 (3) Не копировать
   res.swap(old_head->data);←┤ указатель, а извлечь
  }                                 | данные из узла
  try_reclaim(old_head);←┐ Освободить удаленные
  return res;           (4) узлы, если получится
 }
};
```

Атомарная переменная `threads_in_pop` **(1)** нужна для подсчета потоков, которые в данный момент пытаются извлечь элемент из стека. Она увеличивается на единицу в начале `pop()` **(2)** и уменьшается на единицу внутри функции `try_reclaim()`, которая вызывается после изъятия узла из списка **(4)**. Поскольку мы откладываем удаление самого узла, то можно с помощью `swap()` переместить из него данные **(3)**, а не просто скопировать указатель; тогда данные будут автоматически удалены, когда в них отпадает необходимость, вместо того, чтобы занимать память только потому, что на них ссылается еще не удаленный узел. В следующем листинге показан код функции `try_reclaim()`.

Листинг 7.5. Механизм освобождения памяти на основе подсчёта ссылок

```
template<typename T>
class lock_free_stack {
private:
 std::atomic<node*> to_be_deleted;

 static void delete_nodes(node* nodes) {
  while (nodes) {
   node* next = nodes->next;
   delete nodes;
   nodes = next;
  }
 }

 void try_reclaim(node* old_head) {
```

```
 if (threads_in_pop == 1) ← (1)
 {          Заявляем права на список подлежащих удалению узлов (2)
  node* nodes_to_delete = to_be_deleted.exchange(nullptr);←┘
  if (!--threads_in_pop)←┐ Я — единственный
  {                      (3) поток в pop()?
   delete_nodes(nodes_to_delete);        ← (4)
  } else if(nodes_to_delete) {           ← (5)
   chain_pending_nodes(nodes_to_delete);← (6)
  }
  delete old_head; ← (7)
 } else {
  chain_pending_node(old_head); ← (8)
  --threads_in_pop;
 }
}

void chain_pending_nodes(node* nodes) {
 node* last = nodes;
 while (node* const next =
        last->next) {←┐  По указателям
                      ├ (9) next доходим до
  last = next;        │   конца списка
 }
 chain_pending_nodes(nodes, last);
}

void chain_pending_nodes(node* first, node* last) {
 last->next = to_be_deleted;← (10)
 while (
  !to_be_deleted.compare_exchange_weak(←┐   цикл гарантиру-
  last->next, first));                  ├ (11)ет, что last->next
 }                                      │   корректно

void chain_pending_node(node* n) {
 chain_pending_nodes(n, n);← (12)
}
};
```

Если при попытке освободить занятую узлом **(1)** память счетчик `threads_in_pop` оказывается равен 1, то данный поток — единственный в `pop()`, и, значит, можно безопасно удалять только что исключенный из списка узел **(7)** и, *быть может*, также узлы, ожидающие удаления. Если счетчик *не* равен 1, то никакие узлы удалять нельзя, поэтому мы просто добавляем узел в список ожидающих **(8)**.

Предположим, что `threads_in_pop` равно 1. Тогда нам нужно освободить ожидающие узлы, потому что если этого не сделать, то они так и будут ожидать удаления до момента уничтожения стека. Для этого мы запрашиваем монопольные права на список с помощью атомарной операции `exchange` **(2)**, после чего уменьшаем на единицу счетчик `threads_in_pop` **(3)**. Если в результате счетчик оказался равным нулю, значит, больше ни один поток не работает со списком ожидающих удаления узлов. По ходу дела могли

появиться новые ожидающие узлы, но сейчас — когда можно безопасно очистить список — нам это безразлично. Мы просто вызываем функцию `delete_nodes`, которая обходит список и удаляет узлы **(4)**.

Если счетчик после уменьшения *не* равен нулю, то освобождать узлы небезопасно, поэтому если такие узлы есть **(5)**, то нужно поместить их в начало списка ожидающих **(6)**. Такое может случиться, если к структуре данных одновременно обращаются несколько потоков. Другие потоки могли вызвать `pop()` в промежутке между первой проверкой `threads_in_pop` **(1)** и «заявлением прав» на список **(2)** и добавить в список узлы, к которым все еще обращается один или несколько потоков. На рис. 7.1 поток C добавляет узел Y в список `to_be_deleted`, несмотря на то, что поток B все еще ссылается на него по указателю `old_head` и, значит, будет пробовать читать его указатель `next`. Поэтому поток A не может удалить узлы без риска вызвать неопределенное поведение в потоке B.

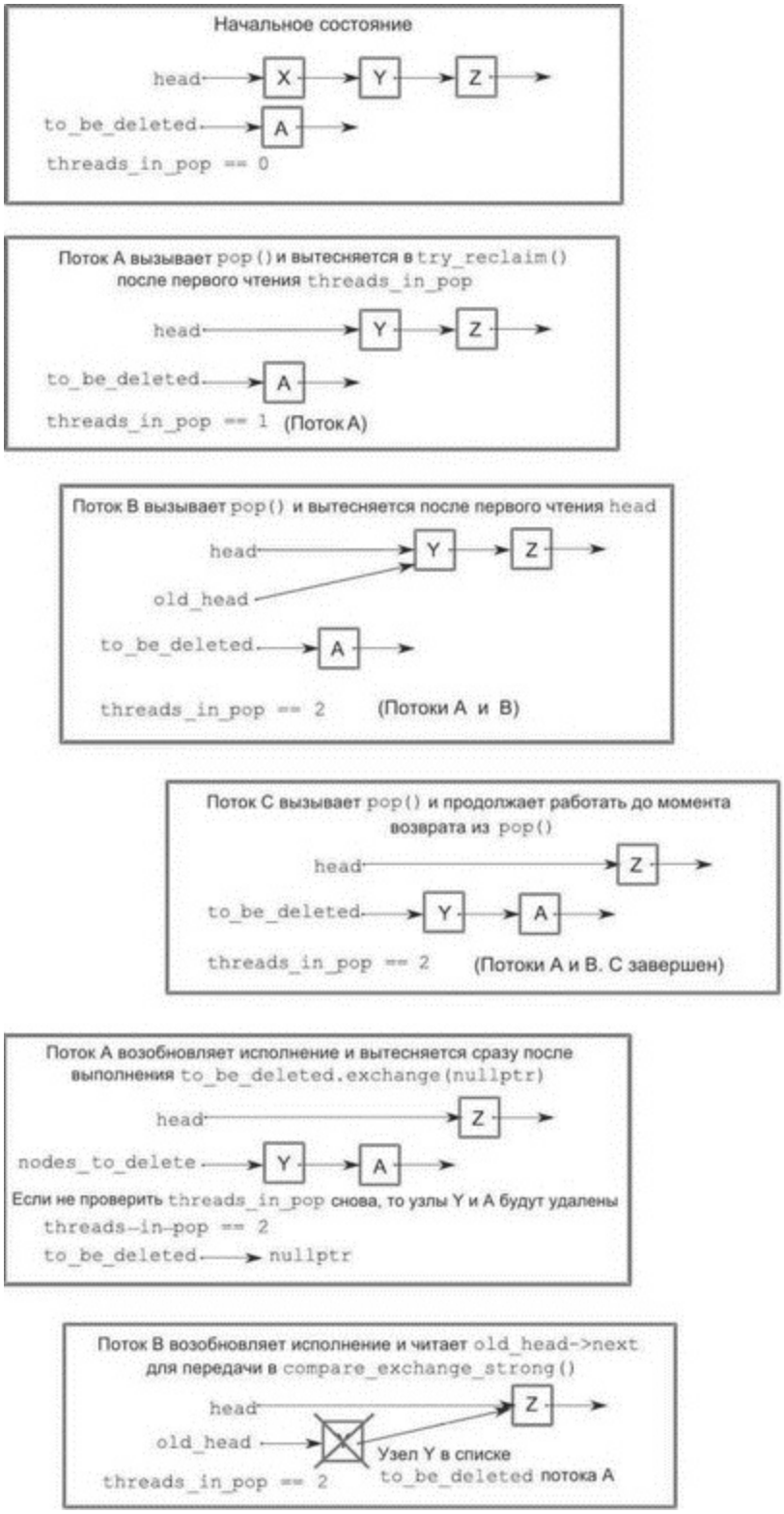

Рис. 7.1. На примере трех потоков, вызывающих `pop()`, видно, почему необходимо проверять `threads_in_pop` после заявления прав на список узлов, ожидающих удаления, в `try_reclaim()`

Чтобы поместить узлы, ожидающие удаления, в список ожидающих, мы используем уже имеющийся указатель `next` для связывания. Для того чтобы добавить цепочку в список, мы проходим до конца цепочки **(9)**, записываем в указатель `next` в последнем узле текущее значение `to_be_deleted` **(10)** и сохраняем указатель на первый узел цепочки как новый указатель `to_be_deleted` **(11)**. Здесь необходимо вызывать `compare_exchange_weak` в цикле, чтобы не допустить утечки узлов, добавленных другим потоком. В результате в `next` записывается указатель на последний узел цепочки, если он изменился. Добавление единственного узла в список — это особый случай, когда первый узел в добавляемой цепочке

совпадает с последним **(12)**.

Этот алгоритм работает вполне приемлемо, если нагрузка невелика, то есть существуют моменты *затишья*, когда в `pop()` нет ни одного потока. Но эта ситуация кратковременна; именно поэтому мы должны проверять, что счетчик `threads_in_pop` после уменьшения обратился в нуль **(3)**, прежде чем освобождать память, и по той же причине проверка стоит *до* удаления только что изъятого из стека узла **(7)**. Удаление узла может занять относительно много времени, а мы хотим, чтобы окно, в котором другие потоки могут модифицировать список, было как можно короче. Чем больше времени проходит между моментом, когда поток впервые обнаружил, что `threads_in_pop` равно 1, и попыткой удалить узлы, тем больше шансов, что какой-то другой поток вызовет `pop()`, после чего `threads_in_pop` перестанет быть равно 1 и, стало быть, удалять узлы станет нельзя.

Если нагрузка высока, то затишье может не наступить *никогда*, поскольку новые потоки входят в `pop()` до того, как пребывавшие там успевают выйти. В таком случае список `to_be_deleted` будет расти неограниченно, и мы снова сталкиваемся с утечкой памяти. Если периодов затишья не ожидается, то необходим другой механизм освобождения узлов. Главное здесь — определить, когда ни один поток больше не обращается к конкретному узлу, который, следовательно, можно освободить. Из всех возможных механизмов такого рода для понимания проще всего тот, в котором используются *указатели опасности* (hazard pointers).

7.2.3. Обнаружение узлов, не подлежащих освобождению, с помощью указателей опасности

Термин *указатели опасности* откосится к технике, предложенной Магедом Майклом (Maged Michael)[12]. Они называются так потому, что удаление узла, на который все еще могут ссылаться другие потоки, — опасное предприятие. Если действительно существует поток, ссылающийся на данный узел, и этот поток попытается обратиться к нему по ссылке, то мы получим неопределенное поведение. Основная идея заключается в том, что поток, собирающийся получить доступ к объекту, который другой поток может захотеть удалить, сначала устанавливает указатель опасности, ссылающийся на этот объект, информируя тем самым другой поток, что удалять этот объект действительно опасно. После того как объект перестает быть нужным, указатель опасности очищается. Если вам доводилось наблюдать гребную регату между командами Оксфорда и Кембриджа, то вы могли видеть аналогичный механизм, применяемый в начале заезда: рулевой в лодке может поднять руку, сообщая, что экипаж еще не готов. Пока хотя бы один рулевой держит руку поднятой, судья не может дать старт заезду. После того как оба рулевых опустят руки, судья может давать старт, однако рулевой вправе снова поднять руку, если заезд еще не начался, а ситуация, на его взгляд, изменилась.

Собираясь удалить объект, поток должен сначала проверить указатели опасности в других имеющихся в системе потоках. Если ни один из указателей опасности не ссылается на данный объект, то его можно спокойно удалять. В противном случае удаление следует отложить. Периодически поток просматривает список отложенных объектов в поисках тех, которые уже можно удалить.

Высокоуровневое описание выглядит достаточно простым, но как это сделать на C++?

Прежде всего, необходимо место для хранения указателя на интересующий нас объект

— сам *указатель опасности*. Это место должно быть видно из всех потоков, причем указатель опасности должен существовать в каждом потоке, который может получить доступ к структуре данных. Корректное и эффективное выделение такого места — непростая задача, поэтому отложим ее на потом, а пока предположим, что существует функция `get_hazard_pointer_for_current_thread()`, которая возвращает ссылку на указатель опасности. Затем нужно установить указатель опасности перед чтением указателя, который мы намерены разыменовать, — в данном случае указателя `head` на начало списка:

```
std::shared_ptr<T> pop() {
 std::atomic<void*>& hp =
  get_hazard_pointer_for_current_thread();
 node* old_head = head.load();←(1)
 node* temp;
 do {
  temp = old_head;
  hp.store(old_head);          ←(2)
  old_head = head.load();
 } while (old_head != temp);  ←(3)
 // ...
}
```

Это необходимо делать в цикле `while`, чтобы узел `node` случайно не был удалён между чтением старого указателя `head` **(1)** и установкой указателя опасности **(2)**. В течение этого промежутка времени ни один поток не знает, что мы собираемся обратиться к этому узлу. К счастью, если кто-то собирается удалить старый узел `head`, то сам указатель `head` должен был быть изменен, так что мы можем это проверить и не выходить из цикла, пока не будем твердо уверены, что указатель `head` по-прежнему имеет то же значение, которое было записано в указатель опасности **(3)**. Такое использование указателей опасности опирается на тот факт, что можно безопасно использовать значение указателя даже после того, как объект, на который он указывает, уже удалён. Технически для стандартных реализаций `new` и `delete` это считается неопределенным поведением, поэтому либо убедитесь, что ваша реализация стандартной библиотеки допускает такое использование, либо реализуйте собственный распределитель.

Установив указатель опасности, мы можем продолжить выполнение `pop()`, будучи уверены, что ни один другой поток не попытается «вытащить» из-под нас узлы. Ну почти уверены: при каждом перечитывании `old_head` необходимо обновлять указатель опасности перед тем, как разыменовывать вновь прочитанное значение указателя. После того как узел извлечён из списка, мы можем очистить наш собственный указатель опасности. Если на наш узел не ссылаются другие указатели опасности, то его можно удалять; в противном случае его следует поместить в список узлов, ожидающих удаления. В листинге ниже приведен полный код функции `pop()`, реализованной по такой схеме.

Листинг 7.6. Реализация функции `pop()` с помощью указателей опасности

```
std::shared_ptr<T> pop() {
 std::atomic<void*>& hp =
  get_hazard_pointer_for_current_thread();
 node* old_head = head.load();
 do {
  node* temp;  (1) Цикл, пока указатель
  do           ←┤ опасности не установлен
```

```
  {                                                    | на head
   temp = old_head;
   hp.store(old_head);
   old_head = head.load();
  } while (old_head != temp);
 }
 while (old_head &&
  !head.compare_exchange_strong(old_head, old_head->next))
  hp.store(nullptr);← (2) Закончив, очищаем указатель опасности
 std::shared_ptr<T> res;
 if (old_head) {
  res.swap(old_head->data);
  if (outstanding_hazard_pointers_for(old_head))←┐ Прежде чем
  {                                              ├ (3) удалять узел,
   reclaim_later(old_head);                      | проверяем, ← (4)
  }                                              | нет ли ссы-
  else                                           | лающихся на
  {                                              | него указате-
                                                 | лей опасности
   delete old_head; ← (5)
  }
  delete_nodes_with_no_hazards();← (6)
 }
 return res;
}
```

Начнём с того, что мы перенесли цикл, в котором устанавливается указатель опасности, во внешний цикл, где перечитывается `old_head`, если операция сравнения с обменом завершается неудачно **(1)**. Здесь мы используем функцию `compare_exchange_strong()`, потому что фактическая работа делается внутри цикла `while`: ложный отказ в `compare_exchange_weak()` привел бы к ненужному сбросу указателя опасности. Таким образом, гарантируется, что указатель опасности установлен перед разыменованием `old_head`. Заявив свои права на узел, мы можем очистить указатель опасности **(2)**. Получив узел в свое распоряжение, мы должны проверить, не ссылаются ли на него указатели опасности, принадлежащие другим потокам **(3)**. Если это так, то удалять узел пока нельзя, а нужно поместить его в список ожидающих **(4)**; в противном случае узел можно удалять немедленно **(5)**. Наконец, мы добавили вызов функции, в которой проверяется, существуют ли узлы, для которых мы ранее вызывали `reclaim_later()`. Если не осталось указателей опасности, ссылающихся на эти узлы, то мы можем спокойно удалить их **(6)**. Те же узлы, на которые еще ссылается хотя бы один указатель опасности, остаются в списке и будут проверены следующим потоком, вызвавшим `pop()`.

Разумеется, в новых функциях — `get_hazard_pointer_for_current_thread()`, `reclaim_later()`, `outstanding_hazard_pointers_for()` и `delete_nodes_with_no_hazards()` — скрыта масса деталей, поэтому отдёрнем занавес и посмотрим, как они работают.

Как именно в функции `get_hazard_pointer_for_current_thread()` выделяется память для принадлежащих потокам указателей опасности, несущественно для логики программы (хотя, как будет показано ниже, может влиять на эффективность). Поэтому пока

ограничимся простой структурой: массивом фиксированного размера, в котором хранятся пары (идентификатор потока, указатель). Функция `get_hazard_pointer_for_current_thread()` ищет в этом массиве первую свободную позицию и записывает в поле ID идентификатор текущего потока. Когда поток завершается, эта позиция освобождается — в поле ID заносится сконструированное по умолчанию значение `std::thread::id()`. Этот алгоритм показан в следующем листинге.

Листинг 7.7. Простая реализация функции `get_hazard_pointer_for_current_thread`

```
unsigned const max_hazard_pointers = 100;
struct hazard_pointer {
 std::atomic<std::thread::id> id;
 std::atomic<void*> pointer;
};

hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner {
 hazard_pointer* hp;
public:
 hp_owner(hp_owner const&) = delete;
 hp_owner operator=(hp_owner const&) = delete;
 hp_owner() :
  hp(nullptr) {
  for (unsigned i = 0; i < max_hazard_pointers; ++i) {
   std::thread::id old_id;
   if (
    hazard_pointers[i].
    id.compare_exchange_strong(          ←┐
     old_id, std::this_thread::get_id())) | Пытаемся заявить
   {                                      | права на владе-
    hp = &hazard_pointers[i];             | ние указателем
    break;                                | опасности
   }
  }
  if (!hp) {← (1)
   throw std::runtime_error("No hazard pointers available");
  }
 }

 std::atomic<void*>& get_pointer() {
  return hp->pointer;
 }

 ~hp_owner() {← (2)
  hp->pointer.store(nullptr);
  hp->id.store(std::thread::id());
 }
};

std::atomic<void*>& get_hazard_pointer_for_current_thread()← (3)
{                                          (4) У каждого потока
```

```
 thread_local static hp_owner hazard;↵ свой указатель опасности
 return hazard.get_pointer();← (5)
}
```

Реализация самой функции `get_hazard_pointer_for_current_thread()` обманчиво проста **(3)**: в ней объявлена переменная типа `hp_owner` в поточно-локальной памяти **(4)**, в которой хранится принадлежащий данному потоку указатель опасности. Затем она просто возвращает полученный от этого объекта указатель **(5)**. Работает это следующим образом: в первый раз, когда *каждый поток* вызывает эту функцию, создается новый экземпляр `hp_owner`. Его конструктор **(1)** ищет в таблице пар (владелец, указатель) незанятую запись (такую, у которой нет владельца). На каждой итерации цикла он с помощью `compare_exchange_strong()` атомарно выполняет два действия: проверяет, что у текущей записи нет владельца, и делает владельцем себя **(2)**. Если `compare_exchange_strong()` возвращает `false`, значит, записью владеет другой поток, поэтому мы идем дальше. Если же функция вернула `true`, то мы успешно зарезервировали запись для текущего потока, поэтому можем сохранить ее адрес и прекратить поиск **(3)**. Если мы дошли до конца списка и не обнаружили свободной записи **(4)**, значит, потоков, использующих указатель опасности, слишком много, так что приходится возбуждать исключение.

После того как экземпляр `hp_owner` для данного потока создан, последующие обращения происходят гораздо быстрее, потому что указатель запомнен и просматривать таблицу снова нет нужды.

Когда завершается поток, для которого был создан объект `hp_owner`, этот объект уничтожается. Прежде чем сохранить в идентификаторе владельца значение `std::thread::id()`, деструктор записывает в сам указатель значение `nullptr`, чтобы другие потоки могли повторно использовать эту запись. При такой реализации `get_hazard_pointer_for_current_thread()` реализация функции `outstanding_hazard_pointers_for()` совсем проста: требуется только найти переданное значение в таблице указателей опасности:

```
bool outstanding_hazard_pointers_for(void* p) {
 for (unsigned i = 0; i < max_hazard_pointers; ++i) {
  if (hazard_pointers[i].pointer.load() == p) {
   return true;
  }
 }
 return false;
}
```

Не нужно даже проверять, есть ли у записи владелец, так как в бесхозных записях все равно хранятся нулевые указатели, поэтому сравнение заведомо вернёт `false`; это еще упрощает код. Теперь функции `reclaim_later()` и `delete_nodes_with_no_hazards()` могут работать с простым связанным списком; `reclaim_later()` добавляет в него узлы, а `delete_nodes_with_no_hazards()` удаляет узлы, на которые не ссылаются указатели опасности. Реализация обеих функций приведена в следующем листинге.

Листинг 7.8. Простая реализация функций освобождения узлов

```
template<typename T>
void do_delete(void* p) {
 delete static_cast<T*>(p);
}
```

```
struct data_to_reclaim {
 void* data;
 std::function<void(void*)> deleter;
 data_to_reclaim* next;

 template<typename T>
 data_to_reclaim(T* p) : ← (1)
  data(p),
  deleter(&do_delete<T>), next(0) {}

 ~data_to_reclaim() {
  deleter(data); ← (2)
 }
};

std::atomic<data_to_reclaim*> nodes_to_reclaim;

void add_to_reclaim_list(data_to_reclaim* node) {← (3)
 node->next = nodes_to_reclaim.load();
 while (
  !nodes_to_reclaim.compare_exchange_weak(node->next, node));
}

template<typename T>
void reclaim_later(T* data) {                    ← (4)
 add_to_reclaim_list(new data_to_reclaim(data));← (5)
}

void delete_nodes_with_no_hazards() {
 data_to_reclaim* current =
  nodes_to_reclaim.exchange(nullptr);                ← (6)
 while(current) {
  data_to_reclaim* const next = current->next;
  if (!outstanding_hazard_pointers_for(current->data)) {← (7)
   delete current;                                   ← (8)
  } else {
   add_to_reclaim_list(current);                     ← (9)
  }
  current = next;
 }
}
```

Полагаю, вы обратили внимание, что `reclaim_later()` — шаблон функции, а не обычная функция **(4)**. Объясняется это тем, что указатели опасности — это универсальный механизм, поэтому не стоит ограничивать его только узлами стека. Ранее для хранения указателей мы использовали тип `std::atomic<void*>`. Поэтому мы должны обрабатывать произвольный указательный тип, но просто указать `void*` нельзя, так как мы собираемся удалять данные по указателю, а оператору `delete` нужно знать реальный тип указателя. Как мы скоро увидим, конструктор `data_to_reclaim` прекрасно справляется с этой проблемой: `reclaim_later()` просто создает новый экземпляр `data_to_reclaim` для переданного

указателя и добавляет его в список отложенного освобождения **(5)**. Сама функция `add_to_reclaim_list()` **(3)** — не более чем простой цикл по `compare_exchange_weak()` для головного элемента списка; мы уже встречались с такой конструкцией раньше.

Но вернёмся к конструктору `data_to_reclaim` **(1)**, который также является шаблоном. Он сохраняет подлежащие удалению данные в виде указателя `void*` в члене `data`, после чего запоминает указатель на подходящую конкретизацию `do_delete()` — простую функцию, которая приводит тип `void*` к типу параметризованного указателя, а затем удаляет объект, на который он указывает. Шаблон `std::function<>` безопасно обертывает этот указатель на функцию, так что впоследствии деструктору `data_to_reclaim` для удаления данных нужно всего лишь вызвать запомненную функцию **(2)**.

Деструктор `data_to_reclaim` не вызывается, когда мы добавляем узлы в список; он вызывается только, когда на узел не ссылается ни один указатель опасности. За это отвечает функция `delete_nodes_with_no_hazards()`.

Эта функция сначала заявляет права на владение всем списком подлежащих освобождению узлов, вызывая `exchange()` **(6)**. Это простое, но крайне важное действие гарантирует, что данный конкретный набор узлов будет освобождать только один поток. Другие потоки вправе добавлять в список новые узлы и даже пытаться освободить их, никак не затрагивая работу этого потока.

Далее мы по очереди просматриваем все узлы списка, проверяя, ссылаются ли на них не сброшенные указатели опасности **(7)**. Если нет, то запись можно удалить (очистив хранящиеся в ней данные) **(8)**. В противном случае мы возвращаем элемент в список, чтобы освободить его позже **(9)**.

Хотя эта простая реализация справляется с задачей безопасного освобождения удаленных узлов, она заметно увеличивает накладные расходы. Для просмотра массива указателей опасности требуется проверить `max_hazard_pointers` атомарных переменных, и это делается при каждом вызове `pop()`. Атомарные операции по необходимости работают медленно — зачастую в 100 раз медленнее эквивалентных обычных операций на настольном ПК, — поэтому `pop()` оказывается дорогостоящей операцией. Мало того что приходится просматривать список указателей опасности для исключаемого из списка узла, так еще надо просмотреть его для каждого узла в списке ожидающих освобождения. Понятно, что это не слишком удачная идея. В списке может храниться `max_hazard_pointers` узлов, и каждый из них нужно сравнить с `max_hazard_pointers` хранимых указателей опасности. Черт! Должно существовать решение получше.

Более быстрые стратегии освобождения с применением указателей опасности

И оно, конечно же, существует. Показанное выше решение — это простая и наивная реализация указателей опасности, которую я привел, только чтобы объяснить идею. Первое, что можно сделать, — пожертвовать памятью ради быстродействия. Вместо того чтобы проверять каждый узел в списке освобождения при каждом обращении к `pop()`, мы вообще не будем пытаться освобождать узлы, пока их число в списке не превысит `max_hazard_pointers`. Тогда мы гарантированно сможем освободить хотя бы один узел. Но если просто ждать, пока в списке накопится `max_hazard_pointers+1` узлов, то выиграем мы немного. После того как число узлов достигает `max_hazard_pointers`, мы будем пытаться

освобождать их почти при каждом вызове `pop()`, так что проблема лишь немного отодвинулась во времени. Но если дождаться, пока в списке наберётся `2*max_hazard_pointers` узлов, то мы гарантированно сможем освободить по крайней мере `max_hazard_pointers` узлов, и тогда следующую попытку нужно будет делать не раньше, чем через `max_hazard_pointers` обращений к `pop()`. Это уже гораздо лучше. Вместо того чтобы просматривать `max_hazard_pointers` узлов при каждом вызове `pop()` (и, возможно, ничего не освободить), мы проверяем `2*max_hazard_pointers` через каждые `max_hazard_pointers` вызовов `pop()` и освобождаем не менее `max_hazard_pointers`. Получается, что в среднем мы проверяем два узла при каждом вызове `pop()`, и один из них точно освобождается.

Но и у этого решения есть недостаток (помимо увеличенного расхода памяти): теперь мы должны подсчитывать узлы в списке освобождения, то есть использовать атомарный счетчик, а, кроме того, за доступ к самому списку конкурируют несколько потоков. Если память позволяет, то можно предложить еще более эффективную схему освобождения: каждый поток хранит собственный список освобождения в поточно-локальной памяти. Тогда ни для счетчика, ни для доступа к списку не понадобятся атомарные переменные. Но в обмен придется выделить память для `max_hazard_pointers * max_hazard_pointers` узлов. Если поток завершается прежде, чем освобождены все его узлы, то оставшиеся можно перенести в глобальный список, как и раньше, а затем добавить в локальный список следующего потока, пожелавшего выполнить процедуру освобождения.

Еще один недостаток указателей опасности состоит в том, что они защищены патентной заявкой, поданной IBM[13]. Если вы пишете программное обеспечение, которое будет применяться в стране, где эти патенты признаются, то придется получить соответствующую лицензию. Это проблема, общая для многих методов освобождения памяти без блокировок; поскольку в этой области ведутся активные исследования, крупные компании берут патенты всюду, где могут. Возможно, вы задаетесь вопросом, зачем я посвятил так много страниц описанию техники, которой многие не смогут воспользоваться. Что ж, вопрос не праздный. Во-первых, в некоторых случаях ей можно воспользоваться, не платя лицензионных отчислений. Например, если вы разрабатываете бесплатную программу на условиях лицензии GPL[14], то она может подпадать под заявление IBM об отказе от патентных притязаний[15]. Во-вторых — и это более существенно — объяснение техники помогает высветить вещи, о которых надо помнить при написании кода, свободного от блокировок, например, о плате за атомарные операции.

А существуют ли непатентованные методы освобождения памяти, применимые в программах без блокировок? К счастью, да. Один из них — подсчет ссылок.

7.2.4. Нахождение используемых узлов с помощью подсчета ссылок

В разделе 7.2.2 мы видели, что проблема удаления узлов сводится к задаче нахождения узлов, к которым еще обращаются потоки-читатели. Если бы можно было точно узнать, на какие узлы есть ссылки и когда количество ссылок обращается в нуль, то узлы можно было бы удалить. Указатели опасности решают эту проблему путем хранения списка используемых узлов, а механизм подсчета ссылок — путем хранения числа потоков, обращающихся к каждому узлу.

Выглядит просто и элегантно, но реализовать на практике очень трудно. Сразу приходит

в голову мысль, что для такой задачи подошло бы что-то вроде `std::shared_ptr<>` — ведь это и есть указатель с подсчетом ссылок. Увы, хотя некоторые операции над `std::shared_ptr<>` атомарны, не гарантируется, что они свободны от блокировок. Сам по себе класс `std::shared_ptr<>` ничем не хуже прочих с точки зрения операций над атомарными типами, но он рассчитан на применение в самых разных контекстах, и попытка сделать атомарные операции над ним свободными от блокировок, скорее всего, привела бы к увеличению накладных расходов при любом его использовании. Если на вашей платформе функция `std::atomic_is_lock_free(&some_shared_ptr)` возвращает `true`, то проблему освобождения памяти можно считать полностью решенной. Достаточно хранить в списке объекты `std::shared_ptr<node>`, как показано в следующем листинге.

Листинг 7.9. Свободный от блокировок стек на основе свободной от блокировок реализации `std::shared_ptr<>`

```
template<typename T>
class lock_free_stack {
private:
 struct node {
  std::shared_ptr<T> data;
  std::shared_ptr<node> next;
  node(T const& data_) :
   data(std::make_shared<T>(data_)) {}
 };

 std::shared_ptr<node> head;

public:
 void push(T const& data) {
  std::shared_ptr<node> const new_node =
   std::make_shared<node>(data);
  new_node->next = head.load();
  while (!std::atomic_compare_exchange_weak(
   &head, &new_node->next, new_node));
 }

 std::shared_ptr<T> pop() {
  std::shared_ptr<node> old_head = std::atomic_load(&head);
  while(old_head && !std::atomic_compare_exchange_weak(
   &head, &old_head, old_head->next));
  return old_head ? old_head->data : std::shared_ptr<T>();
 }
};
```

В том весьма вероятном случае, когда реализация `std::shared_ptr<>` не свободна от блокировок, управлять подсчетом ссылок придется самостоятельно.

В одном из возможных способов используется не один, а два счетчика ссылок — внутренний и внешний. Их сумма равна общему количеству ссылок на узел. Внешний счетчик хранится вместе с указателем на узел и увеличивается при каждом чтении этого указателя. Когда читатель закапчивает работу с узлом, он уменьшает его *внутренний* счетчик. Таким образом, по завершении простой операции чтения указателя внешний счетчик увеличится на единицу, а внутренний уменьшится на единицу.

Когда необходимость в связи между внешним счетчиком и указателем отпадает (то есть

узел невозможно получить из области памяти, доступной другим потокам), внутренний счетчик увеличивается на величину внешнего минус 1, а внешний отбрасывается. Если внутренний счетчик обратился в нуль, значит, никаких ссылок на узел извне не осталось и его можно удалять. Для обновления разделяемых данных по-прежнему необходимо применять атомарные операции. Теперь рассмотрим реализацию свободного от блокировок стека, в которой эта техника используется для гарантии того, что узлы освобождаются, только когда это безопасно.

В листинге ниже показала внутренняя структура данных и реализация функции `push()` — простая и элегантная.

Листинг 7.10. Помещение узла в свободный от блокировок стек с разделённым счётчиком ссылок

```
template<typename T>
class lock_free_stack {
private:
 struct node;

 struct counted_node_ptr {← (1)
  int external_count;
  node* ptr;
 };

 struct node {
  std::shared_ptr<T> data;← (2)
  std::atomic<int> internal_count;
  counted_node_ptr next;  ← (3)
  node(T const& data_) :
   data(std::make_shared<T>(data_)), internal_count(0) {}
 };

 std::atomic<counted_node_ptr> head;← (4)

public:
 ~lock_free_stack() {
  while(pop());
 }

 void push(T const& data) {← (5)
  counted_node_ptr new_node;
  new_node.ptr = new node(data);
  new_node.external_count = 1;
  new_node.ptr->next = head.load();
  while(
   !head.compare_exchange_weak(new_node.ptr->next, new_node));
 }
};
```

Обратите внимание, что внешний счетчик хранится вместе с указателем на узел в структуре `counted_node_ptr` **(1)**. Эта структура затем используется для представления указателя `next` в структуре `node` **(3)**, где хранится также внешний счетчик **(2)**. Поскольку `counted_node_ptr` определена как `struct`, то ее можно использовать в шаблоне

std::atomic<> для представления головы списка head **(4)**.

На платформах, где поддерживается операция сравнения и обмена двойного слова, размер этой структуры достаточно мал для того, чтобы тип данных std::atomic<counted_node_ptr> был свободен от блокировок. Если вы работаете на другой платформе, то лучше пользоваться вариантом std::shared_ptr<>, приведенным в листинге 7.9, потому что в std::atomic<> для гарантирования атомарности используется мьютекс, когда тип слишком велик и с помощью машинных команд обеспечить атомарность невозможно (а, следовательно, ваш алгоритм, якобы «свободный от блокировок», на самом теле таковым не является). Можно поступить и по-другому — если вы готовы ограничить размер счетчика и знаете, что на данной платформе в указателе есть неиспользуемые биты (например, потому что адресное пространство представлено 48 битами, а под указатель отводится 64 бита), то счетчик можно хранить в незанятых битах указателя, и тогда оба поля поместятся в одно машинное слово. Но для таких трюков нужно хорошо знать особенности платформы, так что в этой книге мы их обсуждать не будем.

Функция push() относительно проста **(5)**. Мы конструируем объект counted_node_ptr, ссылающийся на только что выделенный узел node с ассоциированными данными, и в поле next узла node записываем текущее значение head. Затем с помощью compare_exchange_weak() устанавливаем новое значение head, как и раньше. Внутренний счетчик internal_count инициализируется нулем, а внешний external_count — единицей. Поскольку узел новый, на него пока существует только одна внешняя ссылка (из самого указателя head).

Как обычно, все сложности сосредоточены в реализации pop(), показанной в следующем листинге.

Листинг 7.11. Выталкивание узла из свободного от блокировок стека с разделённым счётчиком ссылок

```
template<typename T>
class lock_free_stack {
private:
 void increase_head_count(counted_node_ptr& old_counter) {
  counted_node_ptr new_counter;
  do {
   new_counter = old_counter;
   ++new_counter.external_count;
  }
  while (
   !head.compare_exchange_strong(old_counter, new_counter));← (1)
  old_counter.external_count = new_counter.external_count;
 }

public:
 std::shared_ptr<T> pop() {
  counted_node_ptr old_head = head.load();
  for (;;) {
   increase_head_count(old_head);
   node* const ptr = old_head.ptr;← (2)
   if (!ptr) {
    return std::shared_ptr<T>();
   }
```

```
    if (head.compare_exchange_strong(old_head, ptr->next)) {← (3)
     std::shared_ptr<T> res;
     res.swap(ptr->data);                                    ← (4)
     int const count_increase = old_head.external_count - 2;← (5)
     if (ptr->internal_count.fetch_add(count_increase) ==
         -count_increase) {                                  ← (6)
      delete ptr;
     }
     return res; ← (7)
    } else if (ptr->internal_count.fetch_sub(1) == 1) {
     delete ptr; ← (8)
    }
   }
  }
};
```

Теперь, загрузив значение `head`, мы должны сначала увеличить счетчик внешних ссылок на узел `head`, показав, что ссылаемся на него, — только тогда его можно безопасно будет разыменовывать. Если попытаться разыменовать указатель *до* увеличения счетчика ссылок, то вполне может случиться так, что другой поток освободит узел раньше, чем мы успеем обратиться к нему, и, стало быть, оставит нам висячий указатель. *Именно в этом главная причина использования разделенного счетчика ссылок*: увеличивая внешний счетчик ссылок, мы гарантируем, что указатель останется действительным в течение всего времени работы с ним. Увеличение производится в цикле по `compare_exchange_strong()` **(1)**, где устанавливаются все поля структуры, чтобы быть уверенным, что другой поток не изменил в промежутке указатель.

Увеличив счетчик, мы можем без опаски разыменовать поле `ptr` объекта, загруженного из `head`, и получить тем самым доступ к адресуемому узлу **(2)**. Если оказалось, что указатель нулевой, то мы находимся в конце списка — больше записей нет. В противном случае мы можем попытаться исключить узел из списка, выполнив `compare_exchange_strong()` с головным узлом `head` **(3)**.

Если `compare_exchange_strong()` возвращает `true`, то мы приняли на себя владение узлом и можем с помощью функции `swap()` вытащить из него данные, которые впоследствии вернём **(4)**. Тем самым гарантируется, что данные случайно не изменятся, если вдруг другие обращающиеся к стеку другие потоки удерживают указатели на этот узел. Затем можно прибавить внешний счетчик к внутреннему с помощью атомарной операции `fetch_add` **(6)**. Если теперь счетчик ссылок стал равен нулю, то *предыдущее* значение (то, которое возвращает `fetch_add`) было противоположно только что прибавленному, и тогда узел можно удалять. Важно отметить, что прибавленное значение на самом деле *на 2 меньше* внешнего счетчика **(5)**; мы исключили узел из списка, вследствие чего значение счетчика уменьшилось на 1, и больше не обращаемся к узлу из данного потока, что дает уменьшение еще на 1. Неважно, удаляется узел или нет, наша работа закончена, и мы можем вернуть данные **(7)**.

Если сравнение с обменом **(3)** *не проходит*, значит, другой поток сумел удалить узел раньше нас, либо другой поток добавил в стек новый узел. В любом случае нужно начать с начала — с новым значением `head`, которое вернула функция `compare_exchange_strong()`. Но прежде необходимо уменьшить счетчик ссылок на узел, который мы пытались исключить раньше. Этот поток больше не будет к нему обращаться. Если наш поток — последний,

удерживавший ссылку на этот узел (потому что другой поток вытолкнул его из стека), то внутренний счетчик ссылок равен 1, так что после вычитания 1 он обратится в нуль. В таком случае мы можем удалить узел прямо здесь, не дожидаясь перехода к следующей итерации цикла **(8)**.

До сих мы задавали для всех атомарных операций упорядочение доступа к памяти `std::memory_order_seq_cst`. В большинстве систем это самый неэффективный режим с точки зрения времени выполнения и накладных расходов на синхронизацию, причем в ряде случаев разница весьма ощутима. Но теперь, определившись с логикой структуры данных, можно подумать и о том, чтобы ослабить некоторые требования к упорядочению, — все-таки не хотелось бы, чтобы пользователи стека несли лишние расходы. Итак, перед тем как расстаться со стеком и перейти к проектированию свободной от блокировок очереди, еще раз присмотримся к операциям стека и спросим себя, нельзя ли где-нибудь использовать более слабое упорядочение доступа, сохранив тот же уровень безопасности?

7.2.5. Применение модели памяти к свободному от блокировок стеку

Прежде чем менять схему упорядочения, нужно исследовать все операции и определить, какие между ними должны быть отношения. Затем можно будет вернуться и найти минимальное упорядочение, которое эти отношения обеспечивает. Чтобы это сделать, потребуется взглянуть на ситуацию с точки зрения потоков в нескольких разных сценариях. Простейший сценарий возникает, когда один поток помещает элемент данных в стек, а другой через некоторое время извлекает его оттуда, с него и начнем.

В этом сценарии есть три существенных участника. Во-первых, структура `counted_node_ptr`, используемая для передачи данных — узла `head`. Во-вторых, структура `node`, на которую `head` ссылается. И, в-третьих, сами данные, на которые указывает узел.

Поток, выполняющий `push()`, сначала конструирует элемент данных и объект `node`, затем устанавливает `head`. Поток, выполняющий `pop()`, сначала загружает значение `head`, затем в цикле сравнения с обменом увеличивает хранящийся в нем счетчик ссылок, после чего читает структуру `node`, чтобы извлечь из нее значение `next`. Из этой последовательности можно вывести требуемое отношение; значение `next` — простой неатомарный объект, поэтому для его безопасного чтения должно существовать отношение происходит-раньше между операциями сохранения (в заталкивающем потоке) и загрузки (в выталкивающем потоке). Поскольку в `push()` имеется единственная атомарная операция — `compare_exchange_weak()`, а для существования отношения происходит-раньше между потоками нам нужна операция *освобождения* (release), то для функции `compare_exchange_weak()` необходимо задать упорядочение `std::memory_order_release` или более сильное. Если `compare_exchange_weak()` вернула `false`, то ничего не было изменено, и мы можем продолжить цикл, следовательно в этом случае нужна только семантика `std::memory_order_relaxed`:

```
void push(T const& data) {
 counted_node_ptr new_node;
 new_node.ptr = new node(data);
 new_node.external_count = 1;
 new_node.ptr->next = head.load(std::memory_order_relaxed);
 while (!head.compare_exchange_weak(
  new_node.ptr->next, new_node,
```

```
    std::memory_order_release, std::memory_order_relaxed));
}
```

А что можно сказать о коде `pop()`? Чтобы получить желаемое отношение происходит-раньше, перед доступом к `next` необходима операция с семантикой `std::memory_order_acquire` или более сильной. Указатель, который разыменовывается для доступа к полю `next`, — это прежнее значение, прочитанное операцией `compare_exchange_strong()` в `increase_head_count()`, поэтому указанная семантика нужна в случае успеха. Как и в `push()`, если обмен закончился неудачно, мы просто повторяем цикл, поэтому для отказа можно задать ослабленное упорядочение:

```
void increase_head_count(counted_node_ptr& old_counter) {
 counted_node_ptr new_counter;
 do {
  new_counter = old_counter;
  ++new_counter.external_count;
 }
 while (!head.compare_exchange_strong(
  old_counter, new_counter,
  std::memory_order_acquire, std::memory_order_relaxed));
 old_counter.external_count = new_counter.external_count;
}
```

Если вызов `compare_exchange_strong()` завершается успешно, то мы знаем, что раньше в поле `ptr` прочитанного значения находилось то, что теперь хранится в переменной `old_counter`. Поскольку сохранение в `push()` было операцией освобождения, а данный вызов `compare_exchange_strong()` — операция захвата, то сохранение синхронизируется-с загрузкой, и мы имеем отношение происходит-раньше. Следовательно, сохранение в поле `ptr` в `push()` происходит-раньше доступа к `ptr->next` в `pop()`, и мы в безопасности.

Отметим, что для этого анализа порядок доступа к памяти в начальном вызове `head.load()` не имел значения, поэтому в нем безопасно задать семантику `std::memory_order_relaxed`.

Далее на очереди операция `compare_exchange_strong()`, которая записывает в `head` значение `old_head.ptr->next`. Нужно ли наложить на нее какие-нибудь ограничения, чтобы гарантировать целостность данных в этом потоке? Если обмен завершается успешно, то мы обращаемся к `ptr->data`, поэтому должны быть уверены, что сохранение `ptr->data` в потоке, выполняющем `push()`, происходит-раньше загрузки. Но такая уверенность уже есть: операция захвата в `increase_head_count()` гарантирует, что существует отношение синхронизируется-с между сохранением в потоке, выполняющем `push()`, и операцией сравнения с обменом. Поскольку сохранение данных в потоке, выполняющем `push()`, расположено перед сохранением `head`, а вызов `increase_head_count()` расположен перед загрузкой `ptr->data`, то отношение происходит-раньше имеет место, и всё будет хорошо, даже если для операции сравнения с обменом в `pop()` задана семантика `std::memory_order_relaxed`. Есть еще всего одно место, где изменяется `ptr->data` — тот самый вызов `swap()`, на который вы сейчас смотрите, и ни один другой поток не может оперировать тем же узлом — в этом и заключается смысл сравнения с обменом.

Если `compare_exchange_strong()` завершается неудачно, то к новому значению `old_head` не будет обращений до следующей итерации цикла, и, поскольку мы уже решили, что семантики `std::memory_order_acquire` хватало в `increase_head_count()`, то здесь будет достаточно `std::memory_order_relaxed`.

Что можно сказать насчёт других потоков? Нужны ли более сильные ограничения,

чтобы и другие потоки работали безопасно? Нет, не нужны, потому что `head` модифицируется только операциями сравнения с обменом. Будучи операциями чтения-модификации-записи, они составляют часть последовательности освобождений, начатой операцией сравнения с обменом в `push()`. Поэтому `compare_exchange_weak()` в `push()` синхронизируется-с операцией `compare_exchange_strong()` в `increase_head_count()`, которая прочитает сохраненное значение, даже если в промежутке другие потоки изменят `head`.

Мы почти закончили, осталось только рассмотреть функции, в которых используются операции `fetch_add()`, изменяющие счетчик ссылок. Поток, который добрался до возврата данных из узла, может продолжать в твердой уверенности, что никакой другой поток не сможет модифицировать хранящиеся в узле данные. Однако любой поток, который потерпел *неудачу* при извлечении данных, знает, что какой-то другой поток данные в узле *модифицировал*; он использовал функцию `swap()` для извлечения данных. Следовательно, чтобы предотвратить гонку за данными мы должны гарантировать, что `swap()` происходит-раньше `delete`. Чтобы добиться этого, проще всего задать семантику `std::memory_order_release` при вызове `fetch_add()` в ветви, где мы возвращаем данные, и семантику `std::memory_order_acquire` — в ветви, где мы возвращаемся в начало цикла. Однако даже это перебор — лишь один поток выполняет `delete` (тот, что сбросил счетчик в нуль), поэтому только этому потоку нужно выполнить операцию захвата. К счастью, поскольку `fetch_add()` — операция чтения-модификации-записи, то она составляет часть последовательности освобождений, поэтому для достижения цели нам достаточно дополнительной операции `load()`. Если в ветви, где происходит возврат в начало цикла, счетчик ссылок уменьшается до нуля, то здесь же можно перезагрузить счетчик ссылок с семантикой `std::memory_order_acquire`, чтобы обеспечить требуемое отношение синхронизируется-с, а в самой операции `fetch_add()` достаточно задать `std::memory_order_relaxed`. Окончательная реализация стека с новой версией `pop()` приведена ниже.

Листинг 7.12. Свободный от блокировок стек с подсчётом ссылок и ослабленными атомарными операциями

```
template<typename T>
class lock_free_stack {
private:
 struct node;

 struct counted_node_ptr {
  int external_count;
  node* ptr;
 };

 struct node {
  std::shared_ptr<T> data;
  std::atomic<int> internal_count;
  counted_node_ptr next;
  node(T const& data_):
  data(std::make_shared<T>(data_)), internal_count(0) {}
 };
```

```
 std::atomic<counted_node_ptr> head;

 void increase_head_count(counted_node_ptr& old_counter) {
  counted_node_ptr new_counter;

  do {
   new_counter = old_counter;
   ++new_counter.external_count;
  }
  while (!head.compare_exchange_strong(old_counter, new_counter,
         std::memory_order_acquire,
         std::memory_order_relaxed));

  old_counter.external_count = new_counter.external_count;
 }

public:
 ~lock_free_stack() {
  while(pop());
 }

 void push(T const& data) {
  counted_node_ptr new_node;
  new_node.ptr = new node(data);
  new_node.external_count = 1;
  new_node.ptr->next = head.load(std::memory_order_relaxed);
  while (!head.compare_exchange_weak(
           new_node.ptr->next, new_node,
           std::memory_order_release,
           std::memory_order_relaxed));
 }

 std::shared_ptr<T> pop() {
  counted_node_ptr old_head =
   head.load(std::memory_order_relaxed);
  for (;;) {
   increase_head_count(old_head);
   node* const ptr = old_head.ptr;
   if (!ptr) {
    return std::shared_ptr<T>();
   }
   if (head.compare_exchange_strong(old_head, ptr->next,
       std::memory_order_relaxed)) {
    std::shared_ptr<T> res;
    res.swap(ptr->data);
    int const count_increase = old_head.external_count - 2;
    if (ptr->internal_count.fetch_add(count_increase,
        std::memory_order_release) == -count_increase) {
     delete ptr;
    }
    return res;
   }
   else if (ptr->internal_count.fetch_add(-1,
            std::memory_order_relaxed) == 1) {
    ptr->internal_count.load(std::memory_order_acquire);
    delete ptr;
```

```
    }
   }
  }
};
```

Мы немало потрудились, но наконец-то дошли до конца, и стек теперь стал куда лучше. За счет тщательно продуманного применения ослабленных операций нам удалось повысить производительность, не жертвуя корректностью. Как видите, реализация `pop()` теперь насчитывает 37 строк вместо 8 в эквивалентной реализации `pop()` для стека с блокировками (листинг 7.1) и 7 строк для простого свободного от блокировок стека без управления памятью (листинг 7.2). При рассмотрении свободной от блокировок очереди мы встретимся с аналогичной ситуацией: сложность кода в значительной степени обусловлена именно управлением памятью.

7.2.6. Потокобезопасная очередь без блокировок

Очередь отличается от стека прежде всего тем, что операции `push()` и `pop()` обращаются к разным частям структуры данных, тогда как в стеке та и другая работают с головным узлом списка. Следовательно, и проблемы синхронизации тоже другие. Требуется сделать так, чтобы изменения, произведенные на одном конце, были видны при доступе с другого конца. Однако структура функции `try_pop()` в листинге 6.6 не так уж сильно отличается от структуры `pop()` в простом свободном от блокировок стеке в листинге 7.2, поэтому можно с достаточными основаниями предположить, что и весь свободный от блокировок код будет схожим. Посмотрим, так ли это.

Если взять листинг 6.6 за основу, то нам понадобятся два указателя на `node`: один для головы списка (`head`), второй — для хвоста (`tail`). Поскольку мы собираемся обращаться к ним из нескольких потоков, то надо бы сделать эти указатели атомарными и расстаться с соответствующими мьютексами. Начнём с этого небольшого изменения и посмотрим, куда оно нас приведет. Результат показан в листинге ниже.

Листинг 7.13. Свободная от блокировок очередь с одним производителем и одним потребителем

```
template<typename T>
class lock_free_queue {
private:
 struct node {
  std::shared_ptr<T> data;
  node* next;
  node():
   next(nullptr) {}
 };

 std::atomic<node*> head;
 std::atomic<node*> tail;
 node* pop_head() {
  node* const old_head = head.load();
  if (old_head == tail.load()) {← (1)
   return nullptr;
  }
  head.store(old_head->next);
```

```
    return old_head;
   }

  public:
   lock_free_queue():
    head(new node), tail(head.load()) {}

   lock_free_queue(const lock_free_queue& other) = delete;
   lock_free_queue& operator=(
    const lock_free_queue& other) = delete;

   ~lock_free_queue() {
    while(node* const old_head = head.load()) {
     head.store(old_head->next);
     delete old_head;
    }
   }

   std::shared_ptr<T> pop() {
    node* old_head = pop_head();
    if (!old_head) {
     return std::shared_ptr<T>();
    }

    std::shared_ptr<T> const res(old_head->data);← (2)
    delete old_head;
    return res;
   }

   void push(T new_value) {
    std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
    node* p = new node;                  ← (3)
    node* const old_tail = tail.load(); ← (4)
    old_tail->data.swap(new_data);       ← (5)
    old_tail->next = p;                  ← (6)
    tail.store(p);                       ← (7)
   }
  };
```

На первый взгляд, неплохо, и если в каждый момент времени существует только один поток, вызывающий `push()`, и только один поток, вызывающий `pop()`, то вообще всё прекрасно. Важно отметить, что в этом случае существует отношение происходит-раньше между `push()` и `pop()`, благодаря которому извлечение данных безопасно. Сохранение `tail` **(7)** синхронизируется-с загрузкой `tail` **(1)**, сохранение указателя на `data` в предыдущем узле **(5)** расположено перед сохранением `tail`, а загрузка `tail` расположена перед загрузкой указателя на `data` **(2)**, поэтому сохранение `data` происходит раньше его загрузки, и всё замечательно. Таким образом, мы получили корректно обслуживаемую очередь с *одним производителем и одним потребителем*.

Проблемы начинаются, когда несколько потоков вызывают `push()` или `pop()` одновременно. Сначала рассмотрим `push()`. Если два потока одновременно вызывают `push()`, то оба выделяют память для нового фиктивного узла **(3)**, оба читают *одно и то же*

значение `tail` **(4)** и, следовательно, оба изменяют данные-члены `data` и `next` одного и того же узла **(5)**, **(6)**. А это уже гонка за данными!

Аналогичные проблемы возникают в `pop_head()`. Если два потока вызывают эту функцию одновременно, то оба читают одно и то же значение `head`, и оба перезаписывают старое значение одним и тем же указателем `next`. Оба потока теперь думают, что получили один и тот же узел, — прямой путь к катастрофе. Мы должны не только сделать так, чтобы лишь один поток извлекал данный элемент, но и позаботиться о том, чтобы другие потоки могли безопасно обращаться к члену `next` узла, который прочитали из `head`. Это точно та же проблема, с которой мы сталкивались при написании `pop()` для свободного от блокировок стека, поэтому и любое из предложенных тогда решений можно применить здесь.

Итак, проблему `pop()` можно считать решенной, но как быть с `push()`? Здесь трудность заключается в том, что для получения требуемого отношения происходит-раньше между `push()` и `pop()` мы должны заполнить поля фиктивного узла до обновления `tail`. Но это означает, что одновременные вызовы `push()` конкурируют за те же самые данные, так как был прочитал один и тот же указатель `tail`.

Решение проблемы нескольких потоков в `push()`

Один из способов — добавить фиктивный узел между реальными. Тогда единственной частью текущего узла `tail`, нуждающейся в обновлении, будет указатель `next`, который, следовательно, можно было бы сделать атомарным. Если потоку удалось записать в `next` указатель на свой новый узел вместо `nullptr`, то он успешно добавил узел; в противном случае ему придется начать сначала и снова прочитать `tail`. Это потребует небольшого изменения в `pop()` — нужно будет игнорировать узлы с нулевым указателем на данные и возвращаться в начало цикла. Недостаток этого решения в том, что при каждом вызове `pop()` придется как правило исключать из списка два узла и производить в два раза больше операций выделения памяти.

Второй способ — сделать указатель `data` атомарным и устанавливать его с помощью операции сравнения с обменом. Если она завершится успешно, то мы получили свой хвостовой узел и можем безопасно записать в `next` указатель на наш новый узел, а затем обновить `tail`. Если же сравнение с обменом завершается неудачно, потому что другой поток успел сохранить данные, мы возвращаемся в начало цикла, заново читаем `tail` и пробуем снова. Если атомарные операции над `Std::shared_ptr<>` свободны от блокировок, то дело сделано. Если нет, нужна альтернатива. Можно, например, заставить `pop()` возвращать `std::unique_ptr<>` (в конце концов, это ведь единственная ссылка на объект) и сохранять данные в очереди в виде простого указателя. Тогда его можно было бы хранить как `std::atomic<T*>` и впоследствии обновлять с помощью `compare_exchange_strong()`. Если воспользоваться для поддержки нескольких потоков в `pop()` схемой подсчета ссылок из листинга 7.11, то `push()` будет выглядеть следующим образом.

Листинг 7.14. Первая (неудачная) попытка переработки `push()`

```
void push(T new_value) {
 std::unique_ptr<T> new_data(new T(new_value));
 counted_node_ptr new_next;
 new_next.ptr = new node;
```

```
    new_next.external_count = 1;
    for (;;) {
     node* const old_tail = tail.load();← (1)
     T* old_data = nullptr;
     if (old_tail->data.compare_exchange_strong(
      old_data, new_data.get())) {      ← (2)
      old_tail->next = new_next;
      tail.store(new_next.ptr);         ← (3)
      new_data.release();
      break;
     }
    }
   }
```

Применение схемы подсчета ссылок устраняет эту конкретную гонку, но в `push()` имеются и другие гонки. Взглянув на переработанную версию `push()` в листинге 7.14, вы обнаружите ту же ситуацию, что уже встречалась нам в стеке: загрузка атомарного указателя **(1)** и разыменование этого указателя **(2)**. В промежутке между этими двумя операциями другой поток может изменить указатель **(3)**, что в конечном итоге приведет к освобождению памяти, запятой узлом (в `pop()`). Если это произойдет раньше, чем мы разыменовываем указатель, то получится неопределенное поведение. Ой! Возникает искушение добавить в `tail` внешний счетчик, как мы уже поступили для `head`, однако на каждый узел уже имеется внешний счетчик в указателе `next` в предыдущем узле очереди. Если хранить два внешних счетчика для одного узла, то потребуется модифицировать схему подсчета ссылок, чтобы не удалить узел преждевременно. Проблему можно решить, подсчитывая число внешних счетчиков в структуре `node` и уменьшая это число при уничтожении внешнего счетчика (одновременно с прибавлением значения внешнего счетчика к значению внутреннего). Если внутренний счетчик равен нулю, а внешних не осталось, то узел можно удалять. Эту технику я впервые встретил в проекте Джо Сейга (Joe Seigh) Atomic Ptr Plus[16]. В следующем листинге показано, как выглядит `push()` при использовании такой схемы.

Листинг 7.15. Реализация `push()` для очереди без блокировок с подсчётом ссылок на `tail`

```
    template<typename T>
    class lock_free_queue {
    private:
     struct node;

     struct counted_node_ptr {
      int external_count;
      node* ptr;
     };

     std::atomic<counted_node_ptr> head;
     std::atomic<counted_node_ptr> tail;← (1)

     struct node_counter {
      unsigned internal_count:30;
      unsigned external_counters:2;← (2)
     };
```

```
 struct node {
  std::atomic<T*> data;
  std::atomic<node_counter> count;← (3)
  counted_node_ptr next;
  node() {
   node_counter new_count;
   new_count.internal_count = 0;
   new_count.external_counters = 2;← (4)
   count.store(new_count);
   next.ptr = nullptr;
   next.external_count = 0;
  }
 };

public:
 void push(T new_value) {
  std::unique_ptr<T> new_data(new T(new_value));
  counted_node_ptr new_next;
  new_next.ptr = new node;
  new_next.external_count = 1;
  counted_node_ptr old_tail = tail.load();

  for (;;) {
   increase_external_count(tail, old_tail);        ← (5)
   T* old_data = nullptr;
   if (old_tail.ptr->data.compare_exchange_strong(← (6)
       old_data, new_data.get())) {
    old_tail.ptr->next = new_next;
    old_tail = tail.exchange(new_next);
    free_external_counter(old_tail);               ← (7)
    new_data.release();
    break;
   }
   old_tail.ptr->release_ref();
  }
 }
};
```

В листинге 7.15 `tail` теперь имеет такой же тип `atomic<counted_node_ptr>`, как и `head` **(1)**, а в структуру `node` добавлен член `count` вместо прежнего `internal_count` **(3)**. Член `count` сам представляет собой структуру с двумя полями: `internal_count` и `external_counters` **(2)**. Под поле `external_counters` отведено только 2 бита, потому что внешних счетчиков может быть не более двух. Воспользовавшись битовыми полями и отведя под `internal_count` 30 бит, мы ограничили длину поля счетчика 32 битами. В результате мы убиваем сразу двух зайцев: и значение внутреннего счетчика может быть достаточно велико, и вся структура помещается в машинное слово на 32- и 64-разрядных машинах. Очень важно изменять счетчики как единое целое, чтобы избежать гонки. Как это делается, мы покажем чуть ниже. На многих платформах хранение структуры в одном машинном слове повышает шансы на то, что атомарные операции окажутся свободными от блокировок.

При инициализации структуры `node` в поле `internal_count` записывается 0, а в поле `external_counters` — 2 **(4)**, потому что сразу после добавления нового узла в очередь на

него есть две ссылки: из `tail` и из указателя `next` в предыдущем узле. Код самой функции `push()` похож на приведенный в листинге 7.14 с тем отличием, что перед тем как разыменовывать загруженное из `tail` значение, чтобы вызвать `compare_exchange_strong()` для члена узла `data` **(6)**, мы вызываем новую функцию `increase_external_count()` которая увеличивает счетчик **(5)**, а затем функцию `free_external_counter()` для старого хвоста `old_tail` **(7)**.

Разобравшись с `push()`, обратим наши взоры на `pop()`. В ее коде (см. листинг 7.16) логика подсчета ссылок из реализации `pop()` в листинге 7.11 комбинируется с логикой извлечения из очереди в листинге 7.13.

Листинг 7.16. Извлечение узла из очереди без блокировок с подсчётом ссылок на `tail`

```
template<typename T>
class lock_free_queue {
private:
 struct node {
 void release_ref();
};

public:
 std::unique_ptr<T> pop() {
  counted_node_ptr old_head =
   head.load(std::memory_order_relaxed);    ← (1)
  for (;;) {
   increase_external_count(head, old_head);← (2)
   node* const ptr = old_head.ptr;
   if (ptr == tail.load().ptr) {
    ptr->release_ref();← (3)
    return std::unique_ptr<T>();
   }
   if (head.compare_exchange_strong(old_head, ptr->next)) {← (4)
    T* const res = ptr->data.exchange(nullptr);
    free_external_counter(old_head);← (5)
    return std::unique_ptr<T>(res);
   }
   ptr->release_ref();
  }
 }
};
```

Все начинается с загрузки значения `old_head` перед входом в цикл **(1)** и до увеличения внешнего счетчика в загруженном значении **(2)**. Если узел `head` совпадает с `tail`, то можно освободить ссылку **(3)** и вернуть нулевой указатель, потому что очередь пуста. Если же в очереди есть данные, то мы пытаемся заявить на них свои права с помощью `compare_exchange_strong()` (4). Как и в случае стека в листинге 7.11, мы при этом сравниваем внешний счетчик и указатель как единое целое; если хотя бы один из них изменился, то мы должны вернуться в начало цикла, освободив предварительно ссылку 6. Если обмен завершился удачно, то мы получили в свое распоряжение данные в узле, поэтому можем вернуть их вызывающей программе, освободив предварительно внешний счетчик ссылок на извлеченный узел **(5)**. После того как оба внешних счетчика освобождены, а внутренний счетчик обратился в нуль, сам узел можно удалять. Вспомогательные функции

подсчета ссылок приведены в листингах 7.17, 7.18 и 7.19.

Листинг 7.17. Освобождение ссылки на узел в очереди без блокировок

```
template<typename T>
class lock_free_queue {
private:
 struct node {
  void release_ref() {
   node_counter old_counter =
    count.load(std::memory_order_relaxed);
   node_counter new_counter;
   do {
    new_counter = old_counter;
    --new_counter.internal_count;          ← (1)
   }
   while (!count.compare_exchange_strong(← (2)
          old_counter, new_counter,
          std::memory_order_acquire, std::memory_order_relaxed));

   if (
    !new_counter.internal_count &&
    !new_counter.external_counters) {
    delete this; ← (3)
   }
  }
 };
};
```

Реализация `node::release_ref()` лишь немногим отличается от аналогичного кода в `lock_free_stack::pop()` (см. листинг 7.11). Там мы работали с единственным внешним счетчиком, поэтому достаточно было вызвать `fetch_sub`. Здесь же необходимо атомарно обновить всю структуру `count`, хотя в действительности мы хотим модифицировать только поле `internal_count` **(1)**. Поэтому никуда не деться от цикла сравнения с обменом **(2)**. Если после уменьшения `internal_count` оказалось, что и внутренний, и внешний счетчик равны нулю, то это была последняя ссылка, и мы можем удалять узел **(3)**.

Листинг 7.18. Получение новой ссылки на узел в очереди без блокировок

```
template<typename T>
class lock_free_queue {
private:
 static void increase_external_count(
  std::atomic<counted_node_ptr>& counter,
  counted_node_ptr& old_counter) {
  counted_node_ptr new_counter;
  do {
   new_counter = old_counter;
   ++new_counter.external_count;
  }
  while (!counter.compare_exchange_strong(
   old_counter, new_counter,
   std::memory_order_acquire, std::memory_order_relaxed));

  old_counter.external_count = new_counter.external_count;
```

```
  }
};
```

Листинг 7.18 завершает картину. На этот раз мы не освобождаем ссылку, а получаем новую и увеличиваем внешний счетчик. Функция `increase_external_count()` аналогична `increase_head_count()` из листинга 7.12, отличаясь от нее тем, что преобразована в статическую функцию-член, которая принимает подлежащий обновлению внешний счетчик извне, а не оперирует внутренним членом класса.

Листинг 7.19. Освобождение счётчика внешних ссылок на узел в очереди без блокировок

```
template<typename T>
class lock_free_queue {
private:
 static void free_external_counter(
  counted_node_ptr &old_node_ptr) {
  node* const ptr = old_node_ptr.ptr;
  int const count_increase = old_node_ptr.external_count - 2;
  node_counter old_counter =
   ptr->count.load(std::memory_order_relaxed);
  node_counter new_counter;
  do {
   new_counter = old_counter;
   --new_counter.external_counters;                    ← (1)
   new_counter.internal_count += count_increase;       ← (2)
  }
  while (!ptr->count.compare_exchange_strong(          ← (3)
   old_counter, new_counter,
   std::memory_order_acquire, std::memory_order_relaxed));

  if (!new_counter.internal_count &&
      !new_counter.external_counters) {
   delete ptr;                                         ← (4)
  }
 }
};
```

Функция `free_external_counter()` дополняет `increase_external_count()`. Она аналогична эквивалентной функции из реализации `lock_free_stack::pop()` в листинге 7.11, но модифицировала с учетом появления поля `external_counters`. Она обновляет оба счетчика в одном вызове `compare_exchange_strong()` для всей структуры `count` **(3)** — точно так же мы поступали при уменьшении `internal_count` в `release_ref()`. Значение `internal_count` обновляется, как в листинге 7.11 **(2)**, а `external_counters` уменьшается на единицу **(1)**. Если теперь *оба* значения равны нулю, значит, ссылок на узел не осталось, поэтому его можно удалять **(4)**. Оба обновления необходимо выполнять в одном действии (потому и нужен цикл сравнения с обменом), чтобы избежать гонки. Если бы счетчики обновлялись порознь, то два разных потока могли бы решить, что владеют последней ссылкой на узел, и удалить его, что привело бы к неопределенному поведению.

Хотя теперь функция работает и свободна от блокировок, осталась еще одна проблема, касающаяся производительности. После того как один поток начал операцию `push()`, успешно выполнив `compare_exchange_strong()` от имени `old_tail.ptr->data` (точка **(5)** в

листинге 7.15), никакой другой войти в `push()` не может. Попытавшись это сделать, поток увидит новое значение, отличное от `nullptr`, в результате чего вызов `compare_exchange_strong()` вернет `false`, и потоку придется начать цикл заново. Это активное ожидание, которое только потребляет время процессора, не продвигаясь вперед ни на йоту. По сути дела, это блокировка. Первый удачный вызов `push()` блокирует все остальные потоки, пока не завершится, так что *этот код более не свободен от блокировок*. Хуже того — обычно операционная система может отдать приоритет потоку, удерживающему мьютекс, если существуют заблокированные потоки, но только не в данном случае, поэтому остальные потоки так и будут пожирать процессорное время, пока первый не закончит. И тут мы вытащим на свет очередной припасенный для освобождения от блокировок трюк: ожидающий поток может помочь потоку, который выполняет `push()`.

Освобождение от блокировок одного потока с помощью другого

Чтобы вновь сделать код свободным от блокировок, нам нужно придумать, как ожидающий поток может продвигаться вперед, даже если поток, находящийся в `push()`, застрял. Один из способов — помочь застрявшему потоку, выполнив за него часть работы.

В данном случае мы точно знаем, что нужно сделать: указатель `next` в хвостовом узле требуется установить на новый фиктивный узел, и тогда сам указатель `tail` можно будет обновить. Все фиктивные узлы эквивалентны, поэтому не имеет значения, какой из них использовать — созданный потоком, который успешно поместил в очередь данные, или потоком, ожидающим входа в `push()`. Если сделать указатель `next` в узле атомарным, то для его установки можно будет применить `compare_exchange_strong()`. После того как указатель `next` установлен, в цикле по `compare_exchange_weak()` можно будет установить `tail`, проверяя при этом, указывает ли он по-прежнему на тот же самый исходный узел. Если это не так, значит, узел обновил какой-то другой поток, так что можно прекратить попытки и перейти в начало цикла. Реализация этой идеи потребует также небольшого изменения `pop()`, где нужно будет загрузить указатель `next`; эта модификация показана в листинге ниже.

Листинг 7.20. Модификация `pop()` с целью помочь при выполнении `push()`

```
template<typename T>
class lock_free_queue {
private:
 struct node {
  std::atomic<T*> data;
  std::atomic<node_counter> count;
  std::atomic<counted_node_ptr> next;← (1)
 };

public:
 std::unique_ptr<T> pop() {
  counted_node_ptr old_head =
   head.load(std::memory_order_relaxed);
  for (;;) {
   increase_external_count(head, old_head);
   node* const ptr = old_head.ptr;
```

```
    if (ptr == tail.load().ptr) {
     return std::unique_ptr<T>();
    }
    counted_node_ptr next = ptr->next.load();← (2)
    if (head.compare_exchange_strong(old_head, next)) {
     T* const res = ptr->data.exchange(nullptr);
     free_external_counter(old_head);
     return std::unique_ptr<T>(res);
    }
    ptr->release_ref();
   }
  }
};
```

Как я и говорил, изменения тривиальны: указатель `next` теперь атомарный **(1)**, поэтому операция `load` в точке **(2)** атомарна. В данном примере мы используем упорядочение по умолчанию `memory_order_seq_cst`, поэтому явное обращение к `load()` можно было бы опустить, полагаясь на операцию загрузки в операторе неявного преобразования к типу `counted_node_ptr`, но явное обращение будет напоминать нам, куда впоследствии добавить явное задание порядка обращения к памяти.

Листинг 7.21. Пример реализации функции `push()`, освобождаемой от блокировок благодаря помощи извне

```
template<typename T>
class lock_free_queue {
private:
 void set_new_tail(counted_node_ptr &old_tail, ← (1)
                   counted_node_ptr const &new_tail) {
  node* const current_tail_ptr = old_tail.ptr;
  while (!tail.compare_exchange_weak(old_tail, new_tail) &&← (2)
         old_tail.ptr == current_tail_ptr);
  if (old_tail.ptr == current_tail_ptr)← (3)
   free_external_counter(old_tail);    ← (4)
  else
   current_tail_ptr->release_ref();    ← (5)
 }

public:
 void push(T new_value) {
  std::unique_ptr<T> new_data(new T(new_value));
  counted_node_ptr new_next;
  new_next.ptr = new node;
  new_next.external_count = 1;
  counted_node_ptr old_tail = tail.load();

  for (;;) {
   increase_external_count(tail, old_tail);
   T* old_data = nullptr;
   if (old_tail.ptr->data.compare_exchange_strong(  ← (6)
    old_data, new_data.get())) {
    counted_node_ptr old_next = {0};
    if (!old_tail.ptr->next.compare_exchange_strong(← (7)
        old_next, new_next)) {
```

```
        delete new_next.ptr;← (8)
        new_next = old_next;← (9)
       }
       set_new_tail(old_tail, new_next);
       new_data.release();
       break;
      } else {                ← (10)
       counted_node_ptr old_next = {0};
       if (old_tail.ptr->next.compare_exchange_strong(← (11)
           old_next, new_next)) {
        old_next = new_next;     ← (12)
        new_next.ptr = new node;← (13)
       }
       set_new_tail(old_tail, old_next);← (14)
      }
     }
    }
};
```

В целом похоже на исходную версию `push()` из листинга 7.15, но есть и несколько принципиальных отличий. Если указатель `data` *действительно* установлен **(6)**, то нужно обработать случай, когда нам помог другой поток, и кроме того появилась ветвь `else`, в которой один поток оказывает помощь другому **(10)**.

Установив указатель `data` в узле **(6)**, новая версия `push()` изменяет указатель `next`, вызывая `compare_exchange_strong()` **(7)**. Мы используем `compare_exchange_strong()`, чтобы избежать цикла. Если обмен завершился неудачно, значит, другой поток уже установил указатель `next`, поэтому нам ни к чему узел, выделенный в начале, и его можно удалить **(8)**. Мы также хотим использовать значение `next`, установленное другим потоком, для обновления `tail` **(9)**.

Собственно обновление указателя `tail` вынесено в отдельную функцию `set_new_tail()` **(1)**. В ней мы используем цикл по `compare_exchange_weak()` **(2)**, потому что если другие потоки пытаются поместить в очередь новый узел с помощью `push()`, то значение `external_count` может измениться, а нам не хотелось бы его потерять. Однако нужно позаботиться и о том, чтобы не затереть значение, которое другой поток уже успешно изменил, в противном случае в очереди могут возникнуть циклы, а это уже совершенно лишнее. Следовательно, нужно гарантировать, что часть `ptr` загруженного значения осталась той же самой, если сравнение с обменом не прошло. Если `ptr` не изменился после выхода из цикла **(3)**, то мы успешно установили `tail`, поэтому старый внешний счетчик нужно освободить **(4)**. Если значение `ptr` изменилось, то счетчик уже освобождён другим потоком, поэтому нам нужно только освободить ссылку, которую удерживает наш поток **(5)**.

Если поток, вызвавший `push()`, не сумел установить указатель `data` на этой итерации цикла, то он может помочь более удачливому потоку завершить обновление. Сначала мы пытаемся записать в next указатель на новый узел, выделенный в этом потоке **(11)**. Если это получилось, то выделенный нами узел будет использоваться как новый узел `tail` **(12)**, а нам следует выделить еще один узел, поскольку поместить свои данные в очередь еще только предстоит **(13)**. После этого мы можем попытаться установить узел `tail`, вызвав `set_new_tail` до перехода к очередной итерации **(14)**.

Вероятно, вы обратили внимание на чрезмерно большое для такого крохотного фрагмента количество `new` и `delete`. Вызвало это тем, что новые узлы создаются в `push()`, а уничтожаются в `pop()`. Поэтому быстродействие этого кода существенно зависит от того, насколько эффективно работает распределитель памяти; плохой распределитель может полностью свести на нет свойства масштабируемости, присущие свободному от блокировок контейнеру. Вопрос о выборе и реализации подобных распределителей выходит за рамки данной книги, но имейте в виду, что единственный способ узнать, какой распределитель лучше, — испытывать и замерять производительность. К числу стандартных приемов оптимизации выделения памяти можно отнести создание отдельного распределителя в каждом потоке и использование списка свободных узлов — освободившиеся узлы помещаются в этот список, а не возвращаются распределителю.

Ну и, пожалуй, хватит примеров. Давайте теперь на основе вышеизложенного сформулируем ряд рекомендаций по написанию структур данных, свободных от блокировок.

7.3. Рекомендации по написанию структур данных без блокировок

Если вы тщательно прорабатывали все приведенные в этой главе примеры, то, наверное, оцепили, как трудно правильно написать код без блокировок. Если вы собираетесь проектировать собственные структуры данных, то не лишне будет иметь под рукой рекомендации, на которые можно опираться. Общие советы, относящиеся к параллельным структурам данных, приведенные в начале главы 6, остаются в силе, но этого мало. Из анализа примеров я извлек несколько полезных рекомендаций, к которым вы можете обращаться при проектировании структур данных, свободных от блокировок.

7.3.1. Используйте `std::memory_order_seq_cst` для создания прототипа

Порядок доступа к памяти `std::memory_order_seq_cst` гораздо проще для понимания и анализа, чем любой другой, потому что операции с такой семантикой полностью упорядочены. Во всех примерах из этой главы мы начинали с упорядочения `std::memory_order_seq_cst` и только потом ослабляли ограничения. В этом смысле использование других способов упорядочения можно считать *оптимизацией*, которой не следует заниматься преждевременно. В общем случае, для того чтоб определить, какие операции можно ослабить, нужно иметь перед глазами полный код, правильно работающий со структурой данных. Любая попытка пойти другим путем только осложнит вам жизнь. Проблема еще и в том, что тестирование кода может не выявить ошибок, но это еще не гарантирует их отсутствия. Если нет верификатора алгоритма, способного систематически тестировать все возможные сочетания видимости в разных потоках, которые совместимы с гарантиями, предоставляемыми заданными режимами упорядочения доступа к памяти (а такие программы существуют), то одного лишь прогона кода недостаточно.

7.3.2. Используйте подходящую схему освобождения памяти

Одна из самых сложных сторон написания свободного от блокировок кода — управление памятью. С одной стороны, требуется любой ценой предотвратить удаление объектов, на которые могут ссылаться другие потоки, а, с другой, удалять объекты как можно раньше, чтобы избежать чрезмерного расхода памяти. В этой главе мы познакомились с тремя методами, обеспечивающими безопасное освобождение памяти.

• Дождаться, когда к структуре данных не будет обращаться ни один поток, и удалить разом все объекты, ожидающие удаления.

• Использовать указатели опасности для выявления потока, обращающегося к конкретному объекту.

• Подсчитывать ссылки на объекты и не удалять их, пока ссылки существуют.

В любом случае идея заключается в том, чтобы каким-то образом отслеживать, сколько потоков обращается к объекту, и удалять его лишь в том случае, когда на него не осталось ни одной ссылки. Существует много методов освобождения памяти в структурах данных, свободных от блокировок. В частности, это идеальная ситуация для применения сборщика

мусора. Придумывать алгоритмы гораздо проще, если знаешь, что сборщик мусора удалит узлы, когда они больше не используются, но не раньше.

Другой способ — использовать узлы повторно и полностью освобождать их только тогда, когда уничтожается вся структура данных. Раз узлы используются повторно, то память никогда не становится недействительной, поэтому некоторые проблемы, сопряженные с неопределенным поведением, вообще не возникают. Недостаток же в том, что взамен появляется так называемая *проблема ABA*.

7.3.3. Помните о проблеме ABA

Проблема ABA свойственна любому алгоритму, основанному на сравнении с обменом. Проявляется она следующим образом.

1. Поток 1 читает атомарную переменную `x` и обнаруживает, что она имеет значение `A`.
2. Поток 1 выполняет некоторую операцию, исходя из этого значения, например разыменовывает его (если это указатель), выполняет поиск или делает еще что-то.
3. Операционная система приостанавливает поток 1.
4. Другой поток выполняет некоторые операции с `x`, в результате которых ее значение изменяется и становится равным `B`.
5. Затем этот поток изменяет данные, ассоциированные со значением `A`, после чего значение, хранящееся в потоке 1, оказывается недействительным. Это может быть нечто кардинальное, например освобождение памяти, адресуемой указателем, или просто изменение какого-то ассоциированного значения.
6. Далее поток снова изменяет значение `x` на `A`, но уже с новыми данными. В случае указателя это может быть новый объект, который но случайному стечению обстоятельства имеет тот же адрес, что прежний.
7. Поток 1 возобновляется и выполняет сравнение с обменом для переменной `x`, сравнивая ее значение с `A`. Операция завершается успешно (потому что значение действительно равно `A`), но *это уже не то* `A`. Данные, ранее прочитанные потоком на шаге 2, более не действительны, но поток 1 ничего об этом не знает и повреждает структуру данных.

Ни один из представленных в этой главе алгоритмов не страдает от этой проблемы, но совсем нетрудно написать свободный от блокировок алгоритм, в котором она проявится. Самый распространенный способ избежать ее — хранить вместе с переменной `x` счетчик ABA. Операция сравнения с обменом тогда производится над комбинацией `x` и счетчика, как над единым целым. Всякий раз, как значение заменяется, счетчик увеличивается, поэтому даже если окончательное значение `x` не изменилось, сравнение с обменом не пройдёт, если другой поток в промежутке изменял `x`.

Проблема ABA особенно часто встречается в алгоритмах, в которых используются списки свободных узлов или иные способы повторного использования узлов вместо возврата их распределителю.

7.3.4. Выявляйте циклы активного ожидания и помогайте другим потокам

В последнем примере очереди мы видели, что поток, выполняющий операцию `push()`, должен дождаться, пока другой поток, выполняющий ту же операцию, завершит свои

действия. Если ничего не предпринимать, то этот поток будет крутиться в цикле активного ожидания, впустую расходуя процессорное время и не продвигаясь ни на шаг. Наличие цикла ожидания в действительности эквивалентно блокирующей операции, а тогда с равным успехом можно было бы воспользоваться мьютексами. Модифицировав алгоритм таким образом, что ожидающий поток выполняет неполные шаги, если планировщик выделил ему время до завершения работы в исходном потоке, мы сможем устранить активное ожидание и сделать операцию неблокирующей. В примере очереди нам для этого потребовалось сделать одну переменную-член атомарной и использовать для ее установки операции сравнения с обменом, но в более сложных структурах данных могут понадобиться более обширные изменения.

Отталкиваясь от структур данных с блокировками, описанных в главе 6, мы в этой главе продемонстрировали простые реализации структур данных без блокировок на примере все тех же стека и очереди. Мы видели, как внимательно нужно подходить к упорядочению доступа к памяти в атомарных операциях, чтобы избежать гонок и гарантировать, что каждый поток видит непротиворечивое представление структуры данных. Мы также поняли, что управление памятью в структурах данных без блокировок оказывается значительно сложнее, чем в структурах с блокировками, и изучили несколько подходов к решению этой проблемы. Еще мы узнали, как предотвращать циклы активного ожидания, помогая завершить работу потоку, которого мы ждем.

Проектирование свободных от блокировок структур данных — сложная задача, при решении которой легко допустить ошибки, зато такие структуры обладают свойствами масштабируемости, незаменимыми в некоторых ситуациях. Надеюсь, что, проработав приведенные в этой главе примеры и ознакомившись с рекомендациями, вы будете лучше подготовлены к разработке собственных структур данных без блокировок, реализации алгоритмов, описанных в научных статьях, или к поиску ошибок, допущенных бывшим сотрудником компании.

Во всех случаях, когда некоторые данные разделяются между потоками, нужно задумываться о применяемых структурах данных и о синхронизации. Проектируя структуры данных с учетом параллелизма, вы сможете инкапсулировать ответственность в самой структуре, позволив основной программе сосредоточиться на решаемой задаче, а не на синхронизации доступа к данным. Этот подход будет продемонстрирован в главе 8, где мы перейдём от параллельных структур данных к написанию параллельных программ. В параллельных алгоритмах для повышения производительности используется несколько потоков, и выбор подходящей параллельной структуры данных приобретает решающее значение.

Глава 8.
Проектирование параллельных программ

В этой главе:

- Методы распределения данных между потоками.
- Факторы, влияющие на производительность параллельного кода.
- Как от этих факторов зависит дизайн параллельных структур данных.
- Безопасность многопоточного кода относительно исключений.
- Масштабируемость.
- Примеры реализации параллельных алгоритмов.

В предыдущих главах мы в основном занимались появившимися в новом стандарте C++11 средствами для написания параллельных программ. В главах 6 и 7 мы видели, как эти средства применяются для проектирования простых структур данных, безопасных относительно доступа из нескольких потоков. Но как столяру недостаточно знать, как устроена петля или шарнир, чтобы смастерить шкаф или стол, так и для проектирования параллельных программ недостаточно знакомства с устройством и применением простых структур данных. Теперь мы будем рассматривать проблему в более широком контексте и научимся строить более крупные узлы, способные выполнять полезную работу В качестве примеров я возьму многопоточные реализации некоторых алгоритмов из стандартной библиотеки C++, но те же самые принципы остаются в силе при разработке всех уровней приложения.

Как и в любом программном проекте, тщательное продумывание структуры параллельного кода имеет первостепенное значение. Однако в параллельной программе приходится учитывать еще больше факторов, чем в последовательной. Нужно принимать во внимание не только инкапсуляцию, связанность и сцепленность (эти вопросы подробно рассматриваются во многих книгах по проектированию программного обеспечения), но и то, какие данные разделять, как организовывать доступ к ним, каким потокам придётся ждать других потоков для завершения операции и т.д.

Именно этим мы и будем заниматься в этой главе, начав с высокоуровневых (но фундаментальных) вопросов о том, сколько создавать потоков, какой код выполнять в каждом потоке и как многопоточность влияет на понятность программы, и закончив низкоуровневыми деталями того, как организовать разделяемые данные для достижения оптимальной производительности.

Начнем с методов распределения работы между потоками.

8.1. Методы распределения работы между потоками

Представьте, что вам поручили построить дом. Для этого предстоит вырыть котлован под фундамент, возвести стены, провести водопровод и электропроводку и т.д. Теоретически, имея достаточную подготовку, все это можно сделать и в одиночку, по времени уйдет много и придётся постоянно переключаться с одного занятия на другое. Можно вместо этого нанять несколько помощников. Тогда нужно решить, сколько людей нанимать и каких специальностей. К примеру, пригласить двух универсалов и всё делать совместно. По-прежнему нужно будет переходить от одной работы к другой, но в итоге строительство удастся завершить быстрее, так как теперь в нем принимает участие больше народу

Есть и другой путь — нанять бригаду узких специалистов: каменщика, плотника, электрика, водопроводчика и других. Специалисты делают то, что лучше всего умеют, так что если прокладывать трубы не надо, то водопроводчик попивает чаёк. Работа все равно продвигается быстрее, так как занято больше людей, — водопроводчик может заниматься туалетом, пока электрик делает проводку на кухне, но зато когда для конкретного специалиста работы нет, он простаивает. Впрочем, несмотря на простои, специалисты, скорее всего, справятся быстрее, чем бригада мастеров на все руки. Им не нужно менять инструменты, да и свое дело они по идее должны делать быстрее, чем универсалы. Так ли это на самом деле, зависит от разных обстоятельств — нет другого пути, как взять и попробовать.

Но и специалистов можно нанять много или мало. Наверное, имеет смысл нанимать больше каменщиков, чем электриков. Кроме того, состав бригады и ее общая эффективность могут измениться, если предстоит построить несколько домов. Если в одном доме у водопроводчика мало работы, то на строительстве нескольких его можно занять полностью. Наконец, если вы не обязаны платить специалистам за простой, то можно позволить себе нанять больше людей, пусть даже в каждый момент времени работать будет столько же, сколько раньше.

Но хватит уже о строительстве, какое отношение всё это имеет к потокам? Да просто к ним применимы те же соображения. Нужно решить, сколько использовать потоков и какие задачи поручить каждому Должны ли потоки быть «универсалами», берущимися за любую подвернувшуюся работу или «специалистами», которые умеют хорошо делать одно дело? Или, быть может, нужно сочетание того и другого? Эти решения необходимо принимать вне зависимости от причин распараллеливания программы и от того, насколько они будут удачны, существенно зависит производительность и ясность кода. Поэтому так важно понимать, какие имеются варианты; тогда решения о структуре приложения будут опираться на знания, а не браться с потолка. В этом разделе мы рассмотрим несколько методов распределения задач и начнем с распределения данных между потоками.

8.1.1. Распределение данных между потоками до начала обработки

Легче всего распараллеливаются такие простые алгоритмы, как `std::for_each`, которые производят некоторую операцию над каждым элементом набора данных. Чтобы распараллелить такой алгоритм, можно назначить каждому элементу один из обрабатывающих потоков. Каким образом распределить элементы для достижения

оптимальной производительности, зависит от деталей структуры данных, как мы убедимся ниже в этой главе.

Простейший способ распределения данных заключается в том, чтобы назначить первые *N* элементов одному потоку, следующие *N* — другому и так далее, как показано на рис. 8.1, хотя это и не единственное решение. Как бы ни были распределены данные, каждый поток обрабатывает только назначенные ему элементы, никак не взаимодействуя с другими потоками до тех пор, пока не завершит обработку.

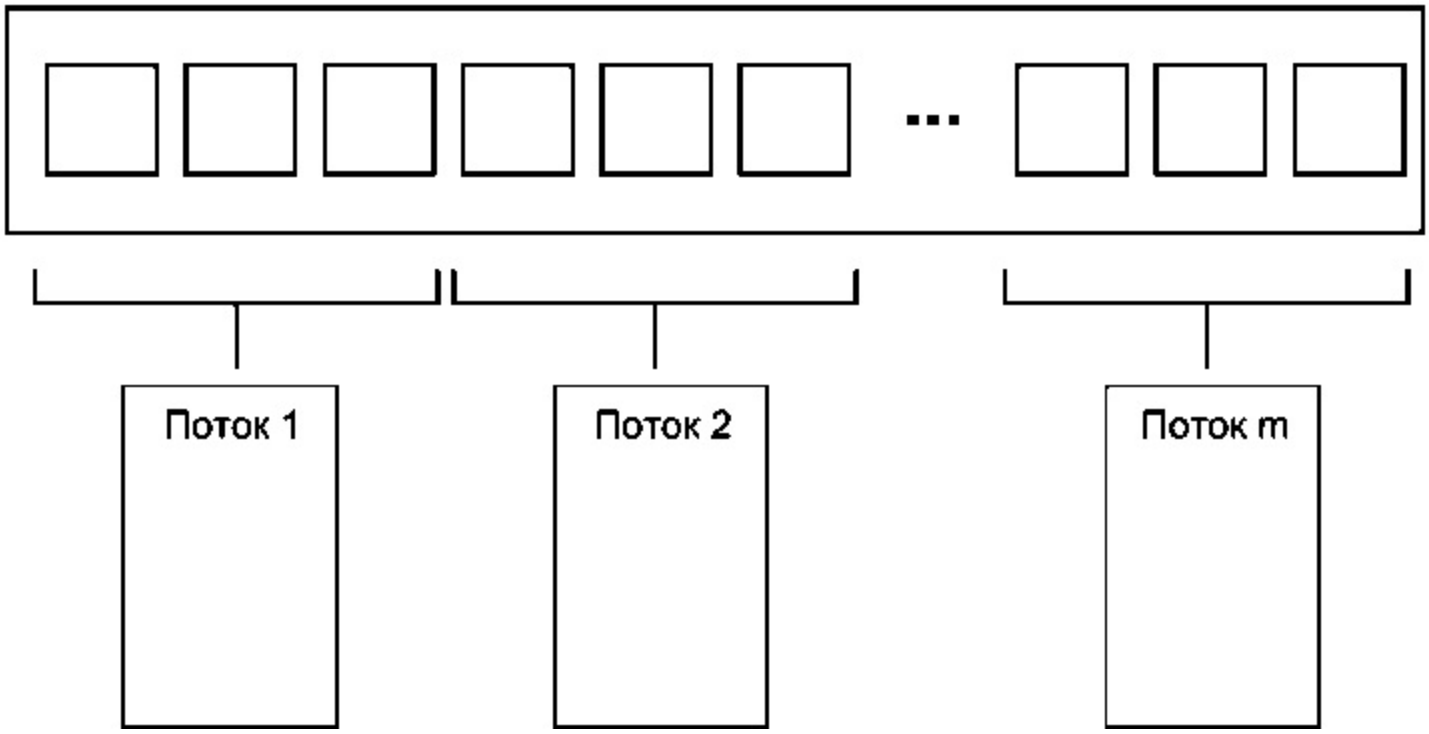

Рис. 8.1. Распределение последовательных блоков данных между потоками

Такая организация программы знакома каждому, кто программировал в системах Message Passing Interface (MPI)[17] или OpenMP[18]: задача разбивается на множество параллельных подзадач, рабочие потоки выполняют их параллельно, а затем результаты объединяются на стадии *редукции*. Этот подход применён в примере функции `accumulate` из раздела 2.4; в данном случае и параллельные задачи, и финальный шаг редукции представляют собой аккумулирование. Для простого алгоритма `for_each` финальный шаг отсутствует, так как нечего редуцировать.

Понимание того, что последний шаг является редукцией, важно; в наивной реализации, представленной в листинге 2.8, финальная редукция выполняется как последовательная операция. Однако часто и ее можно распараллелить; операция `accumulate` по сути дела *сама* является редукцией, поэтому функцию из листинга 2.8 можно модифицировать таким образом, чтобы она вызывала себя рекурсивно, если, например, число потоков больше минимального количества элементов, обрабатываемых одним потоком. Или запрограммировать рабочие потоки так, чтобы по завершении задачи они выполняли некоторые шаги редукции, а не запускать каждый раз новые потоки.

Хотя это действенная техника, применима она не в любой ситуации. Иногда данные не удается заранее распределить равномерно, а как это сделать, становится понятно только в процессе обработки. Особенно наглядно это проявляется в таких рекурсивных алгоритмах, как Quicksort; здесь нужен другой подход.

8.1.2. Рекурсивное распределение данных

Алгоритм Quicksort состоит из двух шагов: разбиение данных на две части — до и после опорного элемента в смысле требуемого упорядочения, и рекурсивная сортировка обеих «половин». Невозможно распараллелить этот алгоритм, разбив данные заранее, потому что состав каждой «половины» становится известен только в процессе обработки элементов. Поэтому распараллеливание по необходимости должно быть рекурсивным. На каждом уровне рекурсии производится *больше* вызовов функции `quick_sort`, потому что предстоит отсортировать элементы, меньшие опорного, и большие опорного. Эти рекурсивные вызовы не зависят друг от друга, так как обращаются к разным элементам, поэтому являются идеальными кандидатами для параллельного выполнения. На рис. 8.2 изображено такое рекурсивное разбиение.

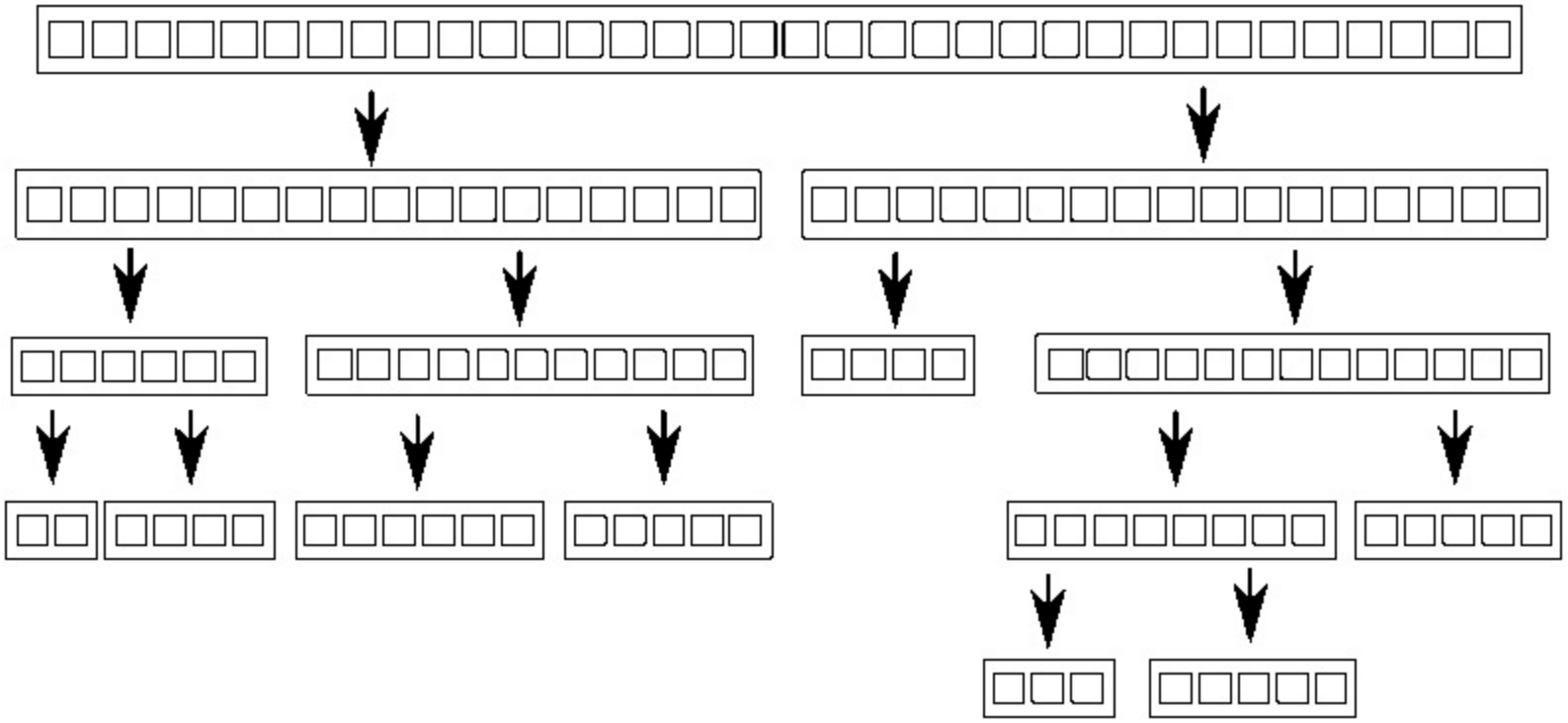

Рис. 8.2. Рекурсивное разбиение данных

В главе 4 была приведена подобная реализация. Вместо того чтобы просто вызывать функцию рекурсивно для большей и меньшей «половины», мы с помощью `std::async()` на каждом шаге запускали асинхронную задачу для меньшей половины. Вызывая `std::async()`, мы просим стандартную библиотеку C++ самостоятельно решить, имеет ли смысл действительно выполнять задачу в новом потоке или лучше сделать это синхронно.

Это существенно: при сортировке большого набора данных запуск нового потока для каждого рекурсивного вызова быстро приводит к образованию чрезмерного количества потоков. Как мы увидим ниже, когда потоков слишком много, производительность может не возрасти, а наоборот *упасть*. Кроме того, если набор данных очень велик, то потоков может просто не хватить. Сама идея рекурсивного разбиения на задачи хороша, нужно только строже контролировать число запущенных потоков. В простых случаях с этим справляется `std::async()`, но есть и другие варианты.

Один из них — воспользоваться функцией `std::thread::hardware_concurrency()` для выбора нужного числа потоков, как мы делали в параллельной версии `accumulate()` из листинга 2.8. Тогда вместо того чтобы запускать новый поток для каждого рекурсивного вызова, мы можем просто поместить подлежащий сортировке блок данных в потокобезопасный стек типа того, что был описан в главах 6 и 7. Если потоку больше нечего делать — потому что он закончил обработку всех своих блоков или потому что ждет, когда

необходимый ему блок будет отсортирован, то он может взять блок из стека и заняться его сортировкой.

В следующем листинге показана реализация этой идеи.

Листинг 8.1. Параллельный алгоритм Quicksort с применением стека блоков, ожидающих сортировки

```
template<typename T>
struct sorter { ← (1)
 struct chunk_to_sort {
  std::list<T> data;
  std::promise<std::list<T> > promise;
 };

 thread_safe_stack<chunk_to_sort> chunks; ← (2)
 std::vector<std::thread> threads;        ← (3)
 unsigned const max_thread_count;
 std::atomic<bool> end_of_data;

 sorter():
  max_thread_count(std::thread::hardware_concurrency() - 1),
  end_of_data(false) {}

 ~sorter() {          ← (4)
  end_of_data = true; ← (5)

  for (unsigned i = 0; i < threads.size(); ++i) {
   threads[i].join(); ← (6)
  }
 }

 void try_sort_chunk() {
  boost::shared_ptr<chunk_to_sort> chunk = chunks.pop();← (7)
  if (chunk) {
   sort_chunk(chunk); ← (8)
  }
 }

 std::list<T> do_sort(std::list<T>& chunk_data) { ← (9)
  if (chunk_data.empty()) {
   return chunk_data;
  }

  std::list<T> result;
  result.splice(result.begin(), chunk_data, chunk_data.begin());
  T const& partition_val = *result.begin();

  typename std::list<T>::iterator divide_point = ← (10)
   std::partition(chunk_data.begin(), chunk_data.end(),
    [&](T const& val) {return val < partition_val; });
  chunk_to_sort new_lower_chunk;
```

```
    new_lower_chunk.data.splice(new_lower_chunk.data.end(),
     chunk_data, chunk_data.begin(),
     divide_point);
    std::future<std::list<T> > new_lower =
     new_lower_chunk.promise.get_future();
    chunks.push(std::move(new_lower_chunk)); ← (11)
    if (threads.size() < max_thread_count) { ← (12)
     threads.push_back(std::thread(&sorter<T>::sort_thread, this));
    }

    std::list<T> new_higher(do_sort(chunk_data));

    result.splice(result.end(), new_higher);
    while (new_lower.wait_for(std::chrono::seconds(0)) !=
     std::future_status::ready) { ← (13)
     try_sort_chunk();            ← (14)
    }

    result.splice(result.begin(), new_lower.get());
    return result;
   }

   void sort_chunk(boost::shared_ptr<chunk_to_sort> const& chunk) {
    chunk->promise.set_value(do_sort(chunk->data));← (15)
   }

   void sort_thread() {
    while (!end_of_data) {     ← (16)
     try_sort_chunk();         ← (17)
     std::this_thread::yield();← (18)
    }
   }
  };

  template<typename T>
  std::list<T> parallel_quick_sort(std::list<T> input) { ← (19)
   if (input.empty()) {
    return input;
   }
   sorter<T> s;

   return s.do_sort(input); ← (20)
  }
```

Здесь функция `parallel_quick_sort` **(19)** делегирует большую часть работы классу `sorter` **(1)**, который объединяет стек неотсортированных блоков **(2)** с множеством потоков **(3)**. Основные действия производятся в функции-члене `do_sort` **(9)**, которая занимается обычным разбиением данных на две части **(10)**. Но на этот раз она не запускает новый поток для каждого блока, а помещает его в стек **(11)** и запускает новый поток, только если еще есть незанятые процессоры **(12)**. Поскольку меньший блок, возможно, обрабатывается другим потоком, нам необходимо дождаться его готовности **(13)**. Чтобы не простаивать зря (в том

случае, когда данный поток единственный или все остальные уже заняты), мы пытаемся обработать блок, находящийся в стеке **(14)**. Функция `try_sort_chunk` извлекает поток из стека **(7)**, сортирует его **(8)** и сохраняет результаты в обещании `promise`, так чтобы их смог получить поток, который поместил в стек данный блок **(15)**.

Запущенные потоки крутятся в цикле, пытаясь отсортировать блоки, находящиеся в стеке **(17)**, ожидая, пока будет установлен флаг `end_of_data` **(16)**. В промежутке между проверками они уступают процессор другим потокам **(18)**, чтобы у тех был шанс поместить в стек новые блоки. Эта программа полагается на то, что деструктор класса `sorter` **(4)** разберется с запущенными потоками. Когда все данные будут отсортированы, `do_sort` вернет управление (хотя рабочие потоки все еще выполняются), так что главный поток вернется из `parallel_quick_sort` **(20)** и, стало быть, уничтожит объект `sorter`. В этот момент деструктор поднимет флаг `end_of_data` **(5)** и дождется завершения рабочих потоков **(6)**. Поднятый флаг является для функции потока указанием выйти из цикла **(16)**.

При таком подходе не возникает проблемы неограниченного количества потоков, с которой мы сталкивались, когда `spawn_task` каждый раз запускает новый поток. С другой стороны, мы не полагаемся на то, что стандартная библиотека C++ выберет количество рабочих потоков за нас, как то происходит при использовании `std::async()`. Мы сами ограничиваем число потоков значением `std::thread::hardware_concurrency()`, чтобы избежать чрезмерно большого количества контекстных переключений. Однако же взамен нас поджидает другая потенциальная проблема: управление потоками и взаимодействие между ними сильно усложняют код. Кроме того, хотя потоки и обрабатывают разные элементы данных, все они обращаются к стеку для добавления новых блоков и извлечения блоков для сортировки. Из-за этой острой конкуренции производительность может снизиться, пусть даже мы используем свободный от блокировок (и, значит, неблокирующий) стек. Почему так происходит, мы увидим чуть ниже.

Этот подход представляет собой специализированную версию *пула потоков* — существует набор потоков, которые получают задачи из списка ожидающих работ, выполняют их, а затем снова обращаются к списку. Некоторые потенциальные проблемы, свойственные пулам потоков (в том числе конкуренция за список работ), и способы их решения рассматриваются в главе 9. Задача масштабирования приложения на несколько процессоров более детально обсуждается в этой главе ниже (см. раздел 8.2.1).

Как при предварительном, так и при рекурсивном распределении данных мы предполагаем, что сами данные фиксированы заранее, а нам нужно лишь найти удачный способ их разбиения. Но так бывает не всегда; если данные порождаются динамически или поступают из внешнего источника, то такой подход не годится. В этом случае имеет смысл распределять работу по типам задач, а не по данным.

8.1.3. Распределение работы по типам задач

Распределение работы между потоками путем назначения каждому потоку блока данных (заранее или рекурсивно во время обработки) все же опирается на предположение о том, что все потоки производят одни те же действия с данными. Альтернативный подход — заводить специализированные потоки, каждый из которых решает отдельную задачу — как водопроводчики и электрики на стройке. Потоки могут даже работать с одними и теми же

данными, но обрабатывать их по-разному.

Такой способ распределения работы — следствие распараллеливания ради разделения обязанностей; у каждого потока есть своя задача, которую он решает независимо от остальных. Время от времени другие потоки могут поставлять нашему потоку данные или генерировать события, на которые он должен отреагировать, но в общем случае каждый поток занимается тем, для чего предназначен. В принципе, это хороший дизайн — у каждой части кода есть своя зона ответственности.

Распределение работы по типам задач с целью разделения обязанностей

Однопоточное приложение должно как-то разрешать противоречия с принципом единственной обязанности, если существует несколько задач, которые непрерывно выполняются в течение длительного промежутка времени, или если приложение должно обрабатывать поступающие события (например, нажатия на клавиши или входящие сетевые пакеты) своевременно, несмотря на наличие других задач. Для решения этой проблемы мы обычную вручную пишем код, который выполняет кусочек задачи A, потом кусочек задачи B, потом проверяет, не нажата ли клавиша и не пришёл ли сетевой пакет, а потом возвращается в начало цикла, чтобы выполнить следующий кусочек задачи A. В результате код задачи A оказывается усложнен из-за необходимости сохранять состояние и периодически возвращать управление главному циклу. Если выполнять в этом цикле чрезмерно много задач, то скорость работы упадёт, а пользователю придётся слишком долго ждать реакции на нажатие клавиши. Уверен, все вы встречались с крайними проявлениями этого феномена не в одном так в другом приложении: вы просите программу выполнить какое-то действие, после чего интерфейс вообще перестаёт реагировать, пока оно не завершится.

Тут-то и приходят на выручку потоки. Если выполнять каждую задачу в отдельном потоке, то всю эту работу возьмет на себя операционная система. В коде для решения задачи A вы можете сосредоточить внимание только на этой задаче и не думать о сохранении состояния, возврате в главный цикл и о том, сколько вам понадобится времени. Операционная система автоматически сохранит состояние и в подходящий момент переключится на задачу B или C, а если система оснащена несколькими процессорами или ядрами, то задачи A и B могут даже выполняться действительно параллельно. Код обработки клавиш или сетевых пакетов теперь будет работать без задержек, и все остаются в выигрыше: пользователь своевременно получает отклик от программы, а разработчик может писать более простой код, так как каждый поток занимается только тем, что входит в его непосредственные обязанности, не отвлекаясь на управление порядком выполнения или на взаимодействие с пользователем.

Звучит, как голубая мечта. Да возможно ли такое? Как всегда, дьявол кроется в деталях. Если всё действительно независимо и потокам не нужно общаться между собой, то это и вправду легко. Увы, мир редко бывает таким идеальным. Эти замечательные фоновые потоки часто выполняют какие-то запросы пользователя и должны уведомлять пользователя о завершении, обновляя графический интерфейс. А то еще пользователь вздумает отменить задачу, и тогда интерфейс должен каким-то образом послать потоку сообщение с требованием остановиться. В обоих случаях необходимо все тщательно продумать и правильно синхронизировать, хотя обязанности таки разделены. Поток пользовательского

интерфейса занимается только интерфейсом, но должен обновлять его по требованию других потоков. Аналогично поток, занятый фоновой задачей, выполняет только операции, свойственные данной задаче, но не исключено, что одна из этих обязанностей заключается в том, чтобы остановиться по просьбе другого потока. Потоку все равно, откуда поступил запрос, важно лишь, что он адресован ему и относится к его компетенции.

На пути разделения обязанностей между потоками нас подстерегают две серьезные опасности. Во-первых, мы можем *неправильно* разделить обязанности. Признаками такой ошибки является большой объем разделяемых данных или положение, при котором потоки должны ждать друг друга; то и другое означает, что взаимодействие между потоками избыточно интенсивно. Когда такое происходит, нужно понять, чем вызвана необходимость взаимодействия. Если все сводится к какой-то одной причине, то, быть может, соответствующую обязанность следует поручить отдельному потоку, освободив от нее всех остальных. Наоборот, если два потока много взаимодействуют друг с другом и мало — с остальными, то, возможно, имеет смысл объединить их в один.

Распределяя задачи по типам, не обязательно полностью изолировать их друг от друга. Если к нескольким наборам входных данных требуется применять одну *последовательность* операций, то работу можно разделить так, что каждый поток будет выполнять какой-то один этап этой последовательности.

Распределение последовательности задач между потоками

Если задача заключается в применении одной последовательности операций ко многим независимым элементам данных, то можно организовать распараллеленный *конвейер*. Здесь можно провести аналогию с физическим конвейером: данные поступают с одного конца, подвергаются ряду операций и выходят с другого конца.

Чтобы распределить работу таким образом, следует создать отдельный поток для каждого участка конвейера, то есть для каждой операции. По завершении операции элемент данных помещается в очередь, откуда его забирает следующий поток. В результате поток, выполняющий первую операцию, сможет приступить к обработке следующего элемента, пока второй поток трудится над первым элементом.

Такая альтернатива стратегии распределения данных между потоками, которая была описана в разделе 8.1.1, хорошо приспособлена для случаев, когда входные данные вообще неизвестны до начала операции. Например, данные могут поступать по сети или первой в последовательности операцией может быть просмотр файловой системы на предмет нахождения подлежащих обработке файлов.

Конвейеры хороши также тогда, когда каждая операция занимает много времени; распределяя между потоками задачи, а не данные, мы изменяем качественные показатели производительности. Предположим, что нужно обработать 20 элементов данных на машине с четырьмя ядрами, и обработка каждого элемента состоит из четырех шагов, по 3 секунды каждый. Если распределить данные между потоками, то каждому потоку нужно будет обработать 5 элементов. В предположении, что нет никаких других операций, которые могли бы повлиять на хронометраж, по истечении 12 секунд будет обработано четыре элемента, по истечении 24 секунд — 8 элементов и т.д. На обработку всех 20 элементов уйдет 1 минута. Если организовать конвейер, ситуация изменится. Четыре шага можно распределить между

четырьмя ядрами. Теперь первый элемент будет обрабатываться каждым из четырех ядер, и в целом на это уйдет 12 секунд — как и раньше. На самом деле, по истечении 12 секунд в нас обработан всего один элемент, что хуже, чем при распределении данных. Однако после того как конвейер *запущен*, работа идет немного по-другому; первое ядро, обработав первый элемент, переходит ко второму. Поэтому, когда последнее ядро закончит обработку первого элемента, второй уже будет наготове. В результате каждые три секунды на выходе получается очередной обработанный элемент, тогда как раньше элементы выходили пачками по четыре каждые 12 секунд.

Суммарное время обработки всего пакета может увеличиться, потому что придётся подождать 9 секунд, пока первый элемент доберется до последнего ядра. Но более плавная обработка в некоторых случаях предпочтительнее. Возьмем, к примеру, систему воспроизведения цифрового видео высокой четкости. Чтобы фильм можно было смотреть без напряжения, частота кадров должна быть не менее 25 в секунду, а лучше — больше. Кроме того, временные промежутки между кадрами должны быть одинаковы для создания иллюзии непрерывного движения. Приложение, способное декодировать 100 кадров секунду, никому не нужно, если оно секунду ждет, потом «выплевывает» сразу 100 кадров, потом еще секунду ждет и снова выдает 100 кадров. С другой стороны, зритель не будет возражать против двухсекундной задержки *перед началом* просмотра. В такой ситуации распараллеливание с помощью конвейера, позволяющее выводить кадры с постоянной скоростью, безусловно, предпочтительнее.

Познакомившись с различными способами распределения работы между потоками, посмотрим теперь, какие факторы влияют на производительность многопоточной системы и как от них зависит выбор решения.

8.2. Факторы, влияющие на производительность параллельного кода

Если вы распараллеливаете программу, чтобы повысить ее быстродействие в системе с несколькими процессорами, то должны знать о том, какие факторы вообще влияют на производительность. Но и в том случае, когда потоки применяются просто для разделения обязанностей, нужно позаботиться о том, чтобы многопоточность не привела к снижению быстродействия. Пользователи не скажут спасибо, если приложение будет работать на новенькой 16-ядерной машине *медленнее*, чем на старой одноядерной.

Как мы скоро увидим, на производительность многопоточного кода оказывают влияние многие факторы и даже решение о том, *какой* элемент данных в каком потоке обрабатывать (при прочих равных условиях) может кардинально изменить скорость работы программы. А теперь без долгих слов приступим к изучению этих факторов и начнем с самого очевидного: сколько процессоров имеется в системе?

8.2.1. Сколько процессоров?

Количество (и конфигурация) процессоров — первый из существенных факторов, влияющих на производительность многопоточного приложения. Иногда вы точно знаете о том, на каком оборудовании будет работать программа и можете учесть это при проектировании, произведя реальные измерения на целевой системе или ее точной копии. Если так, то вам крупно повезло; как правило, разработчик лишен такой роскоши. Быть может, программа пишется на *похожей* системе, но различия могут оказаться весьма значимыми. Например, вы разрабатывали на двух- или четырехъядерной машине, а у заказчика один многоядерный процессор (с произвольным числом ядер), или несколько одноядерных или даже несколько многоядерных процессоров. Поведение и характеристики производительности программы могут существенно зависеть от таких деталей, поэтому нужно заранее продумывать возможные последствия и тестировать в максимально разнообразных конфигурациях.

В первом приближении один 16-ядерный процессор эквивалентен четырем 4-ядерным или 16 одноядерным, во всех случаях одновременно могут выполняться 16 потоков. Чтобы в полной мере задействовать имеющийся параллелизм, в программе должно быть не менее 16 потоков. Если их меньше, то вычислительная мощность используется не полностью (пока оставляем за скобками тот факт, что могут работать и другие приложения). С другой стороны, если готовых к работе (не заблокированных в ожидании чего-то) потоков больше 16, то приложение будет попусту растрачивать процессорное время на контекстное переключение, о чем мы уже говорили в главе 1. Такая ситуация называется *превышением лимита* (oversubscription).

Чтобы приложение могло согласовать количество потоков с возможностями оборудования, в стандартной библиотеке Thread Library имеется функция `std::thread::hardware_concurrency()`. Мы уже видели, как ее можно использовать для определения подходящего количества потоков.

Использовать `std::thread::hardware_concurrency()` напрямую следует с осторожностью; ваш код не знает, какие еще потоки работают в программе, если только вы

не сделали эту информацию разделяемой. В худшем случае, когда несколько потоков одновременно вызывают функцию, которая принимает решение о масштабировании с помощью `std::thread::hardware_concurrency()`, превышение лимита получится очень большим. Функция `std::async()` решает эту проблему, потому что библиотека знает обо всех обращениях к ней и может планировать потоки с учетом этой информации. Избежать этой трудности помогают также пулы потоков, если пользоваться ими с умом.

Однако даже если вы учитываете все потоки в своем приложении, остаются еще другие запущенные в системе программы. Вообще-то в однопользовательских системах редко запускают одновременно несколько счетных задач, но бывают области применения, где это обычное дело. Если система проектировалась специально под такие условия, то обычно в ней есть механизмы, позволяющие каждому приложению заказать подходящее количество потоков, хотя они и выходят за рамки стандарта C++. Один из вариантов — аналог `std::async()`, который при выборе количества потоков учитывает общее число асинхронных задач, выполняемых всеми приложениями. Другой — ограничение числа процессорных ядер, доступных данному приложению. Лично я ожидал бы, что это ограничение будет отражено в значении, которое возвращает функция `std::thread::hardware_concurrency()` на таких платформах, однако это не гарантируется. Если вас интересует подобный сценарий, обратитесь в документации.

Положение осложняется еще и тем, что идеальный алгоритм для решения конкретной задачи может зависеть от размерности задачи в сравнении с количеством процессорных устройств. Если имеется *массивно параллельная* система, где процессоров очень много, то алгоритм с большим числом операций может завершиться быстрее алгоритма с меньшим числом операций, потому что каждый процессор выполняет лишь малую толику общего числа операций.

По мере роста числа процессоров возникает и еще одна проблема, влияющая на производительность: обращение к общим данным со стороны нескольких процессоров.

8.2.2. Конкуренция за данные и перебрасывание кэша

Если два потока, одновременно выполняющиеся на разных процессорах, *читают* одни и те же данные, то обычно проблемы не возникает — данные просто копируются в кэши каждого процессора. Но если один поток *модифицирует* данные, то изменение должно попасть в кэш другого процессора, а на это требуется время. В зависимости от характера операций в двух потоках и от упорядочения доступа к памяти, модификация может привести к тому, что один процессор должен будет остановиться и подождать, пока аппаратура распространит изменение. С точки зрения процессора, это *феноменально* медленная операция, эквивалентная многим сотням машинных команд, хотя точное время зависит в основном от физической конструкции оборудования.

Рассмотрим следующий простой фрагмент кода:

```
std::atomic<unsigned long> counter(0);
void processing_loop() {
 while(counter.fetch_add(
  1, std::memory_order_relaxed) < 100000000) {
  do_something();
 }
}
```

Переменная `counter` глобальная, поэтому любой поток, вызывающий `processing_loop()`, изменяет одну и ту же переменную. Следовательно, после каждого инкремента процессор должен загрузить в свой кэш актуальную копию `counter`, модифицировать ее значение и сделать его доступным другим процессорам. И хотя мы задали упорядочение `std::memory_order_relaxed`, чтобы процессору не нужно было синхронизироваться с другими данными, `fetch_add` — это операция чтения-модификации-записи и, значит, должна получить самое последнее значение переменной. Если другой поток на другом процессоре выполняет этот же код, то значение `counter` придётся передавать из кэша одного процессора в кэш другого, чтобы при каждом инкременте процессор видел актуальное значение `counter`. Если функция `do_something()` достаточно короткая или этот код исполняет много процессоров, то дело кончится тем, что они будут *ожидать* друг друга; один процессор готов обновить значение, но в это время другой уже обновляет, поэтому придётся дождаться завершения операции и распространения изменения. Такая ситуация называется *высокой конкуренцией.* Если процессорам редко приходится ждать друг друга, то говорят о *низкой конкуренции*.

Подобный цикл приводит к тому, что значение counter многократно передается из одного кэша в другой. Это явление называют *перебрасыванием кэша* (cache ping-pong), оно может серьезно сказаться на производительности приложения. Когда процессор простаивает в ожидании передачи в кэш, он не может делать *вообще* ничего, даже если имеются другие потоки, которые могли бы заняться полезной работой. Так что ничего хорошего в этом случае приложению не светит.

Быть может, вы думаете, что к вам это не относится — ведь в вашей-то программе таких циклов нет. Да так ли? А как насчет блокировок мьютексов? Когда программа захватывает мьютекс в цикле, она выполняет очень похожий код — с точки зрения доступа к данным. Чтобы захватить мьютекс, поток должен доставить составляющие мьютекс данные своему процессору и модифицировать их. Затем он снова модифицирует мьютекс, чтобы освободить его, а данные мьютекса необходимо передать следующему потоку, желающему его захватить. Время передачи *добавляется* к времени, в течение которого второй поток должен ждать, пока первый освободит мьютекс:

```
std::mutex m;
my_data data;
void processing_loop_with_mutex() {
 while (true) {
  std::lock_guard<std::mutex> lk(m);
  if (done_processing(data)) break;
 }
}
```

А теперь самое печальное: если к данным и мьютексу обращаются сразу несколько потоков, то при увеличении числа ядер и процессоров ситуация только *ухудшается*, то есть возрастает вероятность получить высокую конкуренцию из-за того, что процессоры ждут друг друга. Если вы запускаете несколько потоков для ускорения обработки одних и тех же данных, то потоки начинают конкурировать за данные и, следовательно, за один и тот же мьютекс. Чем потоков больше, чем вероятнее, что они будут пытаться одновременно захватить мьютекс или получить доступ к атомарной переменной или еще что-нибудь в этом роде.

Последствия конкуренции за мьютексы и за атомарные переменные обычно разнятся по той простой причине, что мьютекс сериализует потоки на уровне операционной системы, а

не процессора. Если количество готовых к выполнению потоков достаточно, то операционная система может запланировать один поток, пока другой ожидает мьютекса. Напротив, застывший процессор прекращает выполнение работающего на нем потока. Тем не менее, он оказывает влияние на производительность потоков, *конкурирующих* за мьютекс, — в конце концов, по определению в каждый момент времени может выполняться только один из них.

В главе 3 мы видели, что редко обновляемую структуру данных можно защитить мьютексом типа «несколько читателей — один писатель» (см. раздел 3.3.2). Эффект перебрасывания кэша может свести на нет преимущества такого мьютекса при неподходящей рабочей нагрузке, потому что все потоки, обращающиеся к данным (пусть даже только для чтения) все равно должны модифицировать сам мьютекс. По мере увеличения числа процессоров, обращающихся к данным, конкуренция за мьютекс возрастает, и строку кэша, в которой находится мьютекс, приходится передавать между ядрами, что увеличивает время захвата и освобождения мьютекса до неприемлемого уровня. Существуют приёмы, позволяющие сгладить остроту этой проблемы; суть их сводится к распределению мьютекса между несколькими строками кэша, но если вы не готовы реализовать такой мьютекс самостоятельно, то должны будете мириться с тем, что дает система.

Если перебрасывание кэша — это так плохо, то можно ли его избежать? Ниже в этой главе мы увидим, что ответ лежит в русле общих рекомендаций но улучшению условий для распараллеливания: делать все возможное для того, чтобы сократить конкуренцию потоков за общие ячейки памяти.

Но это не так-то просто — впрочем, просто никогда не бывает. Даже если к некоторой ячейке памяти в каждый момент времени обращается только один поток, перебрасывание кэша все равно возможно из-за явления, которое называется *ложным разделением* (false sharing).

8.2.3. Ложное разделение

Процессорные кэши имеют дело не с отдельными ячейками памяти, а с блоками ячеек, которые называются *строками кэша.* Обычно размер такого блока составляет 32 или 64 байта, в зависимости от конкретного процессора. Поскольку аппаратный кэш оперирует блоками памяти, небольшие элементы данных, находящиеся в смежных ячейках, часто оказываются в одной строке кэша. Иногда это хорошо: с точки зрения производительности лучше, когда данные, к которым обращается поток, находятся в одной, а не в нескольких строках кэша. Однако, если данные, оказавшиеся в одной строке кэша, логически не связаны между собой и к ним обращаются разные потоки, то возможно возникновение неприятной проблемы.

Допустим, что имеется массив значений типа `int` и множество потоков, каждый из которых обращается к своему элементу массива, в том числе для обновления, причём делает это часто. Поскольку размер `int` обычно меньше строки кэша, то в одной строке окажется несколько значений. Следовательно, хотя каждый поток обращается только к своему элементу, перебрасывание кэша *все равно* имеет место. Всякий раз, как поток хочет обновить значение в позиции 0, вся строка кэша должна быть передана процессору, на котором этот поток исполняется. И для чего? Только для того, чтобы потом быть переданной процессору, на котором исполняется поток, желающий обновить элемент в позиции 1.

Строка кэша разделяется, хотя каждый элемент данных принадлежит только одному потоку. Отсюда и название *ложное разделение*. Решение заключается в том, чтобы структурировать данные таким образом, чтобы элементы, к которым обращается один поток, находились в памяти рядом (и, следовательно, с большей вероятностью попали в одну строку кэша), а элементы, к которым обращаются разные потоки, отстояли далеко друг от друга и попадали в разные строки кэша. Как это влияет на проектирование кода и данных, вы узнаете ниже.

Если обращение из разных потоков к данным, находящимся в одной строке кэша, — зло, то как влияет на общую картину расположение в памяти данных, к которым обращается один поток?

8.2.4. Насколько близки ваши данные?

Ложное разделение вызвано тем, что данные, нужные одному потоку, расположены в памяти слишком близко к данным другого потока. Но есть и еще одна связанная с размещением данных в памяти проблема, которая влияет на производительность одного потока безотносительно к прочим. Эта локальность данных — если данные, к которым обращается поток, разбросаны по памяти, то велика вероятность, что они находятся в разных строках кэша. И наоборот, если данные потока расположены близко друг к другу, то они с большей вероятностью принадлежат одной строке кэша. Следовательно, когда данные разбросаны, из памяти в кэш процессора приходится загружать больше строк кэша, что увеличивает задержку памяти и снижает общую производительность.

Кроме того, если данные разбросаны, то возрастают шансы на то, что строка кэша, содержащая данные текущего потока, содержит также данные *других* потоков. В худшем случае может оказаться, что кэш содержит больше ненужных вам данных, чем нужных. Таким образом, впустую растрачивается драгоценная кэш-память и, значит, количество безрезультатных обращений к кэшу увеличивается, и процессору приходится загружать данные из основной памяти, хотя они уже когда-то находились в кэше, но были вытеснены.

Да, но ведь это относится к однопоточным программам, я-то тут при чем? — спросите вы. А все дело в *контекстном переключении*. Если количество потоков в системе превышает количество ядер, то каждое ядро будет исполнять несколько потоков. Это увеличивает давление на кэш, поскольку мы пытаемся сделать так, чтобы разные потоки обращались к разным строкам кэша — во избежание ложного разделения. Следовательно, при переключении потоков процессором вероятность перезагрузки строк кэша возрастает, если данные каждого потока находятся в разных строках кэша. Если потоков больше, чем ядер или процессоров, то операционная система может назначить потоку один процессор в течение одного кванта времени и другой — в следующем кванте. В результате строки кэша, содержащие данные этого потока, придётся передавать из одного кэша в другой. Чем больше таких передач, тем больше на них уходит времени. Конечно, операционные системы стараются избежать такого развития событий, но иногда это все же происходит и приводит к падению производительности.

Проблемы, связанные с контекстным переключением, возникают особенно часто, когда количество *готовых к выполнению* потоков намного превышает количество *ожидающих*. Мы уже говорили об этом феномене, который называется превышением лимита.

8.2.5. Превышение лимита и чрезмерное контекстное переключение

В многопоточных программах количество потоков нередко превышает количество процессоров, если только не используется *массивно параллельное* оборудование. Однако потоки часто тратят время на ожидание внешнего ввода/вывода, освобождения мьютекса, сигнала условной переменной и т.д., поэтому серьезных проблем не возникает. Наличие избыточных потоков позволяет приложению выполнять полезную работу, а не простаивать, пока потоки чего-то ждут.

Но не всегда это хорошо. Если избыточных потоков *слишком много*, то даже *готовых к выполнению* потоков будет больше, чем процессоров, и операционная система будет вынуждена интенсивно переключать потоки, чтобы никого не обделить временными квантами. В главе 1 мы видели, что это приводит к возрастанию накладных расходов на контекстное переключение, а также к проблемам с кэш-памятью из-за локальности. Превышение лимита может возникать, когда задача порождает новые потоки без ограничений, как, например, в алгоритме рекурсивной сортировки из главы 4, или когда количество потоков, естественно возникающих при распределении работы но типам задач, превышает количество процессоров, а рабочая нагрузка носит счетный характер и мало связана с вводом/выводом.

Количество потоков, запускаемых из-за особенностей алгоритма распределения данных, можно ограничить, как показано в разделе 8.1.2. Если же превышение лимита обусловлено естественным распределением работы, то тут ничего не поделаешь, остается разве что выбрать другой способ распределения. Но в таком случае для выбора подходящего распределения может потребоваться больше информации о целевой платформе, чем вы располагаете, поэтому заниматься этим следует лишь тогда, когда производительность неприемлема и можно убедительно продемонстрировать, что изменение способа распределения действительно повышает быстродействие.

Есть и другие факторы, влияющие на производительность многопоточной программы. Например, стоимость перебрасывания кэша может существенно зависеть от того, оснащена ли система двумя одноядерными процессорами или одним двухъдерным, даже если тип и тактовая частота процессоров одинаковы. Однако все основные факторы, эффект которых проявляется наиболее наглядно, были перечислены выше. Теперь рассмотрим, как от них зависит проектирование кода и структур данных.

8.3. Проектирование структур данных для повышения производительности многопоточной программы

В разделе 8.1 мы видели различные способы распределения работы между потоками, а в разделе 8.2 — факторы, от которых может зависеть производительность программы. Как воспользоваться этой информацией при проектировании структур данных для многопоточного кода? Этот вопрос отличается от рассмотренных в главах 6 и 7, где основное внимание было уделено проектированию структур данных, безопасных относительно одновременного доступа. В разделе 2 было показано, что размещение в памяти данных, используемых одним потоком, тоже может иметь значение, даже если эти данные ни с какими другими потоками не разделяются.

Основное, о чем нужно помнить при проектировании структур данных для многопоточной программы, — это *конкуренция, ложное разделение* и *локальность данных*. Все три фактора могут оказать большое влияние на производительность, так что нередко добиться улучшения удается, просто изменив размещение данных в памяти или распределение данных между потоками. Начнем с низко висящего плода: распределения элементов массива между потоками.

8.3.1. Распределение элементов массива для сложных операций

Допустим, что программа, выполняющая сложные математические расчеты, должна перемножить две больших квадратных матрицы. Элемент в левом верхнем углу результирующей матрицы получается следующим образом: каждый элемент первой *строки* левой матрицы умножается на соответственный элемент первого *столбца* правой матрицы и полученные произведения складываются. Чтобы получить элемент результирующей матрицы, расположенный на пересечении второй строки и первого столбца, эта операция повторяется для второй строки левой матрицы и первого столбца правой матрицы. И так далее. На рис. 8.3 показано, что элемент результирующей матрицы на пересечении второй строки и третьего столбца получается суммированием попарных произведений элементов второй строки левой матрицы и третьего столбца правой.

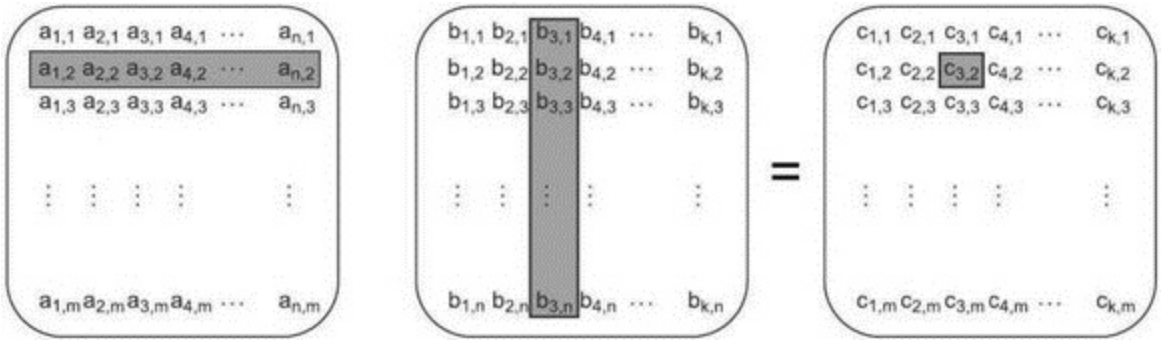

Рис. 8.3. Умножение матриц

Теперь предположим, что матрицы содержат по несколько тысяч строк и столбцов, иначе заводить несколько потоков для оптимизации умножения не имеет смысла. Обычно, если матрица не разрежена, то она представляется большим массивом в памяти, в котором сначала идут все элементы первой строки, потом все элементы второй строки и так далее. Следовательно, для перемножения матриц нам понадобятся три огромных массива. Чтобы

добиться оптимальной производительности, мы должны внимательно следить за порядком доступа к данным, а особенно за операциями записи в третий массив.

Существует много способов распределить эту работу между потоками. В предположении, что строк и столбцов больше, чем имеется процессоров, можно поручить каждому потоку вычисление нескольких столбцов или строк результирующей матрицы или даже вычисление какой-то ее прямоугольной части.

В разделах 8.2.3 и 8.2.4 мы видели, что для улучшения использования кэша и предотвращения ложного разделения лучше, когда поток обращается к данным, находящимся в соседних ячейках, а не разбросанным но всей памяти. Если поток вычисляет множество столбцов, то читать ему придётся все значения из первой матрицы и соответственных столбцов второй матрицы, но писать только в назначенные ему столбцы результирующей матрицы. При том, что матрицы хранятся но строкам, это означает, что мы будем обращаться к *N* элементам первой строки результирующей матрицы, *N* элементам второй строки и т.д. (*N* — количество обрабатываемых столбцов). Другие потоки будут обращаться к другим элементам строк, поэтому, чтобы минимизировать вероятность ложного разделения, столбцы, вычисляемые каждым потоком, должны быть соседними, тогда будут соседними и *N* записываемых элементов в каждой строке. Разумеется, если эти *N* элементов занимают в точности целое число строк кэша, то ложного разделения вообще не будет, потому что каждый поток работает со своим набором строк кэша.

С другой стороны, если каждый поток вычисляет множество *строк*, то он должен будет прочитать все значения *правой* матрицы и значения из соответственных строк *левой* матрицы, но в результирующую матрицу записывать будет только строки. Поскольку матрицы хранятся но строкам, то поток будет обращаться ко *всем* элементам *N* строк. Если поручить потоку вычисление соседних строк, то он окажется *единственным* потоком, который пишет в эти строки, то есть будет владельцем непрерывного участка памяти, к которому больше никакие потоки не обращаются. Вероятно, это лучше, чем вычисление множества столбцов, потому что ложное разделение, если и может возникнуть, то только для нескольких последних элементов предыдущего и нескольких первых элементов следующего участка. Однако это предположение нуждается в подтверждении путем прямых измерений на целевой платформе.

А как насчет третьего варианта — вычисления прямоугольных блоков матрицы? Можно рассматривать его как комбинацию распределения но столбцам и по строкам. Поэтому шансы на ложное разделение такие же, как при распределении но столбцам. Если выбрать число столбцов так, чтобы минимизировать эту вероятность, то у распределения по блокам будет преимущество в части *чтения* — исходную матрицу придётся читать не целиком, а только те столбцы и строки, которые соответствуют назначенному потоку прямоугольному блоку. Рассмотрим конкретный пример умножения двух матриц размерностью 1000×1000. Всего в каждой матрице миллион элементов. При наличии 100 процессоров каждый из них сможет вычислить 10 строк, то есть 10 000 элементов. Однако для их вычисления процессору придётся прочитать всю правую матрицу (миллион элементов) плюс 10 000 элементов из соответственных строк левой матрицы, итого — 1 010 000 элементов. С другой стороны, если каждый процессор вычисляет блок 100×100 (те же 10 000 элементов), то нужно будет прочитать значения из 100 строк левой матрицы (100×1000 = 100 000 элементов) и 100 столбцов правой матрицы (еще 100 000). В итоге мы получаем только 200 000 прочитанных элементов, то есть в пять раз меньше по сравнению с первым случаем.

Если читается меньше элементов, то сокращается вероятность отсутствия нужного значения в кэше, а, значит, производительность потенциально возрастает.

Поэтому лучше разбить результирующую матрицу на небольшие квадратные или почти квадратные блоки, а не вычислять строки целиком. Конечно, размер блока можно определять на этапе выполнения в зависимости от размерности матриц и числа имеющихся процессоров. Как обычно, если производительность существенна, то важно профилировать разные варианты решения на целевой архитектуре.

Но если вы не занимаетесь умножением матриц, то какую пользу можете извлечь из этого обсуждения? Да просто те же принципы применимы в любой ситуации, где нужно назначать потокам вычисление больших блоков данных. Тщательно изучите все аспекты доступа к данным и выявите потенциальные причины снижения производительности. Не исключено, что ваша задача обладает схожими характеристиками, позволяющими улучшить быстродействие всего лишь за счет изменения способа распределения работы без модификации основного алгоритма.

Итак, мы посмотрели, как порядок доступа к элементам массива может отразиться на производительности. А что можно сказать о других структурах данных?

8.3.2. Порядок доступа к другим структурам данных

По существу, к оптимизации доступа к другим структурам данных применимы те же принципы, что и для массивов.

• Попытайтесь выбрать распределение данных между потоками, так чтобы данные, расположенные по соседству, обрабатывались одним потоком.

• Попытайтесь минимизировать объем данных, к которым обращается каждый поток.

• Попытайтесь сделать так, чтобы данные, к которым обращаются разные потоки, находились достаточно далеко друг от друга, чтобы избежать ложного разделения.

Разумеется, к другим структурам данных применить эти принципы не так просто. Например, в двоичном дереве очень трудно выделить части, которые сами не являлись бы деревьями, а полезно это или нет, зависит от того, насколько дерево сбалансировано и на сколько частей его нужно разбить. К тому же, память для узлов деревьев по необходимости выделяется динамически, так что оказывается в разных частях кучи.

Само по себе то, что данные находятся в разных частях кучи, не страшно, но означает, что процессору придётся держать в кэше ячейки из разных участков памяти. На самом деле, это может быть даже хорошо. Если несколько потоков обходят дерево, то всем им нужно получать доступ к узлам. Однако если узлы содержат только *указатели* на реальные данные, то процессор должен будет загружать данные только по мере необходимости. Если данные модифицируются потоками, то за счет этого, возможно, удастся предотвратить падение производительности из-за ложного разделения между данными самого узла и данными, образующими структуру дерева.

Схожая проблема возникает для данных, защищенных мьютексом. Предположим, что имеется простой класс, содержащий какие-то элементы данных и защищающий их мьютекс. Для потока, захватывающего мьютекс, было бы идеально, чтобы мьютекс и данные были размещены в памяти рядом. Тогда необходимые ему данные уже находятся в кэше процессора, потому что были загружены вместе с мьютексом, когда поток модифицировал его для захвата. Но есть и оборотная сторона медали: другие потоки, пытающиеся захватить

мьютекс, удерживаемый первым потоком, должны будут обратиться к той же памяти. Захват мьютекса обычно реализуется в виде атомарной операции чтения-модификации-записи ячейки памяти, принадлежащей мьютексу, с последующим вызовом ядра ОС, если мьютекс уже захвачен. Операция чтения-модификации-записи вполне может сделать недействительными хранящиеся в кэше данные. С точки зрения мьютекса, это несущественно, так как первый поток все равно не стал бы его трогать, пока не подойдёт время освобождения. Но если мьютекс находится в той же строке кэша, что и данные, которыми оперирует захвативший его поток, то получится, что производительность потока, владеющего мьютексом, надает только *потому, что другой поток попытался захватить тот же мьютекс*.

Один из способов проверить, приводит ли такого рода ложное разделение к проблемам, — добавить большие разделительные блоки фиктивных данных между данными, к которым одновременно обращаются разные потоки. Например, следующая структура:

```
struct protected_data {
 std::mutex m;
 char padding[65536];
 my_data data_to_protect;
};
```

65536 на несколько порядков больше, чем длина строки кэша

удобна для проверки конкуренции за мьютекс, а структура

```
struct my_data {
 data_item1 d1;
 data_item2 d2;
 char padding[65536];
};
my_data some_array[256];
```

— для проверки ложного разделения данных массива. Если в результате производительность повысится, значит, ложное разделение составляет проблему, и тогда можно либо оставить заполнитель, либо устранить ложное разделение, по-другому организовав доступ к данным.

Разумеется, порядок доступа к данным — не единственное, что нужно принимать во внимание при проектировании параллельных программ. Рассмотрим некоторые другие аспекты.

8.4. Дополнительные соображения при проектировании параллельных программ

До сих пор мы в этой главе рассматривали различные способы распределения работы между потоками, факторы, влияющие на производительность, и то, как от них зависит выбор порядка доступа к данным и самой структуры данных. Но этим проблематика проектирования параллельных программ не исчерпывается. Необходимо еще принимать во внимание такие вещи, как безопасность относительно исключений и масштабируемость. Говорят, что программа масштабируется, если ее производительность (в терминах повышения быстродействия или увеличения пропускной способности) возрастает при добавлении новых процессорных ядер. В идеале производительность должна расти линейно, то есть система с 100 процессорами должна работать в 100 раз быстрее системы с одним процессором.

Даже немасштабируемая программа может быть работоспособной — в конце концов, однопоточные приложения в этом смысле, безусловно, не масштабируемы — но вот безопасность относительно исключений — вопрос, напрямую связанный с корректностью. Если программа не безопасна относительно исключений, то может наблюдаться нарушение инвариантов, состояния гонки и даже аварийное завершение. Вот этим вопросом мы сейчас и займемся.

8.4.1. Безопасность относительно исключений в параллельных алгоритмах

Безопасность относительно исключений — необходимая составная часть любой приличной программы на C++, и параллельные программы — не исключение. На самом деле, при разработке параллельных алгоритмов часто требуется уделять исключениям даже больше внимания. Если какая-то операция в последовательном алгоритме возбуждает исключение, то алгоритм должен лишь позаботиться о предотвращении утечек памяти и нарушения собственных инвариантов, а потом может передать исключение вызывающей программе для обработки. В параллельных же алгоритмах многие операции выполняются в разных потоках. В этом случае исключение невозможно распространить вверх по стеку вызовов, потому что у каждого потока свой стек. Если выход из функции потока производится в результате исключения, то приложение завершается.

В качестве конкретного примера рассмотрим еще раз функцию `parallel_accumulate` из листинга 2.8, который воспроизведен ниже.

Листинг 8.2. Наивная параллельная организация `std::accumulate` (из листинга 2.8)

```
template<typename Iterator, typename T>
struct accumulate_block {
 void operator()(Iterator first, Iterator last, T& result) {
  result = std::accumulate(first, last, result); ← (1)
 }
};

template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
```

```
unsigned long const length = std::distance(first, last);← (2)
if (!length)
 return init;

unsigned long const min_per_thread = 25;
unsigned long const max_threads =
 (length + min_per_thread - 1) / min_per_thread;

unsigned long const hardware_threads =
 std::thread::hardware_concurrency();

unsigned long const num_threads =
 std::min(
  hardware_threads != 0 ? hardware_threads : 2, max_threads);

unsigned long const block_size = length / num_threads;

std: :vector<T> results (num_threads); ← (3)
std::vector<std::thread> threads(num_threads - 1); ← (4)

Iterator block_start = first; ← (5)
for (unsigned long i = 0; i < (num_threads - 1); ++i) {
 Iterator block_end = block_start; ← (6)
 std::advance(block_end, block_size);
 threads[i] = std::thread( ← (7)
  accumulate_block<Iterator, T>(),
  block_start, block_end, std::ref(results[i]));
 block_start = block_end; ← (8)
}

accumulate_block()(
 block_start, last, results[num_threads - 1]);← (9)

std::for_each(threads.begin(), threads.end(),
std::mem_fn(&std::thread::join));
return
 std::accumulate(results.begin(), results.end(), init); ← (10)
}
```

Посмотрим, где могут возникнуть исключения. Вообще говоря, это вызовы библиотечных функций, которые могут возбуждать исключения, а также операции, определенные в пользовательском типе.

Итак, начнем. В точке **(2)** мы обращаемся к функции `distance`, которая выполняет операции над пользовательским типом итератора. Поскольку мы еще не начали работу, и обращение к этой функции произведено из вызывающего потока, то тут всё нормально. Далее мы выделяем память для векторов `results` **(3)** и `threads` **(4)**. И эти обращения произведены из вызывающего потока до начала работы и до создания новых потоков, так что и здесь всё хорошо. Разумеется, если конструктор `threads` возбудит исключение, то нужно будет освободить память, выделенную для `results`, но об этом позаботится деструктор.

С инициализацией объекта `block_start` **(5)** всё чисто по тем же причинам, так что

перейдём к операциям в цикле запуска потоков **(6)**, **(7)**, **(8)**. Если после создания первого же потока **(7)** возникнет исключение, и мы его не перехватим, появится проблема; деструкторы объектов `std::thread` вызывают `std::terminate`, что приводит к аварийному завершению программы. Нехорошо.

Обращение к `accumulate_block` в точке **(9)** может возбуждать исключения — с точно такими же последствиями: объекты потоков будут уничтожены, а их деструкторы вызовут `std::terminate`. С другой стороны, исключение, возбуждаемое в последнем обращении к `std::accumulate` **(10)**, не так опасно, потому что к этому моменту все потоки уже присоединились к вызывающему.

Таким образом, обращения к `accumulate_block` из новых потоков могут возбуждать исключения в точке **(1)**. Так как блоки `catch` отсутствуют, то исключение останется необработанным и приведёт к вызову `std::terminate()` и завершению программы.

Если это еще не очевидно, скажем прямо: *этот код не безопасен относительно исключений*.

Делаем код безопасным относительно исключений

Итак, мы выявили все возможные точки возбуждения исключений и поняли, к каким печальным последствиям это приведёт. Что с этим можно сделать? Начнем с вопроса об исключениях, возбуждаемых в созданных нами потоках.

В главе 4 мы уже познакомились с подходящим для решения проблемы средством. Для чего нам вообще нужны новые потоки? Для того чтобы вычислить результат и при этом учесть возможность возникновения исключений. Но это *именно то*, для чего предназначено сочетание `std::packaged_task` и `std::future`. В листинге 8.3 показан код, переписанный с использованием `std::packaged_task`.

Листинг 8.3. Параллельная версия `std::accumulate` с применением `std::packaged_task`

```
    template<typename Iterator, typename T>
    struct accumulate_block {
     T operator()(Iterator first, Iterator last) {← (1)
      return std::accumulate(first, last, T());   ← (2)
     }
    };

    template<typename Iterator, typename T>
    T parallel_accumulate(Iterator first, Iterator last, T init) {
     unsigned long const length = std::distance(first, last);

     if (!length)
      return init;

     unsigned long const min_per_thread = 25;
     unsigned long const max_threads =
      (length + min_per_thread – 1) / min_per_thread;

     unsigned long const hardware_threads =
```

```
 std::thread::hardware_concurrency();

unsigned long const num_threads =
 std::min(
  hardware_threads i = 0 ? hardware_threads : 2, max_threads);

unsigned long const block_size = length / num_threads;

std::vector<std::future<T> > futures(num_threads-1);← (3)
std::vector<std::thread> threads(num_threads – 1);

Iterator block_start = first;
for (unsigned long i = 0; i < (num_threads - 1); ++i) {
 Iterator block_end = block_start;
 std::advance(block_end, block_size);
 std::packaged_task<T(Iterator, Iterator)> task( ←(4)
  accumulate_block<Iterator, T>());
 futures[i] = task.get_future(); ←(5)
 threads[i] =
  std::thread(std::move(task), block_start, block_end);←(6)
 block_start = block_end;
}
T last_result = accumulate_block()(block_start, last); ←(7)

std::for_each(threads.begin(), threads.end(),
 std::mem_fn(&std::thread::join));

T result = init; ←(8)
for (unsigned long i = 0; i < (num_threads - 1); ++i) {
 result += futures[i].get(); ←(9)
}
result += last_result; ←(10)
return result;
}
```

Первое изменение заключается в том, что оператор вызова в `accumulate_block` теперь возвращает результат по значению, а не принимает ссылку на место, где его требуется сохранить **(1)**. Для обеспечения безопасности относительно исключений мы используем `std::packaged_task` и `std::future`, поэтому можем воспользоваться этим и для передачи результата. Правда, для этого требуется при вызове `std::accumulate` **(2)** явно передавать сконструированный по умолчанию экземпляр `T`, а не использовать повторно предоставленное значение `result`, но это не слишком существенное изменение.

Далее, вместо того заводить вектор результатов, мы создаем вектор `futures` **(3)**, в котором храним объекты `std::future<T>` для каждого запущенного потока. В цикле запуска потоков мы сначала создаем задачу для `accumulate_block` **(4)**. В классе `std::packaged_task<T(Iterator, Iterator)>` объявлена задача, которая принимает два объекта `Iterator` и возвращает `T`, а это именно то, что делает наша функция. Затем мы получаем будущий результат для этой задачи **(5)** и исполняем ее в новом потоке, передавая начало и конец обрабатываемого блока **(6)**. Результат работы задачи, равно как и исключение, если оно возникнет, запоминается в объекте `future`.

Поскольку используются будущие результаты, массива `results` больше нет, поэтому мы должны сохранить результат обработки последнего блока в переменной **(7)**, а не в элементе массива. Кроме того, поскольку мы получаем значения из будущих результатов, проще не вызывать `std::accumulate`, а написать простой цикл `for`, в котором к переданному начальному значению **(8)** будут прибавляться значения, полученные из каждого будущего результата **(9)**. Если какая-то задача возбудит исключение, то оно будет запомнено в будущем результате и повторно возбуждено при обращении к `get()`. Наконец, перед тем как возвращать окончательный результат вызывающей программе, мы прибавляем результат обработки последнего блока **(10)**.

Таким образом, мы устранили одну из потенциальных проблем: исключения, возбужденные в рабочих потоках, повторно возбуждаются в главном. Если исключения возникнут в нескольких рабочих потоках, то вверх распространится только одно, но это не очень страшно. Если вы считаете, что это все-таки важно, то можете воспользоваться чем-то вроде класса `std::nested_exception`, чтобы собрать все такие исключения и передать их главному потоку.

Осталось решить проблему утечки потоков в случае, когда исключение возникает между моментом запуска первого потока и присоединением всех запущенных. Для этого проще всего перехватить любое исключение, дождаться присоединения потоков, которые все еще находятся в состоянии `joinable()`, а потом возбудить исключение повторно:

```
try {
 for (unsigned long i = 0; i < (num_threads - 1); ++i) {
  // ... как и раньше
 }
 T last_result = accumulate_block()(block_start, last);
 std::for_each(threads.begin(), threads.end(),
 std::mem_fn(&std::thread::join));
} catch (...) {
 for (unsigned long i = 0; i < (num_thread - 1); ++i) {
 if (threads[i].joinable())
  thread[i].join();
 }
 throw;
}
```

Теперь все работает. Все потоки будут присоединены вне зависимости от того, как завершилась обработка блока. Однако блоки `try-catch` выглядят некрасиво, и часть кода дублируется. Мы присоединяем потоки как в «нормальной» ветке, так и в блоке `catch`. Дублирование кода — вещь почти всегда нежелательная, потому что изменения придётся вносить в несколько мест. Давайте лучше перенесём этот код в деструктор — ведь именно такова идиома очистки ресурсов в C++. Вот как выглядит этот класс:

```
class join_threads {
 std::vector<std::thread>& threads;
public:
 explicit join_threads(std::vector<std::thread>& threads_):
  threads(threads_) {}


 ~join_threads() {
  for (unsigned long i = 0; i < threads.size(); ++i) {
   if (threads[i].joinable())
    threads[i].join();
  }
```

```
 }
};
```

Это похоже на класс `thread_guard` из листинга 2.3, только на этот раз мы «охраняем» целый вектор потоков. Теперь наш код упрощается.

Листинг 8.4. Безопасная относительно исключений версия `std::accumulate`

```
template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
 unsigned long const length = std::distance(first, last);

 if (!length)
  return init;

 unsigned long const min_per_thread = 25;
 unsigned long const max_threads =
  (length + min_per_thread - 1) / min_per_thread;

 unsigned long const hardware_threads =
  std::thread::hardware_concurrency();

 unsigned long const num_threads =
  std::min(
   hardware_threads i = 0 ? hardware_threads : 2, max_threads);

 unsigned long const block_size = length / num_threads;

 std::vector<std::future<T> > futures(num_threads — 1);
 std::vector<std::thread> threads(num_threads - 1);
 join_threads joiner(threads); ← (1)

 Iterator block_start = first;
 for (unsigned long i = 0; i < (num_threads - 1); ++i) {
  Iterator block_end = block_start;
  std::advance(block_end, block_size);
  std::packaged_task<T(Iterator, Iterator)> task(
   accumulate_block<Iterator, T>());
  futures[i] = task.get_future();
  threads[i] =
   std::thread(std:move(task), block_start, block_end);
  block_start = block_end;
 }
 T last_result = accumulate_block()(block_start, last);

 T result = init;
 for (unsigned long i = 0; i < (num_threads - 1); ++i) {
  result + = futures[i].get(); ← (2)
 }
 result += last_result;
 return result;
}
```

Создав контейнер потоков, мы создаем объект написанного выше класса **(1)**, который присоединяет все завершившиеся потоки. Теперь явный цикл присоединения можно удалить, точно зная, что при выходе из функции потоки будут присоединены. Отметим, что вызовы

`futures[i].get()` **(2)** приостанавливают выполнение программы до готовности результатов, поэтому в этой точке явно присоединять потоки не нужно. Этим данный код отличается от первоначальной версии в листинге 8.2, где присоединять потоки было необходимо для нрав ильного заполнения вектора `results`. Мало того что мы получили безопасный относительно исключений код, так еще и функция стала короче, потому что код присоединения вынесен в новый класс (который, кстати, можно использовать и в других программах).

Обеспечение безопасности относительно исключений при работе с `std::async()`

Теперь, когда мы знаем, что необходимо для обеспечения безопасности относительно исключений в программе, которая явно управляет потоками, посмотрим, как добиться того же результата при использовании `std::async()`. Мы уже видели, что в этом случае управление потоками берет на себя библиотека, а запущенный ей поток завершается, когда будущий результат *готов*. В плане безопасности относительно исключений важно то, что если уничтожить будущий результат, не дождавшись его, то деструктор будет ждать завершения потока. Тем самым мы элегантно решаем проблему утечки потоков, которые еще исполняются и хранят ссылки на данные. В следующем листинге показана безопасная относительно исключений реализация с применением `std::async()`.

Листинг 8.5. Безопасная относительно исключений версия `std::accumulate` с применением `std::async`

```
template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
 unsigned long const length = std::distance(first, last);← (1)
 unsigned long const max_chunk_size = 25;
 if (length <= max_chunk_size) {
  return std::accumulate(first, last, init); ← (2)
 } else {
  Iterator mid_point = first;
  std::advance(mid_point, length / 2); ← (3)
  std::future<T> first_half_result =
   std::async(parallel_accumulate<Iterator, T>, ← (4)
   first, mid_point, init);
  T second_half_result =
   parallel_accumulate(mid_point, last, T());          ← (5)
  return first_half_result.get() + second_half_result;← (6)
 }
}
```

В этой версии мы распределяем данные рекурсивно, а не вычисляем распределение по блокам заранее, но в целом код намного проще предыдущей версии и при этом *безопасен относительно исключений*. Как и раньше, сначала мы вычисляем длину последовательности **(1)**, и, если она меньше максимального размера блока, то вызываем `std::accumulate` напрямую **(2)**. В противном случае находим среднюю точку последовательности **(3)** и запускаем асинхронную задачу, которая будет обрабатывать левую половину **(4)**. Для обработки правой половины мы вызываем себя рекурсивно **(5)**, а затем складываем результаты обработки обеих половин **(6)**. Библиотека гарантирует, что `std::async`

задействует имеющийся аппаратный параллелизм, не создавая слишком большого количества потоков. Некоторые «асинхронные» вызовы на самом деле будут исполняться синхронно при обращении к `get()` **(6)**.

Изящество этого решения не только в том, что задействуется аппаратный параллелизм, но и в том, что безопасность относительно исключений обеспечивается тривиальным образом. Если рекурсивный вызов **(5)** возбудит исключение, то будущий результат, созданный при обращении к `std::async` **(4)**, будет уничтожен при распространении исключения вверх по стеку. Это в свою очередь приведёт к ожиданию завершения асинхронного потока, так что «висячий поток» не образуется. С другой стороны, если исключение возбуждает асинхронный вызов, то оно будет запомнено в будущем результате и повторно возбуждено при обращении к `get()` **(6)**.

Какие еще соображения следует принимать во внимание при проектировании параллельного кода? Давайте поговорим о *масштабируемости*. Насколько увеличится производительность программы, если запустить ее на машине с большим количеством процессоров?

8.4.2. Масштабируемость и закон Амдала

Под *масштабируемостью* понимается свойство программы использовать дополнительные имеющиеся в системе процессоры. На одном полюсе находятся однопоточные приложения, которые вообще не масштабируемы, — даже если система оснащена 100 процессорами, быстродействие такой программы не возрастет. На другом полюсе мы встречаем проект SETI@Home[19], который рассчитан на использование тысяч процессоров (в виде отдельных компьютеров, добровольно подключаемых к сети пользователями).

В многопоточной программе количество потоков, выполняющих полезную работу, может изменяться в процессе исполнения. Даже если каждый поток на всем протяжении своего существования делает что-то полезное, первоначально в приложении имеется всего один поток, который должен запустить все остальные. Но такой сценарий крайне маловероятен. Потоки часто не работают, а ожидают друг друга или завершения операций ввода/вывода.

Всякий раз, как один поток чего-то ждет (неважно, чего именно), а никакого другого потока, готового занять вместо него процессор, нет, процессор, который мог бы выполнять полезную работу, простаивает.

Упрощенно можно представлять, что программа состоит из «последовательных» участков, в которых полезные действия выполняет только один поток, и «параллельных», где задействованы все имеющиеся процессоры. Если программа исполняется на машине с большим числом процессоров, то теоретически «параллельные» участки могли бы завершаться быстрее, а «последовательные» такими бы и остались. Приняв такие упрощающие предположения, можно оценить потенциальное повышение производительности при увеличении количества процессоров: если доля «последовательных» участков равна f_s, то коэффициент повышения производительности P при числе процессоров N составляет

$$P = \frac{1}{f_s + \frac{1 - f_s}{N}}$$

Это *закон Амдала*, на который часто ссылаются при обсуждении производительности параллельных программ. Если код полностью распараллелен, то есть доля последовательных участков нулевая, то коэффициент ускорения равен *N*. Если же, к примеру, последовательные участки составляют треть программы, то даже при бесконечном количестве процессоров не удастся добиться ускорения более чем в три раза.

Конечно, эта картина чересчур упрощённая, потому что редко встречаются бесконечно делимые задачи, без чего это соотношение неверно, и не менее редко вся работа сводится только к процессорным вычислениям, как то предполагается в законе Амдала. Во время исполнения потоки могут ожидать разных событий.

Но из закона Амдала все же следует, что если целью распараллеливания является повышение производительности, то следует проектировать всё приложение так, чтобы процессорам всегда было чем заняться. За счет уменьшения длины «последовательных» участков или времени ожидания можно повысить выигрыш от добавления новых процессоров. Альтернативный подход — подать на вход системы больше данных и тем самым загрузить параллельные участки работой; при этом можно будет уменьшить долю последовательных участков и повысить коэффициент *P*.

По существу, масштабируемость — это возможность либо уменьшить время, затрачиваемое на какое-то действие, либо увеличить объем данных, обрабатываемых в единицу времени, при увеличении количества процессоров. Иногда оба свойства эквивалентны (можно обработать больше данных, если каждый элемент обрабатывается быстрее), но это необязательно. Прежде чем выбирать способ распределения работы между потоками, важно определить, какие аспекты масштабируемости представляют для вас наибольший интерес.

В начале этого раздела я уже говорил, что у потоков не всегда есть чем заняться. Иногда они вынуждены ждать другие потоки, завершения ввода/вывода или еще чего-то. Если на время этого ожидания загрузить систему какой-нибудь полезной работой, такое простаивание можно «скрыть».

8.4.3. Сокрытие латентности с помощью нескольких потоков

При обсуждении производительности многопоточного кода мы часто предполагали, что потоки трудятся изо всех сил и, получая в свое распоряжение процессор, всегда имеют полезную работу. Конечно, это не так — потоки часто оказываются блокированы в ожидании какого-то события: завершения ввода/вывода, освобождения мьютекса, завершения операции в каком-то другом потоке, сигнала условной переменной, готовности будущего результата… Наконец, они могут просто спать какое-то время.

Но какова бы ни была причина ожидания, если потоков столько же, сколько физических процессоров, наличие заблокированных потоков означает, что процессоры работают вхолостую. Процессор, который мог бы исполнять поток, вместо этого не делает ничего. Следовательно, если заранее известно, что какой-то поток будет проводить много времени в ожидании, то имеет смысл задействовать на это время процессор, запустив один или

несколько дополнительных потоков.

Возьмем, к примеру, антивирусный сканер, который для распределения работы использует конвейер. Первый поток просматривает файловую систему и помещает имена файлов в очередь. Второй поток выбирает имена файлов из очереди и сканирует их на предмет наличия вирусов. Мы знаем, что поток просмотра файловой системы определённо будет простаивать в ожидании завершения ввода/вывода, поэтому «лишнее» процессорное время отдаем дополнительному потоку сканирования. Таким образом, у нас будет поток выбора файлов и столько потоков сканирования, сколько имеется процессоров. Поскольку потоку сканирования тоже нужно читать большие куски файлов, то имеет смысл еще увеличить количество таких потоков. Однако в какой-то момент потоков может стать слишком много, и система начнет работать медленнее, потому что вынуждена будет расходовать все больше и больше времени на контекстное переключение (см. раздел 8.2.5).

Такого рода настройка является оптимизацией, поэтому следует измерять производительность до и после изменения количества потоков; оптимальное их число зависит как от характера работы, так и от доли времени, затрачиваемой потоком на ожидание.

Иногда удается полезно использовать свободное процессорное время, не запуская дополнительные потоки. Например, если поток блокируется в ожидании завершения ввода/вывода, то имеет смысл воспользоваться асинхронным вводом/выводом, если платформа его поддерживает. Тогда поток сможет заняться полезной работой, пока ввод/вывод выполняется в фоне. С другой стороны, поток, ожидающий завершения операции в другом потоке, может в это время заняться чем-то полезным. Как это делается, мы видели при рассмотрении реализации свободной от блокировок очереди в главе 7. В крайнем случае, когда поток ждет завершения задачи, которая еще не была запущена другим потоком, ожидающий поток может сам выполнить эту задачу целиком или помочь в выполнении какой-то другой задачи. Такой пример мы видели в листинге 8.1, где функция сортировки пыталась отсортировать находящиеся в очереди блоки, пока блоки, которых она ждет, сортируются другими потоками.

Иногда потоки добавляются не для того, чтобы загрузить имеющиеся процессоры, а чтобы быстрее обрабатывать внешние события, то есть повысить *быстроту реакции* системы.

8.4.4. Повышение быстроты реакции за счет распараллеливания

Большинство современных графических интерфейсов пользователя являются *событийно-управляемыми* — пользователь выполняет в интерфейсе какие-то действия — нажимает клавиши или двигает мышь, в результате чего порождается последовательность событий или сообщений, которые приложение затем обрабатывает. Система может и сама порождать сообщения или события. Чтобы все события и сообщения были корректно обработаны, в приложении обычно присутствует цикл такого вида:

```
while (true) {
 event_data event = get_event();
 if (event.type == quit)
  break;
 process(event);
}
```

Детали API, конечно, могут отличаться, но структура всегда одна и та же: дождаться события, обработать его и ждать следующего. В однопоточном приложении такая структура затрудняет программирование длительных задач (см. раздел 8.1.3). Чтобы система оперативно реагировала на действия пользователя, функции `get_event()` и `process()` должны вызываться достаточно часто вне зависимости от того, чем занято приложение. Это означает, что задача должна либо периодически приостанавливать себя и возвращать управление циклу обработки событий, либо сама вызывать функции `get_event()` и `process()` в подходящих точках. То и другое решение усложняет реализацию задачи.

Применив распараллеливание для разделения обязанностей, мы сможем вынести длительную задачу в отдельный поток, а выделенному потоку GUI поручить обработку событий. В дальнейшем потоки могут взаимодействовать с помощью простых механизмов, и мешать код обработки событий с кодом задачи не придётся. В листинге ниже приведён набросок такого разделения обязанностей.

Листинг 8.6. Отделение потока GUI от потока задачи

```
std::thread task_thread;
std::atomic<bool> task_cancelled(false);

void gui_thread() {
 while (true) {
  event_data event = get_event();
  if (event.type == quit)
   break;
  process(event);
 }
}

void task() {
 while (!task_complete() && !task_cancelled) {
  do_next_operation();
 }
 if (task_cancelled) {
  perform_cleanup();
 } else {
  post_gui_event(task_complete);
 }
}

void process(event_data const& event) {
 switch(event.type) {
 case start_task:
  task_cancelled = false;
  task_thread = std::thread(task);
  break;
 case stop_task:
  task_cancelled = true;
  task_thread.join();
  break;
 case task_complete:
  task_thread.join();
  display_results();
  break;
```

```
  default:
   //...
  }
}
```

В результате такого разделения обязанностей поток пользовательского интерфейса всегда будет своевременно реагировать на события, даже если задача занимает много времени. *Быстрота реакции* часто является основной характеристикой приложения с точки зрения пользователя — с приложением, которое полностью зависает на время выполнения некоторой операции (неважно, какой именно), работать неприятно. За счет выделения специального потока для обработки событий пользовательский интерфейс может сам обрабатывать относящиеся к нему сообщения (например, изменение размера или перерисовка окна), не прерывая длительной операции, но передавая адресованные ей сообщения, если таковые поступают.

До сих пор в этой главе мы говорили о том, что следует учитывать при проектировании параллельного кода. Поначалу количество разных факторов может привести в изумление, но постепенно они войдут в плоть и кровь и станут вашей второй натурой. Если описанные выше соображения внове для вас, то, надеюсь, они станут яснее после того, как мы рассмотрим конкретные примеры многопоточного кода.

8.5. Проектирование параллельного кода на практике

В какой мере следует учитывать описанные выше факторы при проектировании, зависит от конкретной задачи. Для демонстрации мы рассмотрим реализацию параллельных версий трех функций из стандартной библиотеки C++. При этом у нас будет знакомая платформа, на которой можно изучать новые вещи. Попутно мы получим работоспособные версии функций, которые можно будет применить при распараллеливании более крупной программы.

Я ставил себе задачей продемонстрировать определенные приёмы, а не написать самый оптимальный код. Реализации, в которых лучше используется имеющееся оборудование, можно найти в академической литературе по параллельным алгоритмам или в специализированных многопоточных библиотеках типа Intel Threading Building Blocks[20].

Концептуально простейшим параллельным алгоритмом является параллельная версия `std::for_each`, с которой я и начну.

8.5.1. Параллельная реализация `std::for_each`

Идея `std::for_each` проста — этот алгоритм вызывает предоставленную пользователем функцию для каждого элемента диапазона. Различие между параллельной и последовательной реализацией `std::for_each` заключается, прежде всего, в порядке вызовов функции. Стандартная версия `std::for_each` вызывает функцию сначала для первого элемента диапазона, затем для второго и так далее, тогда как параллельная версия не дает гарантий относительно порядка обработки элементов, они даже могут (и хочется надеяться, *будут*) обрабатываться параллельно.

Для реализации параллельной версии нужно всего лишь разбить диапазон на участки, которые будут обрабатываться каждым потоком. Количество элементов известно заранее, поэтому такое разбиение можно произвести до начала работы (см. раздел 8.1.1). Мы будем предполагать, что это единственная исполняемая параллельная задача, поэтому вычислить количество требуемых потоков можно с помощью функции `std::thread::hardware_concurrency()`. Мы также знаем, что элементы можно обрабатывать абсолютно независимо, поэтому для предотвращения ложного разделения (см. раздел 8.2.3) имеет смысл использовать соседние блоки.

По своей структуре этот алгоритм похож на параллельную версию `std::accumulate`, описанную в разделе 8.4.1, только вместо вычисления суммы элементов он применяет к ним заданную функцию. На первый взгляд, это должно бы существенно упростить код, потому что не нужно возвращать никакой результат. Но если мы собираемся передавать исключения вызывающей программе, то все равно придется воспользоваться механизмами `std::packaged_task` и `std::future`, чтобы передавать исключения из одного потока в другой. Ниже приведен пример реализации.

Листинг 8.7. Параллельная реализация `std::for_each`

```
template<typename Iterator, typename Func>
void parallel_for_each(Iterator first, Iterator last, Func f) {
 unsigned long const length = std::distance(first, last);
```

```
 if (!length)
  return;

 unsigned long const min_per_thread = 25;
 unsigned long const max_threads =
  (length + min_per_thread - 1) / min_per_thread;

 unsigned long const hardware_threads =
  std::thread::hardware_concurrency();

 unsigned long const num_threads =
  std::min(
   hardware_threads != 0 ? hardware_threads : 2, max_threads);

 unsigned long const block_size = length / num_threads;

 std::vector<std::future<void> > futures(num_threads - 1); ← (1)
 std::vector<std::thread> threads(num_threads - 1);
 join_threads joiner(threads);

 Iterator block_start = first;
 for (unsigned long i = 0; i < (num_threads - 1); ++i) {
  Iterator block_end = block_start;
  std::advance(block_end, block_size);
  std::packaged_task<void(void)> task( ← (2)
   [=]() {
    std::for_each(block_start, block_end, f);
   });
  futures[i] = task.get_future();
  threads[i] = std::thread(std::move(task)); ← (3)
  block_start = block_end;
 }
 std::for_each(block_start, last, f);
 for (unsigned long i = 0; i < (num_threads - 1); ++i) {
  futures[i].get(); ← (4)
 }
}
```

Структурно код ничем не отличается от приведенного в листинге 8.4, что и неудивительно. Основное различие состоит в том, что в векторе futures хранятся объекты `std::future<void>` **(1)**, потому что рабочие потоки не возвращают значение, а в качестве задачи мы используем простую лямбда-функцию, которая вызывает функцию `f` для элементов из диапазона от `block_start` до `block_end` **(2)**. Это позволяет не передавать конструктору потока **(3)** диапазон. Поскольку рабочие потоки ничего не возвращают, обращения к `futures[i].get()` **(4)** служат только для получения исключений, возникших в рабочих потоках; если мы не хотим передавать исключения, то эти обращения можно вообще опустить.

Реализацию `parallel_for_each` можно упростить, воспользовавшись `std::async`, — точно так же, как мы делали при распараллеливании `std::accumulate`.

Листинг 8.8. Параллельная реализация `std::for_each` с применением `std::async`

```
template<typename Iterator, typename Func>
```

```
void parallel_for_each(Iterator first, Iterator last, Func f) {
 unsigned long const length = std::distance(first, last);

 if (!length)
  return;

 unsigned long const min_per_thread = 25;

 if (length < (2 * min_per_thread)) {
  std::for_each(first, last, f); ← (1)
 } else {
  Iterator const mid_point = first + length / 2;
  std::future<void> first_half = ← (2)
   std::async(&parallel_for_each<Iterator, Func>,
   first, mid_point, f);
  parallel_for_each(mid_point, last, f); ← (3)
  first_half.get();                      ← (4)
 }
}
```

Как и в случае реализации `parallel_accumulate` с помощью `std::async` в листинге 8.5, мы разбиваем данные рекурсивно в процессе выполнения, а не заранее, потому что не знаем, сколько потоков задействует библиотека. На каждом шаге данные делятся пополам, пока их не останется слишком мало для дальнейшего деления. При этом одна половина обрабатывается асинхронно **(2)**, а вторая — непосредственно **(3)**. Когда дальнейшее деление становится нецелесообразным, вызывается `std::for_each` **(1)**. И снова использование `std::async` и функции-члена `get()` объекта `std::future` **(4)** обеспечивает семантику распространения исключения.

Теперь перейдем от алгоритмов, которые выполняют одну и ту же операцию над каждым элементом (к их числу относятся также `std::count` и `std::replace`), к чуть более сложному случаю — `std::find`.

8.5.2. Параллельная реализация `std::find`

Далее будет полезно рассмотреть алгоритм `std::find`, потому что это один из нескольких алгоритмов, которые могут завершаться еще до того, как обработаны все элементы. Например, если уже первый элемент в диапазоне отвечает условию поиска, то рассматривать остальные не имеет смысла. Как мы скоро увидим, это свойство существенно для повышения производительности, и от него напрямую зависит структура параллельной реализации. На этом примере мы продемонстрируем, как порядок доступа к данным может оказать влияние на проектирование программы (см. раздел 8.3.2). К той же категории относятся алгоритмы `std::equal` и `std::any_of`.

Если вы вместе с женой или другом ищете какую-нибудь старую фотографию в сваленных на чердаке альбомах, то вряд ли захотите, чтобы они продолжали перелистывать страницы, когда вы уже нашли то, что нужно. Наверное, вы просто сообщите им, что искомое найдено (быть может, крикнув «Есть!»), чтобы они могли прекратить поиски и заняться чем-нибудь другим. Но многие алгоритмы по природе своей должны обработать каждый элемент и, стало быть, не имеют эквивалента восклицанию «Есть!». Для алгоритмов

типа `std::find` умение «досрочно» прекращать работу — важное свойство, которое нельзя игнорировать. И, следовательно, его нужно учитывать при проектировании кода — закладывать возможность прерывания других задач, когда ответ уже известен, чтобы программа не ждала, пока прочие рабочие потоки обработают оставшиеся элементы.

Без прерывания других потоков последовательная версия может работать быстрее параллельной, потому что прекратит поиск, как только будет найден нужный элемент. Если, например, система поддерживает четыре параллельных потока, то каждый из них должен будет просмотреть четверть полного диапазона, поэтому при наивном распараллеливании каждый поток потратит на просмотр своих элементов четверть всего времени. Если искомый элемент окажется в первой четверти диапазона, то последовательный алгоритм завершится раньше, так как не должен будет просматривать оставшиеся элементы.

Один из способов прервать другие потоки — воспользоваться атомарной переменной в качестве флага и проверять его после обработки каждого элемента. Если флаг поднят, то один из потоков уже нашел нужный элемент, поэтому можно прекращать обработку. При таком способе прерывания мы не обязаны обрабатывать все элементы, и, значит, параллельная версия чаще будет обгонять последовательную. Недостаток состоит в том, что загрузка атомарной переменной — медленная операция, которая тормозит работу каждого потока.

Мы уже знаем о двух способах возврата значений и распространения исключений. Можно использовать массив будущих результатов и объекты `std::packaged_task` для передачи значений и исключений, после чего обработать частичные результаты в главном потоке. Или с помощью `std::promise` устанавливать окончательный результат прямо в рабочем потоке. Все зависит от того, как мы хотим обрабатывать исключения, возникающие в рабочих потоках. Если требуется остановиться при первом возникновении исключения (несмотря на то, что обработаны не все элементы), то можно использовать `std::promise` для передачи значения и исключения. С другой стороны, если мы хотим, чтобы рабочие потоки продолжали поиск, то используем `std::packaged_task`, сохраняем все исключения, а затем повторно возбуждаем одно из них, если искомый элемент не найден.

В данном случае я остановился на `std::promise`, потому что такое поведение больше походит на поведение `std::find`. Надо только не забыть о случае, когда искомого элемента в указанном диапазоне нет. Поэтому необходимо дождаться завершения всех потоков *перед* тем, как получать значение из будущего результата. Если просто блокировать исполнение при обращении к `get()`, то при условии, что искомого элемента нет, мы будем ждать вечно. Получившийся код приведён ниже.

Листинг 8.9. Параллельная реализация алгоритма `find()`

```
template<typename Iterator, typename MatchType>
Iterator parallel_find(Iterator first, Iterator last,
 MatchType match) {
 struct find_element { ← (1)
  void operator()(Iterator begin, Iterator end,
   MatchType match,
   std::promise<Iterator>* result,
   std::atomic<bool>* done_flag) {
   try {
    for(; (begin != end) && !done_flag->load(); ++begin) {← (2)
     if (*begin == match) {
```

```
      result->set_value(begin);← (3)
      done_flag->store(true);  ← (4)
      return;
     }
    }
   } catch (...) { ← (5)
    try {
     result->set_exception(std::current_exception());← (6)
     done_flag->store(true);
    } catch (...) ← (7)
    {}
   }
  }
};

unsigned long const length = std::distance(first, last);

if (!length)
 return last;

unsigned long const min_per_thread = 25;
unsigned long const max_threads =
 (length + min_per_thread − 1) / min_per_thread;

unsigned long const hardware_threads =
 std::thread::hardware_concurrency();

unsigned long const num_threads =
 std::min(
  hardware_threads != 0 ? hardware_threads : 2, max_threads);

unsigned long const block_size = length / num_threads;

std::promise<Iterator> result;     ← (8)
std::atomic<bool> done_flag(false);← (9)
std::vector<std::thread> threads(num_threads − 1); {← (10)
 join_threads joiner(threads);

Iterator block_start = first;
for (unsigned long i = 0; i < (num_threads - 1); ++i) {
 Iterator block_end = block_start;
 std::advance(block_end, block_size);
 threads[i] = std::thread(find_element(), ← (11)
  block_start, block_end, match,
  &result, &done_flag);
 block_start = block_end;
}
find_element()(
 block_start, last, match, &result, &done_flag);← (12)
if (!done_flag.load()) { ← (13)
 return last;
}
```

```
 return result.get_future().get() ← (14)
}
```

В основе своей код в листинге 8.9 похож на предыдущие примеры. На этот раз вся работа производится в операторе вызова, определенном в локальном классе `find_element` **(1)**. Здесь мы в цикле обходим элементы из назначенного потоку блока, проверяя флаг на каждой итерации **(2)**. Если искомый элемент найден, то мы записываем окончательный результат в объект-обещание **(3)** и перед возвратом устанавливаем флаг `done_flag` **(4)**.

Если было возбуждено исключение, то его перехватит универсальный обработчик **(5)** и попытается сохранить исключение в обещании **(6)** перед установкой `done_flag`. Но установка значения объекта-обещания может возбудить исключение, если значение уже установлено, поэтому мы перехватываем и игнорируем любые возникающие здесь исключения **(7)**.

Это означает, что если поток, вызвавший `find_element`, найдет искомый элемент или возбудит исключение, то все остальные потоки увидят поднятый флаг `done_flag` и прекратят работу. Если несколько потоков одновременно найдут искомое или возбудят исключение, то возникнет гонка за установку результата в обещании. Но это безобидная гонка: победивший поток считается «первым», и установленный им результат приемлем.

В самой функции `parallel_find` мы определяем обещание **(8)** и флаг прекращения поиска **(9)**, которые передаем новым потокам вместе с диапазоном для просмотра **(11)**. Кроме того, главный поток пользуется классом `find_element` для поиска среди оставшихся элементов **(12)**. Как уже отмечалось, мы должны дождаться завершения всех потоков перед тем, как проверять результат, потому что искомого элемента может вообще не оказаться. Для этого мы заключаем код запуска и присоединения потоков в блок **(10)**, так что к моменту проверки флага **(13)** все потоки гарантировано присоединены. Если элемент был найден, то, обратившись к функции `get()` объекта `std::future<Iterator>`, мы либо получим результат из обещания, либо возбудим сохраненное исключение.

Как и раньше, в этой реализации предполагается, что мы собираемся использовать все доступные аппаратные потоки или располагаем каким-то механизмом, который позволит заранее определить количество потоков для предварительного разделения между ними работы. И снова мы можем упростить код, воспользовавшись функцией `std::async` и рекурсивным разбиением данных, если готовы принять автоматический механизм масштабирования, скрытый в стандартной библиотеке C++. Реализация `parallel_find` с применением `std::async` приведена в листинге ниже.

Листинг 8.10. Параллельная реализация алгоритма `find()` с применением `std::async`

```
template<typename Iterator, typename MatchType> ← (1)
Iterator parallel_find_impl(Iterator first, Iterator last,
 MatchType match,
 std::atomic<bool>& done) {
 try {
  unsigned long const length = std::distance(first, last);
  unsigned long const min_per_thread = 25;          ← (2)
  if (length < (2 * min_per_thread)) {              ← (3)
   for(; (first != last) && !done.load(); ++first) {← (4)
    if (*first == match) {
     done = true;                                   ← (5)
```

```
      return first;
     }
    }
    return last; ← (6)
   } else {
    Iterator const mid_point = first + (length / 2);     ← (7)
    std::future<Iterator> async_result =
     std::async(&parallel_find_impl<Iterator, MatchType>,← (8)
    mid_point, last, match, std::ref(done));
    Iterator const direct_result =
     parallel_find_impl(first, mid_point, match, done);  ← (9)
    return (direct_result == mid_point) ?
     async_result.get() : direct_result; ← (10)
   }
  } catch (...) {
   done = true; ← (11)
   throw;
  }
}

template<typename Iterator, typename MatchType>
Iterator parallel_find(
 Iterator first, Iterator last, MatchType match) {
 std::atomic<bool> done(false);
 return parallel_find_impl(first, last, match, done); ← (12)
}
```

Желание закончить поиск досрочно при обнаружении совпадения заставило нас ввести флаг, разделяемый между всеми потоками. Этот флаг, следовательно, нужно передавать во все рекурсивные вызовы. Проще всего сделать это, делегировав работу отдельной функции **(1)**, которая принимает дополнительный параметр — ссылку на флаг `done`, передаваемый из главной точки входа **(12)**.

Основная же ветвь кода не таит никаких неожиданностей. Как и во многих предыдущих реализациях, мы задаем минимальное количество элементов, обрабатываемых в одном потоке **(2)**; если размер обеих половин диапазона меньше этой величины, то весь диапазон обрабатывается в текущем потоке **(3)**. Собственно алгоритм сводится к простому циклу — он продолжается, пока не будет достигнут конец заданного диапазона или не установлен флаг `done` **(4)**. При обнаружении совпадения мы устанавливаем флаг `done` и выходим из функции **(5)**. Если мы дошли до конца списка или вышли из цикла, потому что другой поток установил флаг `done`, то возвращаем значение `last`, означающее, что совпадение не найдено **(6)**.

Если диапазон можно разбивать, то мы сначала находим среднюю точку **(7)**, а потом через `std::async` запускаем поиск во второй половине диапазона **(8)**, не забыв передать ссылку на флаг `done` с помощью `std::ref`. Одновременно мы просматриваем первую половину диапазона, рекурсивно вызывая себя же **(9)**. И асинхронный, и рекурсивный вызов могут разбивать диапазон и дальше, если он достаточно велик.

Если прямой рекурсивный вызов вернул `mid_point`, значит, он не нашел совпадения, поэтому нужно получить результат асинхронного поиска. Если и в той половине ничего не было найдено, то мы получим `last` **(10)**. Если «асинхронный» вызов на самом деле был не

асинхронным, а отложенным, то выполняться он начнет именно при обращении к `get()`; в таком случае поиск во второй половине списке вообще не будет производиться, если поиск в первой оказался успешным. Если же асинхронный поиск действительно выполнялся в другом потоке, то деструктор переменной `async_result` будет ждать завершения этого потока, поэтому утечки потоков не произойдет.

Как и раньше, применение `std::async` гарантирует безопасность относительно исключений и распространения исключений вверх по стеку вызовов. Если прямой рекурсивный вызов возбудит исключение, то деструктор будущего результата позаботится о том, чтобы поток, в котором работал асинхронный поиск, завершился до возврата из функции. Если исключение возбудит асинхронный вызов, то оно распространится вверх при вызове `get()` **(10)**. Внешний блок `try/catch` нужен только для того, чтобы установить флаг `done` и обеспечить тем самым быстрое завершение всех потоков в случае исключения **(11)**. Программа правильно работала бы и без этого, по продолжала бы сравнивать элементы до естественного завершения всех потоков.

Существенной особенностью обеих реализаций этого алгоритма (характерной и для других рассмотренных выше параллельных алгоритмов) является тот факт, что элементы могут обрабатываться не в том же порядке, что в стандартном алгоритме `std::find`. Это важный момент при распараллеливании любого алгоритма. Если порядок имеет значение, то обрабатывать элементы параллельно нельзя. В таких алгоритмах, как `parallel_for_each`, порядок обработки независимых элементов не важен, однако `parallel_find` может вернуть элемент, найденный где-то в конце диапазона, хотя имеется другой такой же элемент в начале. Это может оказаться неприятной неожиданностью.

Итак, нам удалось распараллелить `std::find`. В начале этого раздела я говорил, что существуют и другие алгоритмы, которые могут завершаться раньше, чем будут обработаны все элементы. К ним применима такая же техника. В главе 9 мы еще вернёмся к вопросу о прерывании потоков.

В последнем из трех примеров мы направимся в другую сторону и рассмотрим алгоритм `std::partial_sum`. Он не очень широко известен, но интересен с точки зрения распараллеливания, поскольку позволяет проиллюстрировать некоторые дополнительные проектные решения.

8.5.3. Параллельная реализация `std::partial_sum`

Алгоритм `std::partial_sum` вычисляет частичные суммы по диапазону, то есть каждый элемент заменяется суммой всех элементов от начала диапазона до него самого включительно. Таким образом, последовательность 1, 2, 3, 4, 5 преобразуется в 1, (1+2)=3, (1+2+3)=6, (1+2+3+4)=10, (1+2+3+4+5)=15. С точки зрения распараллеливания этот алгоритм интересен тем, что невозможно разбить диапазон на части и обрабатывать каждый блок независимо. Действительно, значение первого элемента необходимо складывать с каждым из остальных элементов, так что независимой обработки не получается.

Один из возможных подходов — вычислить частичные суммы отдельных блоков, а затем прибавить полученное значение последнего элемента в первом блоке ко всем элементам в следующем блоке и так далее. Например, если исходную последовательность 1, 2, 3, 4, 5, 6, 7, 8, 9 разбить на три равных блока, то после первого прохода получатся блоки {1, 3, 6}, {4, 9, 15}, {7, 15, 24}. Если теперь прибавить 6 (значение последнего элемента в первом блоке) ко

всем элементам второго блока, то получится {1, 3, 6}, {10, 15, 21}, {7, 15, 24}. Далее прибавляем последний элемент второго блока (21) ко всем элементам третьего блока и получаем окончательный результат: {1, 3, 6}, {10, 15, 21}, {28, 36, 55}.

Процедуру прибавления частичной суммы предыдущего блока ко всем элементам следующего также можно распараллелить. Если сначала изменить последний элемент блока, то оставшиеся элементы того же блока могут модифицироваться одним потоком, тогда как другой поток одновременно будет модифицировать следующий блок. И так далее. Такое решение хорошо работает, если количество элементов в списке намного превышает количество процессорных ядер, поскольку в этом случае у каждого ядра будет достаточно элементов для обработки на каждом этапе.

Если же процессорных ядер очень много (столько же, сколько элементов в списке, или больше), то описанный подход оказывается не столь эффективен. Если разбить всю работу между процессорами, то на первом шаге каждый процессор будет работать всего с двумя элементами. Но тогда на этапе распространения результатов большинство процессоров будут ждать, и хорошо бы их чем-то запять. В таком случае можно подойти к задаче по-другому. Вместо того чтобы сначала вычислять частичные суммы всех блоков, а затем распространять их от предыдущего к следующему, мы можем распространять суммы по частям. Сначала, как и раньше, вычисляем суммы соседних элементов. На следующем шаге каждое вычисленное значение прибавляется к элементу, отстоящему от него на расстояние 2. Затем новые значения прибавляются к элементам, отстоящим на расстояние 4, и так далее. Если начать с тех же самых девяти элементов, то после первого шага мы получим последовательность 1, 3, 5, 7, 9, 11, 13, 15, 17, в которой правильно вычислены первые два элемента. После второго шага получается последовательность 1, 3, 6, 10, 14, 18, 22, 26, 30, в которой правильны первые четыре элемента. После третьего мы получим последовательность 1, 3, 6, 10, 15, 21, 28, 36, 44, в которой правильны первые восемь элементов. И после четвертого шага получаем окончательный результат 1, 3, 6, 10, 15, 21, 28, 36, 45. Общее количество шагов здесь больше, чем в первом варианте, зато и возможности для распараллеливания на большое число процессоров шире — на каждом шаге каждый процессор может обновлять одно число.

Всего во втором варианте требуется выполнить $\log_2(N)$ шагов по N операций на каждом шаге (по одной на процессор), где N — число элементов в списке. В первом же варианте каждый поток производит N/k операций для вычисления частичной суммы своего блока и еще N/k операций для распространения сумм (здесь k — число потоков). Следовательно, вычислительная сложность первого варианта в терминах количества операций составляет $O(N)$, а второго — $O(N \log(N))$. Однако, если процессоров столько же, сколько элементов в списке, то во втором варианте требуется произвести только $\log(N)$ операций *на каждом процессоре*, тогда как в первом при большом k операции из-за распространения частичных сумм вперед фактически сериализуются. Таким образом, если количество процессоров невелико, то первый алгоритм завершится быстрее, тогда как в массивно параллельных системах победителем окажется второй алгоритм. Это крайнее проявление феномена, обсуждавшегося в разделе 8.2.1.

Но оставим в стороне эффективность и перейдем к коду. В листинге 8.11 приведена реализация первого подхода.

Листинг 8.11. Параллельное вычисление частичных сумм путём разбиения задачи на части

```
template<typename Iterator>
void parallel_partial_sum(Iterator first, Iterator last) {
 typedef typename Iterator::value_type value_type;

 struct process_chunk { ←(1)
  void operator()(Iterator begin, Iterator last,
   std::future<value_type>* previous_end_value,
   std::promise<value_type>* end_value) {
   try {
    Iterator end = last;
    ++end;
    std::partial_sum(begin, end, begin); ←(2)
    if (previous_end_value) {            ←(3)
     value_type& addend = previous_end_value->get();←(4)
     *last += addend;                    ←(5)
     if (end_value) {
      end_value->set_value(*last); ←(6)
     }
     std::for_each(begin, last, [addend](value_type& item) {←(7)
      item += addend;
     });
    } else if (end_value) {
     end_value->set_value(*last); ←(8)
    }
   } catch (...) { ←(9)
    if (end_value) {
     end_value->set_exception(std::current_exception()); ←(10)
    } else {
     throw; ←(11)
    }
   }
  }
 };

 unsigned long const length = std::distance(first, last);

 if (!length)
  return last;

 unsigned long const min_per_thread = 25; ←(12)

 unsigned long const max_threads =
  (length + min_per_thread - 1) / min_per_thread;

 unsigned long const hardware_threads =
  std::thread::hardware_concurrency();

 unsigned long const num_threads =
  std::min(
   hardware_threads!= 0 ? hardware_threads : 2, max_threads);
```

```
    unsigned long const block_size = length / num_threads;

    typedef typename Iterator::value_type value_type;
    std::vector<std::thread> threads(num_threads - 1);← (13)
    std::vector<std::promise<value_type> >
     end_values(num_threads - 1);                    ← (14)
    std::vector<std::future<value_type> >
     previous_end_values;                            ← (15)
    previous_end_values.reserve(num_threads - 1);    ← (16)
    join_threads joiner(threads);
    Iterator block_start = first;
    for (unsigned long i = 0; i < (num_threads - 1); ++i) {
     Iterator block_last = block_start;
     std::advance(block_last, block_size - 1); ← (17)
     threads[i] = std::thread(process_chunk(), ← (18)
      block_start, block_last,
      (i !=0 ) ? &previous_end_values[i - 1] : 0, &end_values[i]);
     block_start = block_last;
     ++block_start; ← (19)
     previous_end_values.push_back(end_values[i].get_future());← (20)
    }
    Iterator final_element = block_start;
    std::advance(
     final_element, std::distance(block_start, last) - 1);← (21)
     process_chunk()(block_start, final_element, ← (22)
      (num_threads > 1) ? &previous_end_values.back() : 0, 0);
}
```

Общая структура этого кода не отличается от рассмотренных ранее алгоритмов: разбиение задачи на блоки с указанием минимального размера блока, обрабатываемого одним потоком **(12)**. В данном случае, помимо вектора потоков **(13)**, мы завели вектор обещаний **(14)**, в котором будут храниться значения последних элементов в каждом блоке, и вектор будущих результатов **(15)**, используемый для получения последнего значения из предыдущего блока. Так как мы знаем, сколько всего понадобится будущих результатов, то можем заранее зарезервировать для них место в векторе **(16)**, чтобы избежать перераспределения памяти при запуске потоков.

Главный цикл выглядит так же, как раньше, только на этот раз нам нужен итератор, который указывает на последний элемент в блоке **(17)**, а не на элемент за последним, чтобы можно было распространить последние элементы поддиапазонов. Собственно обработка производится в объекте-функции `process_chunk`, который мы рассмотрим ниже; в качестве аргументов передаются итераторы, указывающие на начало и конец блока, а также будущий результат, в котором будет храниться последнее значение из предыдущего диапазона (если таковой существует), и объект-обещание для хранения последнего значения в текущем диапазоне **(18)**.

Запустив поток, мы можем обновить итератор, указывающий на начало блока, не забыв продвинуть его на одну позицию вперед (за последний элемент предыдущего блока) **(19)**, и поместить будущий результат, в котором будет храниться последнее значение в текущем блоке, в вектор будущих результатов, откуда он будет извлечён на следующей итерации цикла **(20)**.

Перед тем как обрабатывать последний блок, мы должны получить итератор, указывающий на последний элемент **(21)**, который передается в `process_chunk` **(22)**. Алгоритм `std::partial_sum` не возвращает значения, поэтому по завершении обработки последнего блока больше ничего делать не надо. Можно считать, что операция закончилась, когда все потоки завершили выполнение.

Теперь настало время поближе познакомиться с объектом-функцией `process_chunk`, который собственно и делает всю содержательную работу **(1)**. Сначала вызывается `std::partial_sum` для всего блока, включая последний элемент **(2)**, но затем нам нужно узнать, первый это блок или нет **(3)**. Если это *не* первый блок, то должно быть вычислено значение `previous_end_value` для предыдущего блока, поэтому нам придется его подождать **(4)**. Чтобы добиться максимального распараллеливания, мы затем сразу же обновляем последний элемент **(5)**, так чтобы это значение можно было передать дальше, в следующий блок (если таковой имеется) **(6)**. Сделав это, мы можем с помощью `std::for_each` и простой лямбда-функции **(7)** обновить остальные элементы диапазона.

Если `previous_end_value` *не* существует, то это первый блок, поэтому достаточно обновить `end_value` для следующего блока (опять же, если таковой существует, — может случиться, что блок всего один) **(8)**.

Наконец, если какая-то операция возбуждает исключение, мы перехватываем его **(9)** и сохраняем в объекте-обещании **(10)**, чтобы оно было распространено в следующий блок, когда он попытается получить последнее значение из предыдущего блока **(4)**. В результате все исключения доходят до последнего блока, где и возбуждаются повторно **(11)**, поскольку в этой точке мы гарантированно работаем в главном потоке.

Из-за необходимости синхронизации между потоками этот код не получится переписать под `std::async`. Каждая задача ждет результатов, которые становятся доступны по ходу выполнения других задач, поэтому все задачи должны работать параллельно.

Разобравшись с решением, основанным на распространении результатов обработки предыдущих блоков, посмотрим на второй из описанных алгоритмов вычисления частичных сумм.

Реализация прогрессивно-попарного алгоритма вычисления частичных сумм

Второй способ вычисления частичных сумм, основанный на сложении элементов, расположенных на все большем расстоянии друг от друга, работает лучше всего, когда процессоры могут выполнять операции сложения синхронно. В таком случае никакой дополнительной синхронизации не требуется, потому что все промежуточные результаты можно сразу распространить на следующий нуждающийся в них процессор. Но на практике такие системы встречаются редко — разве что в случаях, когда один процессор может выполнять одну и ту же команду над несколькими элементами данных одновременно с помощью так называемых SIMD-команд (Single-Instruction/Multiple-Data — одиночный поток команд, множественный поток данных). Поэтому мы должны проектировать программу в расчете на общий случай и производить явную синхронизацию на каждом шаге.

Сделать это можно, например, с помощью *барьера* — механизма синхронизации, который заставляет потоки ждать, пока указанное количество потоков достигнет барьера.

Когда все потоки подошли к барьеру, они разблокируются и могут продолжать выполнение. В библиотеке C++11 Thread Library готовая реализация барьера отсутствует, так что нам придется написать свою собственную.

Представьте себе американские горки в парке аттракционов. Если желающих покататься достаточно, то смотритель не запустит состав, пока не будут заняты все места. Барьер работает точно так же: вы заранее задаете число «мест», и потоки будут ждать, пока все «места» заполнятся. Когда ожидающих потоков собирается достаточно, они все получают возможность продолжить выполнение; барьер при этом сбрасывается и начинает ждать следующую партию потоков. Часто такая конструкция встречается в цикле, где на каждой итерации у барьера собираются одни и те же потоки. Идея в том, чтобы все потоки «шли в ногу» — никто не должен забегать вперед. В рассматриваемом алгоритме такой «поток-торопыга» недопустим, потому что он мог бы модифицировать данные, которые еще используются другими потоками, или, наоборот, сам попытался бы использовать еще не модифицированные должным образом данные.

В следующем листинге приведена простая реализация барьера.

Листинг 8.12. Простой класс барьера

```
class barrier {
 unsigned const count;
 std::atomic<unsigned> spaces;
 std::atomic<unsigned> generation;
public:
 explicit barrier (unsigned count_) : ← (1)
 count(count_), spaces(count), generation(0) {}

 void wait() {
  unsigned const my_generation = generation; ← (2)
  if (!--spaces) {← (3)
   spaces = count;← (4)
   ++generation;  ← (5)
  } else {
   while (generation == my_generation) ← (6)
   std::this_thread::yield();          ← (7)
  }
 }
};
```

Здесь при конструировании барьера мы указываем количество «мест» **(1)**, которое сохраняется в переменной count. В начале количество мест у барьера spaces равно `count`. Когда очередной поток выполняет функцию `wait()`, значение `spaces` уменьшается на единицу **(3)**. Как только оно обращается в нуль, количество мест возвращается к исходному значению `count` **(4)**, а переменная `generation` увеличивается, что служит для других потоков сигналом к продолжению работы **(5)**. Если число свободных мест больше нуля, то поток должен ждать. В этой реализации используется простой спинлок **(6)**, который сравнивает текущее значение `generation` с тем, что было запомнено в начале `wait()` **(2)**. Поскольку `generation` изменяется только после того, как все потоки подошли к барьеру **(5)**, мы можем во время ожидания уступить процессор с помощью `yield()` **(7)**, чтобы ожидающий поток не потреблял процессорное время.

Эта реализация действительно очень простая — в ней для ожидания используется спинлок, поэтому она не идеальна для случая, когда потоки могут ждать долго, и совсем не годится в ситуации, когда в каждый момент времени может существовать более count потоков, выполнивших `wait()`. Если требуется, чтобы и в этих случаях барьер работал правильно, то следует использовать более надежную (но и более сложную) реализацию. Кроме того, я ограничился последовательно согласованными операциями над атомарными переменными, потому что они проще для понимания, но вы, возможно, захотите ослабить ограничения на порядок доступа к памяти. Такая глобальная синхронизация дорого обходится в массивно параллельных архитектурах, так как строку кэша, содержащую состояние барьера, приходится передавать всем участвующим процессорам (см. обсуждение перебрасывания кэша в разделе 8.2.2). Поэтому нужно тщательно продумать, является ли это решение оптимальным.

Как бы то ни было, барьер — это именно то, что нам необходимо в данном случае; имеется фиксированное число потоков, которые должны синхронизироваться на каждой итерации цикла. Ну *почти* фиксированное. Как вы, наверное, помните, элементы, расположенные близко к началу списка, получают окончательные значения всего через пару шагов. Это означает, что либо все потоки должны крутиться в цикле, пока не будет обработан весь диапазон, либо барьер должен поддерживать выпадение потоков, уменьшая значение `count`. Я предпочитаю второй вариант, чтобы не заставлять потоки выполнять бессмысленную работу в ожидании завершения последнего шага.

Но тогда нужно сделать `count` атомарной переменной, чтобы ее можно было изменять из нескольких потоков без внешней синхронизации:

```
std::atomic<unsigned> count;
```

Инициализация не меняется, но при переустановке `spaces` теперь нужно явно загружать `spaces` с помощью операции `load()`:

```
spaces = count.load();
```

Больше никаких изменений в `wait()` вносить не надо, но необходима новая функция-член для уменьшения count. Назовем ее `done_waiting()`, потому что с ее помощью поток заявляет, что больше ждать не будет.

```
void done_waiting() {
 --count;                    ← (1)
 if (!--spaces) {            ← (2)
  spaces = count.load();     ← (3)
  ++generation;
 }
}
```

Прежде всего мы уменьшаем `count (1)`, чтобы при следующей переустановке `spaces` было отражено новое, меньшее прежнего, количество ожидающих потоков. Затем уменьшаем количество свободных мест `spaces` **(2)**. Если этого не сделать, то остальные потоки будут ждать вечно, так как при инициализации `spaces` было задано старое, большее, значение. Если текущий поток является последним в партии, то количество мест нужно переустановить и увеличить `generation` на единицу **(3)** — так же, как в `wait()`. Основное отличие от `wait()` заключается в том, что поток не должен ждать — он же сам объявляет, что больше ждать не будет!

Теперь мы готовы написать вторую реализацию алгоритма вычисления частичных сумм. На каждом шаге каждый поток вызывает функцию `wait()` барьера, чтобы все потоки

пересекли его вместе, а, закончив работу, поток вызывает функцию `done_waiting()`, чтобы уменьшить счетчик. Если наряду с исходным диапазоном использовать второй буфер, то барьер обеспечивает необходимую синхронизацию. На каждом шаге потоки либо читают исходный диапазон и записывают новое значение в буфер, либо наоборот — читают буфер и записывают новое значение в исходный диапазон. На следующем шаге исходный диапазон и буфер меняются местами. Тем самым мы гарантируем отсутствие гонки между операциями чтения и записи в разных потоках. Перед выходом из цикла каждый поток должен позаботиться о записи окончательного значения в исходный диапазон. В листинге 8.13 все это собрано воедино.

Листинг 8.13. Параллельная реализация `partial_sum` методом попарных обновлений

```
struct barrier {
 std::atomic<unsigned> count;
 std::atomic<unsigned> spaces;
 std::atomic<unsigned> generation;

 barrier(unsigned count_) :
  count(count_), spaces(count_), generation(0) {}

 void wait() {
  unsigned const gen = generation.load();
  if (!--spaces) {
   spaces = count.load();
   ++generation;
  } else {
   while (generation.load() == gen) {
    std::this_thread::yield();
   }
  }
 }

 void done_waiting() {
  --count;
  if (!--spaces) {
   spaces = count.load();
   ++generation;
  }
 }
};

template<typename Iterator>
void parallel_partial_sum(Iterator first, Iterator last) {
 typedef typename Iterator::value_type value_type;

 struct process_element { ← (1)
  void operator()(Iterator first, Iterator last,
   std::vector<value_type>& buffer,
   unsigned i, barrier& b) {
   value_type& ith_element = *(first + i);
   bool update_source = false;

   for (unsigned step = 0, stride = 1;
```

```
        stride <= i; ++step, stride *= 2) {
    value_type const& source = (step % 2) ? ← (2)
     buffer[i] : ith_element;
    value_type& dest = (step % 2) ?
     ith_element:buffer[i];
    value_type const& addend = (step % 2) ? ← (3)
     buffer[i - stride] : *(first + i - stride);

    dest = source + addend; ← (4)
    update_source = !(step % 2);
    b.wait(); ← (5)
   }
   if (update_source) { ← (6)
    ith_element = buffer[i];
   }
   b.done_waiting(); ← (7)
  }
 };

 unsigned long const length = std::distance(first, last);

 if (length <= 1)
  return;

 std::vector<value_type> buffer(length);
 barrier b(length);
 std::vector<std::thread> threads(length - 1); ← (8)
 join_threads joiner(threads);

 Iterator block_start = first;
 for (unsigned long i = 0; i < (length - 1); ++i) {
  threads[i] = std::thread(process_element(), first, last,← (9)
   std::ref(buffer), i, std::ref(b));
 }
 process_element()(first, last, buffer, length - 1, b);    ← (10)
}
```

Общая структура кода вам, наверное, уже понятна. Имеется класс с оператором вызова (`process_element`), который выполняет содержательную работу **(1)** и вызывается из нескольких потоков **(9)**, хранящихся в векторе **(8)**, а также из главного потока **(10)**. Важное отличие заключается в том, что теперь число потоков зависит от числа элементов в списке, а не от результата, возвращаемого функцией `std::thread::hardware_concurrency`. Я уже говорил, что эта идея не слишком удачна, если только вы не работаете на машине с массивно параллельной архитектурой, где потоки обходятся дешево. Но это неотъемлемая часть самой идеи решения. Можно было бы обойтись и меньшим числом потоков, поручив каждому обработку нескольких значений из исходного диапазона, но тогда при относительно небольшом количестве потоков этот алгоритм оказался бы менее эффективен, чем алгоритм с прямым распространением.

Так или иначе, основная работа выполняется в операторе вызове из класса `process_element`. На каждом шаге берется либо *i*-ый элемент из исходного диапазона, либо

i-ый элемент из буфера **(2)** и складывается с предшествующим элементом, отстоящим от него на расстояние `stride` **(3)**; результат сохраняется в буфере, если мы читали из исходного диапазона, и в исходном диапазоне — если мы читали из буфера **(4)**. Перед тем как переходить к следующему шагу, мы ждем у барьера **(5)**. Работа заканчивается, когда элемент, отстоящий на расстояние `stride`, оказывается слева от начала диапазона. В этом случае мы должно обновить элемент в исходном диапазоне, если сохраняли окончательный результат в буфере **(6)**. Наконец, мы вызываем функцию `done_waiting()` **(7)**, сообщая барьеру, что больше ждать не будем.

Отметим, что это решение не безопасно относительно исключений. Если один из рабочих потоков возбудит исключение в `process_element`, то приложение аварийно завершится. Решить эту проблему можно было бы, воспользовавшись `std::promise` для запоминания исключения, как в реализации `parallel_find` из листинга 8.9, или просто с помощью объекта `std::exception_ptr`, защищенного мьютексом.

Вот и подошли к концу обещанные три примера. Надеюсь, они помогли уложить в сознании соображения, высказанные в разделе 8.1, 8.2, 8.3 и 8.4, и показали, как описанные приемы воплощаются в реальном коде.

8.6. Резюме

Эта глава оказалась очень насыщенной. Мы начали с описания различных способов распределения работы между потоками, в частности предварительного распределения данных и использования нескольких потоков для формирования конвейера. Затем мы остановились на низкоуровневых аспектах производительности многопоточного кода, обратив внимание на феномен ложного разделения и проблему конкуренции за данные. Далее мы перешли к вопросу о том, как порядок доступа к данным влияет на производительность программы. После этого мы поговорили о дополнительных соображениях при проектировании параллельных программ, в частности о безопасности относительно исключений и о масштабируемости. И закончили главу рядом примеров реализации параллельных алгоритмов, на которых показали, какие проблемы могут возникать при проектировании многопоточного кода.

В этой главе пару раз всплывала идея пула потоков — предварительно сконфигурированной группы потоков, выполняющих задачи, назначенные пулу. Разработка хорошего пула потоков — далеко не тривиальное дело. В следующей главе мы рассмотрим некоторые возникающие при этом проблемы, а также другие, более сложные, аспекты управления потоками.

Глава 9.
Продвинутое управление потоками

В этой главе:

- Пулы потоков.
- Учет зависимостей между задачами, адресованными пулу.
- Занимание работ у потоков из пула.
- Прерывание потоков.

В предыдущих главах мы управляли потоками явно — путем создания объектов `std::thread` для каждого потока. В нескольких местах мы видели, что это не всегда желательно, так как приходится самостоятельно управлять временем жизни этих объектов, определять, сколько потоков создать для решения данной задачи с учетом имеющегося оборудования и т.д. В идеале хотелось бы просто разбить код на максимально мелкие блоки, которые можно выполнить параллельно, а потом передать их компилятору и библиотеке, сказав: «Распараллель и обеспечь оптимальную производительность».

В ряде примеров нам встречалась еще одна повторяющаяся тема — мы можем использовать несколько потоков для решения задачи, но хотим, чтобы они завершались досрочно, если выполнено некоторое условие, например: результат уже получен, или произошла ошибка, или пользователь потребовал отменить операцию. В любом случае потокам нужно отправить запрос «Прекратить работу», который означает, что они должны прервать выполняемое задание, прибраться за собой и как можно скорее завершиться.

В этой главе мы рассмотрим различные механизмы управления потоками и задачами и начнем с автоматического выбора числа потоков и распределения задач между ними.

9.1. Пулы потоков

Во многих компаниях сотрудники, которые обычно работают в офисе, время от времени должны выезжать к клиентам или поставщикам, посещать выставки и конференции. Хотя такие поездки могут считаться обязательными, и в любой день в командировке может находиться сразу несколько человек, но для любого конкретного работника промежуток между поездками может составлять месяцы, а то и годы. В таких условиях резервировать машину для каждого работника было бы дорого и непрактично, поэтому компании содержат парк, или *пул машин*, то есть ограниченное количество машин, предоставляемых в распоряжении всем работникам. Когда работнику нужно съездить в командировку, он заказывает машину на определенное время, а по завершении поездки возвращает ее в общий пул. Если в какой-то день свободных машин в пуле не оказалось, то командировку придется перенести на другой день.

В основе *пула потоков* лежит аналогичная идея, только в общее распоряжение предоставляются не машины, а потоки. Во многих системах бессмысленно заводить отдельный поток для каждой задачи, которая потенциально может выполняться одновременно с другими, но тем не менее хотелось бы воспользоваться преимуществами параллелизма там, где это возможно. Именно это и позволяет сделать пул потоков: задачи, которые в принципе могут выполняться параллельно, отправляются в пул, а тот ставит их в очередь ожидающих работ. Затем один из *рабочих потоков* забирает задачу из очереди, исполняет ее и идет за следующей задачей.

При разработке пула потоков нужно принять несколько важных проектных решений, например: сколько потоков будет в пуле, как эффективнее всего назначать потоки задачам, можно ли будет ждать завершения задачи. В этом разделе мы рассмотрим несколько реализаций пула потоков, начав с самого простого.

9.1.1. Простейший пул потоков

В простейшем случае пул состоит из фиксированного числа *рабочих потоков* (обычно равного значению, которое возвращает функция `std::thread::hardware_concurrency()`). Когда у программы появляется какая-то работа, она вызывает функцию, которая помещает эту работу в очередь. Рабочий поток забирает работу из очереди, выполняет указанную в ней задачу, после чего проверяет, есть ли в очереди другие работы. В этой реализации никакого механизма ожидания завершения задачи не предусмотрело. Если это необходимо, то вы должны будете управлять синхронизацией самостоятельно.

В следующем листинге приведена реализация такого пула потоков.

Листинг 9.1. Простой пул потоков

```
class thread_pool {
 std::atomic_bool done;
 thread_safe_queue<std::function<void()> > work_queue;← (1)
 std::vector<std::thread> threads; ← (2)
 join_threads joiner; ← (3)

 void worker_thread() {
```

```
    while (!done) { ← (4)
     std::function<void()> task;
     if (work_queue.try_pop(task)) { ← (5)
      task(); ← (6)
     } else {
      std::this_thread::yield(); ← (7)
     }
    }
   }

  public:
   thread_pool():
    done(false), joiner(threads) {
    unsigned const thread_count =
     std::thread::hardware_concurrency();← (8)
    try {
     for (unsigned i = 0; i < thread_count; ++i) {
      threads.push_back(
       std::thread(&thread_pool::worker_thread, this)); ← (9)
     }
    } catch (...) {
     done = true; ← (10)
     throw;
    }
   }

   ~thread_pool() {
    done = true; ← (11)
   }

   template<typename FunctionType>
   void submit(FunctionType f) {
    work_queue.push(std::function<void()>(f)); ← (12)
   }
  };
```

Здесь мы определили вектор рабочих потоков **(2)** и используем одну из потокобезопасных очередей из главы 6 **(1)** для хранения очереди работ. В данном случае пользователь не может ждать завершения задачи, а задача не может возвращать значения, поэтому для инкапсуляции задач можно использовать тип `std::function<void()>`. Функция `submit()` обертывает переданную функцию или допускающий вызов объект в объект `std::function<void()>` и помещает его в очередь **(12)**.

Потоки запускаются в конструкторе; их количество равно значению, возвращаемому функцией `std::thread::hardware_concurrency()`, то есть мы создаем столько потоков, сколько может поддержать оборудование **(8)**. Все эти потоки исполняют функцию-член нашего класса `worker_thread()` **(9)**.

Запуск потока может завершиться исключением, поэтому необходимо позаботиться о том, чтобы уже запущенные к этому моменту потоки корректно завершались. Для этого мы включили блок `try-catch`, который в случае исключения поднимает флаг `done` **(10)**. Кроме того, мы воспользовались классом `join_threads` из главы 8 **(3)**, чтобы обеспечить

присоединение всех потоков. То же самое происходит в деструкторе: мы просто поднимаем флаг `done` **(11)**, а объект `join_threads` гарантирует, что потоки завершатся до уничтожения пула. Отметим, что порядок объявления членов важен: и флаг `done` и объект `worker_queue` должны быть объявлены раньше вектора `threads`, который, в свою очередь, должен быть объявлен раньше `joiner`. Только тогда деструкторы членов класса будут вызываться в правильном порядке; в частности, нельзя уничтожать очередь раньше, чем остановлены все потоки.

Сама функция `worker_thread` проста до чрезвычайности: в цикле, который продолжается, пока не поднят флаг `done` **(4)**, она извлекает задачи из очереди **(5)** и выполняет их **(6)**. Если в очереди нет задач, функция вызывает `std::this_thread::yield()` **(7)**, чтобы немного отдохнуть и дать возможность поработать другим потокам.

Часто даже такого простого пула потоков достаточно, особенно если задачи независимы, не возвращают значений и не выполняют блокирующих операций. Но бывает и по-другому: во-первых, у программы могут быть более жесткие требования, а, во-вторых, в таком пуле возможны проблемы, в частности, из-за взаимоблокировок. Кроме того, в простых случаях иногда лучше прибегнуть к функции `std::async`, как неоднократно демонстрировалось в главе 8. В этой главе мы рассмотрим и более изощренные реализации пула потоков с дополнительными возможностями, которые призваны либо удовлетворить особые потребности пользователя, либо уменьшить количество потенциальных ошибок. Для начала разрешим ожидать завершения переданной пулу задачи.

9.1.2. Ожидание задачи, переданной пулу потоков

В примерах из главы 8, где потоки запускались явно, главный поток после распределения работы между потоками всегда ждал завершения запущенных потоков. Тем самым гарантировалось, что вызывающая программа получит управление только после полного завершения задачи. При использовании пула потоков ждать нужно завершения задачи, переданной пулу, а не самих рабочих потоков. Это похоже на то, как мы ждали будущих результатов при работе с `std::async` в главе 8. В случае простого пула потоков, показанного в листинге 9.1, организовывать ожидание придется вручную, применяя механизмы, описанные в главе 4: условные переменные и будущие результаты. Это усложняет код; намного удобнее было бы ждать задачу напрямую.

За счет переноса сложности в сам пул потоков мы *сумеем* добиться желаемого. Функция `submit()` могла бы возвращать некий описатель задачи, по которому затем можно было бы ждать ее завершения. Сам описатель должен был бы инкапсулировать условную переменную или будущий результат. Это упростило бы код, пользующийся пулом потоков.

Частный случай ожидания завершения запущенной задачи возникает, когда главный поток нуждается в вычисленном ей результате. Мы уже встречались с такой ситуацией выше, например, в функции `parallel_accumulate()` из главы 2. В таком случае путем использования будущих результатов мы можем объединить ожидание с передачей результата. В листинге 9.2 приведен код модифицированного пула потоков, который разрешает ожидать завершения задачи и передает возвращенный ей результат ожидающему потоку. Поскольку экземпляры класса `std::packaged_task<>` допускают только *перемещение*, но не *копирование*, мы больше не можем воспользоваться классом `std::function<>` для обертывания элементов очереди, потому что `std::function<>` требует, чтобы в обернутых

объектах-функциях был определён копирующий конструктор. Вместо этого мы напишем специальный класс-обертку, умеющий работать с объектами, обладающими только перемещающим конструктором. Это простой маскирующий тип класс (type-erasure class), в котором определён оператор вызова. Нам нужно поддержать функции, которые не принимают параметров и возвращают `void`, поэтому оператор всего лишь вызывает виртуальный метод `call()`, который в свою очередь вызывает обернутую функцию.

Листинг 9.2. Пул потоков, ожидающий завершения задачи

```
class function_wrapper {
 struct impl_base {
  virtual void call() = 0;
  virtual ~impl_base() {}
 };

 std::unique_ptr<impl_base> impl;

 template<typename F>
 struct impl_type: impl_base {
  F f;
  impl_type(F&& f_): f(std::move(f_)) {}
  void call() { f(); }
 };

public:
 template<typename F> function_wrapper(F&& f):
  impl(new impl_type<F>(std::move(f))) {}

 void operator()() { impl->call(); }

 function_wrapper() = default;

 function_wrapper(function_wrapper&& other):
  impl(std::move(other.impl)) {}

 function_wrapper& operator=(function_wrapper&& other) {
  impl = std::move(other.impl);
  return *this;
 }

 function_wrapper(const function_wrapper&) = delete;
 function_wrapper(function_wrapper&) = delete;
 function_wrapper& operator=(const function_wrapper&) = delete;
};

class thread_pool {
 thread_safe_queue<function_wrapper> work_queue;←┐
                                                 | Используем
 void worker_thread()                            | function_
 {                                               | wrapper
  while (!done)                                  | вместо std::
  {                                              | function
```

```
    function_wrapper task;                                    ↵
    if (work_queue.try_pop(task))
     task();
    else
     std::this_thread::yield();
   }
  }

 public:
  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type>↵ (1)
  submit(FunctionType f) {
   typedef typename std::result_of<FunctionType()>::type
    result_type;                                              ↵ (2)
   std::packaged_task<result_type()> task(std::move(f));↵ (3)
   std::future<result_type> res(task.get_future());           ↵ (4)
   work_queue.push(std::move(task));                          ↵ (5)
   return res;                                                ↵ (6)
  }

  // остальное, как и раньше
 };
```

Прежде всего отметим, что модифицированная функция `submit()` **(1)** возвращает объект `std::future<>`, который будет содержать возвращенное задачей значение и позволит вызывающей программе ждать ее завершения. Для этого нам необходимо знать тип значения, возвращаемого переданной функцией `f`, и здесь на помощь приходит шаблон `std::result_of<>`: `std::result_of<FunctionType()>::type` — это тип результата, возвращенного вызовом объекта типа `FunctionType` (например, `f`) без аргументов. Выражение `std::result_of<>` мы используем также в определении псевдонима типа `result_type` **(2)** внутри функции.

Затем `f` обертывается объектом `std::packaged_task<result_type()>` **(3)**, потому что `f` — функция или допускающий вызов объект, который не принимает параметров и возвращает результат типа `result_type`. Теперь мы можем получить будущий результат из `std::packaged_task<>` **(4)**, перед тем как помещать задачу в очередь **(5)** и возвращать будущий результат **(6)**. Отметим, что при помещении задачи в очередь мы должны использовать функцию `std::move()`, потому что класс std::packaged_task<> не допускает копирования. Именно поэтому в очереди хранятся объекты `function_wrapper`, а не объекты типа.

Этот пул позволяет ожидать завершения задач и получать возвращаемые ими результаты. В листинге ниже показано, как выглядит функция `parallel_accumulate`, работающая с таким пулом потоков.

Листинг 9.3. Функция `parallel_accumulate`, реализованная с помощью пула потоков, допускающего ожидание задач

```
template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
 unsigned long const length = std::distance(first, last);
```

```
 if (!length)
  return init;

 unsigned long const block_size = 25;
 unsigned long const num_blocks =
  (length + block_size - 1) / block_size; ← (1)

 std::vector<std::future<T> > futures(num_blocks-1);
 thread_pool pool;

 Iterator block_start = first;
 for (unsigned long i = 0; i < (num_blocks - 1); ++i) {
  Iterator block_end = block_start;
  std::advance(block_end, block_size);
  futures[i] = pool.submit(accumulate_block<Iterator, T>());← (2)
  block_start = block_end;
 }
 T last_result =
  accumulate_block<Iterator, T>()(block_start, last);
 T result = init;
 for (unsigned long i = 0; i < (num_blocks - 1); ++i) {
  result += futures[i].get();
 }
 result += last_result;
 return result;
}
```

Сравнивая этот код с листингом 8.4, следует обратить внимание на две вещи. Во-первых, мы работаем с количеством блоков (`num_blocks` **(1)**), а не потоков. Чтобы в полной мере воспользоваться масштабируемостью пула потоков, мы должны разбить работу на максимально мелкие блоки, с которыми имеет смысл работать параллельно. Если потоков в пуле немного, то каждый поток будет обрабатывать много блоков, но по мере роста числа потоков, поддерживаемых оборудованием, будет расти и количество блоков, обрабатываемых параллельно.

Но, выбирая «максимально мелкие блоки, с которыми имеет смысл работать параллельно», будьте осторожны. Отправка задачи пулу потоков, выбор ее рабочим потоком из очереди и передача возвращенного значения с помощью `std::future<>` — всё это операции не бесплатные, и для совсем мелких задач они не окупятся. *Если размер задачи слишком мал, то программа, в которой используется пул потоков, может работать медленнее, чем однопоточная.*

В предположении, что размер блока выбран разумно, вам не надо заботиться об упаковке задач, получении будущих результатов и хранении объектов `std::thread`, чтобы впоследствии их можно было присоединить; все это пул берет на себя. Вам остается лишь вызвать функцию `submit()`, передав ей свою задачу **(2)**.

Пул потоков обеспечивает также безопасность относительно исключений. Любое возбужденное задачей исключение передается через будущий результат, возвращенный `submit()`, а, если выход из функции происходит в результате исключения, то деструктор пула потоков снимет еще работающие задачи и дождется завершения потоков, входящих в пул.

Эта схема работает для простых случаев, когда задачи независимы. Но не годится, когда одни задачи зависят от других, также переданных пулу.

9.1.3. Задачи, ожидающие других задач

В этой книге я уже неоднократно приводил пример алгоритма Quicksort. Его идея проста — подлежащие сортировке данные разбиваются на две части: до и после опорного элемента (в смысле заданной операции сравнения). Затем обе части рекурсивно сортируются и объединяются для получения полностью отсортированной последовательности. При распараллеливании алгоритма надо позаботиться о том, чтобы рекурсивные вызовы задействовали имеющийся аппаратный параллелизм.

В главе 4, где этот пример впервые был представлен, мы использовали `std::async` для выполнения одного из рекурсивных вызовов на каждом шаге и оставляли библиотеке решение о том, запускать ли новый поток или сортировать синхронно при обращении к `get()`. Этот подход неплохо работает — каждая задача либо выполняется в отдельном потоке, либо в тот момент, когда нужны ее результаты.

В главе 8 мы переработали эту реализацию, продемонстрировав альтернативный подход, когда количество потоков фиксировано и определяется уровнем аппаратного параллелизма. В данном случае мы воспользовались стеком ожидающих сортировки блоков. Разбивая на части предложенные для сортировки данные, каждый поток помещал один блок в стек, а второй сортировал непосредственно. Бесхитростное ожидание завершения сортировки второго блока могло бы закончиться взаимоблокировкой, потому что число потоков ограничено, и некоторые из них ждут. Очень легко оказаться в ситуации, когда все потоки ждут завершения сортировки блоков, и ни один ничего не делает. Тогда мы решили эту проблему, заставив поток извлекать блоки из стека и сортировать их, пока тот конкретный блок, которого он ждет, еще не отсортирован.

Точно такая же проблема возникает при использовании одного из рассмотренных выше простых пулов потоков вместо `std::async`, как в примере из главы 4. Число потоков тоже ограничено, и может случиться так, что все они будут ждать задач, которые еще запланированы, так как нет свободных потоков. И решение должно быть таким же,

как в главе 8: обрабатывать стоящие в очереди блоки во время ожидания завершения сортировки своего блока. Но если мы применяем пул потоков для управления списком задач и их ассоциациями с потоками — а именно в этом и состоит смысл пула потоков, — то доступа к списку задач у нас нет. Поэтому необходимо модифицировать сам пул, чтобы он делал это автоматически.

Проще всего будет добавить в класс `thread_pool` новую функцию, чтобы исполнять задачу из очереди и управлять циклом самостоятельно. Так мы и поступим. Более развитые реализации пула могли бы включить дополнительную логику в функцию ожидания или добавить другие функции ожидания для обработки этого случая, быть может, даже назначая приоритеты ожидаемым задачам. В листинге ниже приведена новая функция `run_pending_task()`, а модифицированный алгоритм Quicksort, в котором она используется, показан в листинге 9.5.

Листинг 9.4. Реализация функции `run_pending_task()`

```
void thread_pool::run_pending_task() {
 function_wrapper task;
 if (work_queue.try_pop(task)) {
  task();
 } else {
```

```
    std::this_thread::yield();
   }
  }
```

Код `run_pending_task()` вынесен из главного цикла внутри функции `worker_thread()`, которую теперь можно будет изменить, так чтобы она вызывала `run_pending_task()`. Функция `run_pending_task()` пытается получить задачу из очереди и в случае успеха выполняет ее; если очередь пуста, то функция уступает управление ОС, чтобы та могла запланировать другой поток. Показанная ниже реализация Quicksort гораздо проще, чем версия в листинге 8.1, потому что вся логика управления потоками перенесена в пул.

Листинг 9.5. Реализация Quicksort на основе пула потоков

```
template<typename T>
struct sorter {     ← (1)
 thread_pool pool; ← (2)

 std::list<T> do_sort(std::list<T>& chunk_data) {
  if (chunk_data.empty()) {
   return chunk_data;
  }

  std::list<T> result;
  result.splice(result.begin(), chunk_data, chunk_data.begin());
  T const& partition_val = *result.begin();

  typename std::list<T>::iterator divide_point =
   std::partition(chunk_data.begin(), chunk_data.end(),
    [&](T const& val){ return val < partition_val; });

  std::list<T> new_lower_chunk;
  new_lower_chunk.splice(new_lower_chunk.end(),
   chunk_data, chunk_data.begin(),
   divide_point);

  std::future<std::list<T> > new_lower = ← (3)
  pool.submit(std::bind(&sorter::do_sort, this,
   std::move(new_lower_chunk)));

  std::list<T> new_higher(do_sort(chunk_data));
  result.splice(result.end(), new_higher);
  while (!new_lower.wait_for(std::chrono::seconds(0)) ==
   std::future_status::timeout) {
   pool.run_pending_task(); ← (4)
  }

  result.splice(result.begin(), new_lower.get());
  return result;
 }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input) {
 if (input.empty()) {
```

```
  return input;
 }

 sorter<T> s;
 return s.do_sort(input);
}
```

Как и в листинге 8.1, реальная работа делегируется функции-члену `do_sort()` шаблона класса `sorter` **(1)**, хотя в данном случае этот шаблон нужен лишь для обертывания экземпляра `thread_pool` **(2)**.

Управление потоками и задачами теперь свелось к отправке задачи пулу **(3)** и исполнению находящихся в очереди задач в цикле ожидания **(4)**. Это гораздо проще, чем в листинге 8.1, где нужно было явно управлять потоками и стеком подлежащих сортировке блоков. При отправке задачи пулу мы используем функцию `std::bind()`, чтобы связать указатель `this` с `do_sort()` и передать подлежащий сортировке блок. В данном случае мы вызываем `std::move()`, чтобы данные `new_lower_chunk` перемещались, а не копировались.

Мы решили проблему взаимоблокировки, возникающую из- за того, что одни потоки ждут других, но этот пул все еще далек от идеала. Отметим хотя бы, что все вызовы `submit()` и `run_pending_task()` обращаются к одной и той же очереди. В главе 8 мы видели, что модификация одного набора данных из разных потоков может негативно сказаться на производительности, стало быть, с этим нужно что-то делать.

9.1.4. Предотвращение конкуренции за очередь работ

Всякий раз, как поток вызывает функцию `submit()` экземпляра пула потоков, он помещает новый элемент в единственную разделяемую очередь работ. А рабочие потоки постоянно извлекают элементы из той же очереди. Следовательно, по мере увеличения числа процессоров будет возрастать конкуренция за очередь работ. Это может ощутимо отразиться на производительности; даже при использовании свободной от блокировок очереди, в которой нет явного ожидания, драгоценное время может тратиться на перебрасывание кэша.

Чтобы избежать перебрасывания кэша, мы можем завести по одной очереди работ на каждый поток. Тогда каждый поток будет помещать новые элементы в свою собственную очередь и брать работы из глобальной очереди работ только тогда, когда в его очереди работ нет. В следующем листинге приведена реализация с использованием переменной типа `thread_local`, благодаря которой каждый поток обладает собственной очередью работ в дополнение к глобальной.

Листинг 9.6. Пул с очередями в поточно-локальной памяти

```
class thread_pool {
 thread_safe_queue<function_wrapper> pool_work_queue;

 typedef std::queue<function_wrapper> local_queue_type;← (1)
 static thread_local std::unique_ptr<local_queue_type>
  local_work_queue; ← (2)

 void worker_thread() {
```

```
    local_work_queue.reset(new local_queue_type);← (3)
    while (!done) {
     run_pending_task();
    }
   }

  public:
   template<typename FunctionType>
   std::future<typename std::result_of<FunctionType()>::type>
   submit(FunctionType f) {
    typedef typename std::result_of<FunctionType()>::type
     result_type;

    std::packaged_task<result_type()> task(f);
    std::future<result_type> res(task.get_future());
    if (local_work_queue) { ← (4)
     local_work_queue->push(std::move(task));
    } else {
     pool_work_queue.push(std::move(task)); ← (5)
    }
    return res;
   }

   void run_pending_task() {
    function_wrapper task;
    if (local_work_queue && !local_work_queue->empty()) {← (6)
     task = std::move(local_work_queue->front());
     local_work_queue->pop();
     task();
    } else if(pool_work_queue.try_pop(task)) { ← (7)
     task();
    } else {
     std::this_thread::yield();
    }
   }

   // остальное, как и раньше
  };
```

Для хранения очереди работ в поточно-локальной памяти мы воспользовались указателем `std::unique_ptr<>` **(2)**, потому что не хотим, чтобы у потоков, не входящих в пул, была очередь; этот указатель инициализируется в функции `worker_thread()` до начала цикла обработки **(3)**. Деструктор `std::unique_ptr<>` позаботится об уничтожении очереди работ по завершении потока.

Затем функция `submit()` проверяет, есть ли у текущего потока очередь работ **(4)**. Если есть, то это поток из пула, и мы можем поместить задачу в локальную очередь. В противном случае задачу следует помещать в очередь пула, как и раньше **(5)**.

Аналогичная проверка имеется в функции `run_pending_task()` **(6)**, только на этот раз нужно еще проверить, есть ли что-нибудь в локальной очереди. Если есть, то можно извлечь элемент из начала очереди и обработать его. Обратите внимание, что локальная очередь может быть обычным объектом `std::queue<>` **(1)**, так как к ней обращается только один поток. Если в локальной очереди задач нет, то мы проверяем очередь пула, как и раньше **(7)**.

Таким образом мы действительно уменьшаем конкуренцию, но если распределение работ неравномерно, то может случиться, что в очереди одного потока скопится много задач, тогда как остальным будет нечем заняться. Например, в случае Quicksort только самый первый блок попадает в очередь пула, а остальные окажутся в локальной очереди того потока, который этот блок обработал. Это сводит на нет всю идею пула потоков.

К счастью, у этой проблемы есть решение: позволить потоку *занимать* (steal) работы из очередей других потоков, если ни в его собственной, ни в глобальной очереди ничего нет.

9.1.5. Занимание работ

Если мы хотим, чтобы «безработный» поток мог брать работы из очереди другого потока, то эта очередь должна быть доступна занимающему потоку в `run_pending_tasks()`. Для этого каждый поток должен зарегистрировать свою очередь в пуле или получать очередь от пула. Кроме того, необходимо позаботиться о надлежащей синхронизации и защите очереди работ, чтобы не нарушались инварианты.

Можно написать свободную от блокировок очередь, которая позволит потоку-владельцу помещать и извлекать элементы с одного конца, а другим потокам — занимать элементы с другого конца, однако реализация такой очереди выходит за рамки данной книги. Чтобы продемонстрировать идею, мы поступим проще — воспользуемся мьютексом для защиты данных очереди. Мы надеемся, что занимание работ — редкое событие, поэтому конкуренция за мьютекс будет невелика, и накладные расходы на такую очередь окажутся минимальны. Ниже приведена простая реализация с блокировками.

Листинг 9.7. Очередь с блокировкой, допускающей занимание работ

```
class work_stealing_queue {
private:
 typedef function_wrapper data_type;
 std::deque<data_type> the_queue; ← (1)
 mutable std::mutex the_mutex;

public:
 work_stealing_queue() {}
 work_stealing_queue(const work_stealing_queue& other)=delete;
 work_stealing_queue& operator=(
  const work_stealing_queue& other)=delete;

 void push(data_type data) { ← (2)
  std::lock_guard<std::mutex> lock(the_mutex);
  the_queue.push_front(std::move(data));
 }

 bool empty() const {
  std::lock_guard<std::mutex> lock(the_mutex);
  return the_queue.empty();
 }

 bool try_pop(data_type& res) { ← (3)
  std::lock_guard<std::mutex> lock(the_mutex);
```

```
  if (the_queue.empty()) {
   return false;
  }

  res = std::move(the_queue.front());
  the_queue.pop_front();
  return true;
 }

 bool try_steal(data_type& res) { ← (4)
  std::lock_guard<std::mutex> lock(the_mutex);
  if (the_queue.empty()) {
   return false;
  }

  res = std::move(the_queue.back());
  the_queue.pop_back();
  return true;
 }
};
```

Этот класс является простой оберткой вокруг `std::deque<function_wrapper>` **(1)**, которая защищает все операции доступа к очереди с помощью мьютекса. Функции `push()` **(2)** и `try_ pop()` **(3)** работают с началом очереди, а функция `try_steal()` — с концом **(4)**.

Получается, что эта «очередь» для потока-владельца на самом деле является стеком, обслуживаемым согласно дисциплине «последним пришёл, первым обслужен», — задача, которая была помещена последней, извлекается первой. С точки зрения кэш-памяти это даже может повысить производительность, так как относящиеся к последней задаче данные с большей вероятностью окажутся в кэше, чем данные, относящиеся к предыдущей задаче. К тому же, такая дисциплина прекрасно подходит для алгоритмов типа Quicksort. В предшествующих реализациях каждое обращение к `do_sort()` помещает элемент в очередь, а затем ждет его. Обрабатывая последний помещенный в очередь элемент первым, мы гарантируем, что блок, необходимый текущему вызову для завершения работы, будет обработан раньше блоков, нужных другим ветвям, а, значит, уменьшается как количество активных задач, так и занятый размер стека. Функция `try_steal()` извлекает элементы из противоположного по сравнению с `try_pop()` конца очереди, чтобы минимизировать конкуренцию; в принципе, можно было бы применить технику, обсуждавшуюся в главах 6 и 7, чтобы поддержать одновременные обращения к `try_pop()` и `try_steal()`.

Итак, теперь у нас есть замечательная очередь работ, допускающая занимание. Но как воспользоваться ей в пуле потоков? Ниже приведена одна из возможных реализаций.

Листинг 9.8. Пул потоков с использованием занимания работ

```
class thread_pool {
 typedef function_wrapper task_type;

 std::atomic_bool done;
 thread_safe_queue<task_type> pool_work_queue;
 std::vector<std::unique_ptr<work_stealing_queue> > queues;← (1)
 std::vector<std::thread> threads;
 join_threads joiner;
```

```
 static thread_local work_stealing_queue* local_work_queue;← (2)
 static thread_local unsigned my_index;

 void worker_thread(unsigned my_index_) {
  my_index = my_index_;
  local_work_queue = queues[my_index].get(); ← (3)
  while (!done) {
   run_pending_task();
  }
 }

 bool pop_task_from_local_queue(task_type& task) {
  return local_work_queue && local_work_queue->try_pop(task);
 }

 bool pop_task_from_pool_queue(task_type& task) {
  return pool_work_queue.try_pop(task);
 }

 bool pop_task_from_other_thread_queue(task_type& task) { ← (4)
  for (unsigned i = 0; i < queues.size(); ++i) {
   unsigned const index = (my_index + i + 1) % queues.size();← (5)
   if (queues[index]->try_steal(task)) {
    return true;
   }
  }

  return false;
 }

public:
 thread_pool() :
  done(false), joiner(threads) {
  unsigned const thread_count =
   std::thread::hardware_concurrency();
  try {
   for (unsigned i = 0; i < thread_count; ++i) {
    queues.push_back(std::unique_ptr<work_stealing_queue> (← (6)
     new work_stealing_queue));
    threads.push_back(
     std::thread(&thread_pool::worker_thread, this, i));
   }
  } catch (...) {
   done = true;
   throw;
  }
 }

 ~thread_pool() {
  done = true;
 }
```

```
 template<typename FunctionType>
 std::future<typename std::result_of<FunctionType()>::type> submit(
  FunctionType f) {
  typedef typename std::result_of<FunctionType()>::type
   result_type;

  std::packaged_task<result_type()> task(f);
  std::future<result_type> res(task.get_future());
  if (local_work_queue) {
   local_work_queue->push(std::move(task));
  } else {
   pool_work_queue.push(std::move(task));
  }
  return res;
 }

 void run_pending_task() {
  task_type task;
  if (pop_task_from_local_queue(task) || ← (7)
      pop_task_from_pool_queue (task) || ← (8)
      pop_task_from_other_thread_queue(task)) { ← (9)
   task();
  } else {
   std::this_thread::yield();
  }
 }
};
```

Этот код очень похож на код из листинга 9.6. Первое отличие состоит в том, что локальная очередь каждого потока — объект класса `work_stealing_queue`, а не просто `std::queue<>` **(2)**. Новый поток не выделяет очередь для себя самостоятельно; это делает конструктор пула потоков **(6)**, и он же сохраняет новую очередь в списке очередей для данного пула **(1)**. Индекс очереди в списке передаётся функции потока и используется затем для получения указателя на очередь **(3)**. Это означает, что пул потоков может получить доступ к очереди, когда пытается занять задачу для потока, которому нечего делать. Новая версия `run_pending_task()` сначала пытается получить задачу из очереди исполняемого потока **(7)**, затем из очереди пула **(8)** и, наконец, из очереди другого потока **(9)**.

Функция `pop_task_from_other_thread_queue()` **(4)** обходит очереди, принадлежащие всем потокам пула, пытаясь занять задачу у каждой. Чтобы не случилось так, что все потоки занимают задачи у первого потока в списке, каждый поток начинает просмотр с позиции, равной его собственному индексу **(5)**.

Теперь у нас имеется пул потоков, пригодный для самых разных целей. Разумеется, есть масса способов улучшить его для работы в конкретной ситуации, но это я оставляю в качестве упражнения для читателя. В частности, мы совсем не исследовали идею динамического изменения размера пула, так чтобы обеспечить оптимальное использование процессоров, даже когда потоки блокированы в ожидании какого-то события, например, завершения ввода/вывода или освобождения мьютекса.

Следующим в нашем списке «продвинутых» приёмов управления потоками стоит прерывание потоков.

9.2. Прерывание потоков

Часто бывает необходимо сообщить долго работающему потоку о том, что пришло время остановиться. Например, потому что это рабочий поток пула, а мы собираемся уничтожить сам пул, или потому что пользователь отменил работу, выполняемую этим потоком. Причин миллион. Но идея в любом случае одна и та же: послать из одного потока другому сигнал с требованием прекратить работу до ее естественного завершения, и сделать это так, чтобы поток завершился корректно, а не просто выбить почву у него из-под ног.

Можно было бы придумывать такой механизм специально для каждого случая, но это, пожалуй, перебор. Мало того что общий механизм в дальнейшем упростит написание кода, так он еще и позволит писать код, допускающий прерывание, не заботясь о том, где конкретно он используется. Стандарт C++11 такого механизма не предоставляет, но реализовать его самостоятельно не слишком сложно. Я покажу, как это сделать, но сначала взгляну на проблему с точки зрения интерфейса запуска и прерывания потока, а не с точки зрения самого прерываемого потока.

9.2.1. Запуск и прерывание другого потока

Начнем с рассмотрения внешнего интерфейса. Что нам нужно от допускающего прерывание потока? На самом элементарном уровне интерфейс должен быть таким же, как у `std::thread`, но с дополнительной функцией `interrupt()`:

```
class interruptible_thread {
public:
 template<typename FunctionType>
 interruptible_thread(FunctionType f);
 void join();
 void detach();
 bool joinable() const;
 void interrupt();
};
```

В реализации можно было бы использовать `std::thread` для управления потоком и какую-то структуру данных для обработки прерывания. А как это выглядит с точки зрения самого потока? Как минимум, нужна возможность сказать: «Меня можно прерывать здесь», то есть нам требуется *точка прерывания*. Чтобы не передавать дополнительные данные, соответствующая функция должна вызываться без параметров: `interruption_point()`. Отсюда следует, что относящаяся к прерываниям структура данных должна быть доступна через переменную типа `thread_local`, которая устанавливается при запуске потока. Поэтому, когда поток обращается к функции `interruption_point()`, та проверяет структуру данных для текущего исполняемого потока. С реализацией `interruption_point()` мы познакомимся ниже.

Флаг типа `thread_local` — основная причина, по которой мы не можем использовать для управления потоком просто класс `std::thread`; память для него нужно выделить таким образом, чтобы к ней имел доступ как экземпляр `interruptible_thread`, так и вновь запущенный поток. Для этого функцию, переданную конструктору, можно специальным образом обернуть перед тем, как передавать конструктору `std::thread`. Как это делается, показано в следующем листинге.

Листинг 9.9. Простая реализация `interruptible_thread`

```
class interrupt_flag {
public:
 void set();
 bool is_set() const;
};

thread_local interrupt_flag this_thread_interrupt_flag; ← (1)

class interruptible_thread {
 std::thread internal_thread;
 interrupt_flag* flag;

public:
 template<typename FunctionType>
 interruptible_thread(FunctionType f) {
  std::promise<interrupt_flag*> p; ← (2)
  internal_thread = std::thread([f,&p] { ← (3)
   p.set_value(&this_thread_interrupt_flag);
   f(); ← (4)
  });
  flag = p.get_future().get(); ← (5)
 }

 void interrupt() {
  if (flag) {
   flag->set(); ← (6)
  }
 }
};
```

Переданная функция `f` обертывается лямбда-функцией **(3)**, которая хранит копию `f` и ссылку на локальный объект-обещание `p` **(2)**. Перед тем как вызывать переданную функцию **(4)**, лямбда-функция устанавливает в качестве значения обещания адрес переменной `this_thread_interrupt_flag` (объявленной с модификатором `thread_local` **(1)**) в новом потоке. Затем вызывающий поток дожидается готовности будущего результата, ассоциированного с обещанием, и сохраняет этот результат в переменной-члене `flag` **(5)**. Отметим, что лямбда-функция исполняется в новом потоке и хранит висячую ссылку на локальную переменную `p`, но ничего страшного в этом нет, так как конструктор `interruptible_thread` ждет, пока на `p` не останется ссылок в новом потоке, и только потом возвращает управление. Еще отметим, что эта реализация не обрабатывает присоединение или отсоединение потока. Мы сами должны позаботиться об очистке переменной `flag` в случае выхода или отсоединения потока, чтобы избежать появления висячего указателя.

Теперь написать функцию `interrupt()` несложно: имея указатель на флаг прерывания, мы знаем, какой поток прерывать, поэтому достаточно просто поднять этот флаг **(6)**. Что делать дальше, решает сам прерываемый поток. О том, как принимается это решение, мы и поговорим ниже.

9.2.2. Обнаружение факта прерывания потока

Итак, мы умеем устанавливать флаг прерывания, но толку от него чуть, если поток не проверяет, что его хотят прервать. В простейшем случае это можно сделать с помощью функции `interruption_point()`, которую можно вызывать в точке, где прерывание безопасно. Если флаг прерывания установлен, то эта функция возбуждает исключение `thread_interrupted`:

```
void interruption_point() {
 if (this_thread_interrupt_flag.is_set()) {
  throw thread_interrupted();
 }
}
```

Обращаться к этой функции можно там, где нам удобно:

```
void foo() {
 while (!done) {
  interruption_point();
  process_next_item();
 }
}
```

Такое решение работает, но оно не идеально. Лучше всего прерывать поток тогда, когда он блокирован в ожидании чего-либо, но именно в этот момент поток как раз и не работает, а, значит, не может вызвать `interruption_point()`! Нам требуется какой-то механизм прерываемого ожидания.

9.2.3. Прерывание ожидания условной переменной

Итак, мы можем обнаруживать прерывание в подходящих местах программы с помощью обращений к функции `interruption_point()`, но это ничем не помогает в случае, когда поток блокирован в ожидании какого-то события, например сигнала условной переменной. Нам необходима еще одна функция, `interruptible_wait()`, которую можно будет перегрузить для различных ожидаемых событий, и нужно придумать, как вообще прерывать ожидание. Я уже говорил, что среди прочего ожидать можно сигнала условной переменной, поэтому с нее и начнем. Что необходимо для того, чтобы можно было прервать поток, ожидающий условную переменную? Проще всего было бы известить условную переменную в момент установки флага и поставить точку прерывания сразу после ожидания. Но в этом случае придётся разбудить все потоки, ждущие эту условную переменную, хотя заинтересован в этом только прерываемый поток. Впрочем, потоки, ждущие условную переменную, в любом случае должны обрабатывать ложные пробуждения, а отличить посланный нами сигнал от любого другого они не могут, так что ничего страшного не случится. В структуре `interrupt_flag` нужно будет сохранить указатель на условную переменную, чтобы при вызове `set()` ей можно было послать сигнал. В следующем листинге показана возможная реализация функции `interruptible_wait()` для условных переменных.

Листинг 9.10. Неправильная реализация `interruptible_wait()` для `std::condition_variable`

```
void interruptible_wait(std::condition_variable& cv,
 std::unique_lock<std::mutex>& lk) {
```

```
 interruption_point();← (1)
 this_thread_interrupt_flag.set_condition_variable(cv);
 cv.wait(lk);          ← (2)
 this_thread_interrupt_flag.clear_condition_variable();← (3)
 interruption_point();
}
```

В предположении, что существуют функции, которые устанавливают и разрывают ассоциацию условной переменной с флагом прерывания, этот код выглядит просто и понятно. Он проверяет, не было ли прерывания, ассоциирует условную переменную с флагом `interrupt_flag` для текущего потока **(1)**, ждет условную переменную **(2)**, разрывает ассоциацию с условной переменной **(3)** и снова проверяет, не было ли прерывания. Если поток прерывается во время ожидания условной переменной, то прерывающий поток пошлёт этой переменной сигнал, что пробудит нас и даст возможность проверить факт. К сожалению, этот код не работает, в нем есть две проблемы. Первая довольно очевидна: функция `std::condition_variable::wait()` может возбуждать исключения, поэтому из `interruptible_wait()` возможен выход без разрыва ассоциации флага прерывания с условной переменной. Это легко исправляется с помощью структуры, которая разрывает ассоциацию в ее деструкторе.

Вторая, не столь очевидная, проблема связана с гонкой. Если поток прерывается после первого обращения к `interruption_point()`, но до обращения к `wait()`, то не имеет значения, ассоциирована условная переменная с флагом прерывания или нет, потому что *поток еще ничего не ждет и, следовательно, не может быть разбужен сигналом, посланным условной переменной*. Мы должны гарантировать, что потоку не может быть послан сигнал между последней проверкой прерывания и обращением к `wait()`. Если не залезать в код класса `std::condition_variable`, то сделать это можно только одним способом: использовать для защиты мьютекс, хранящийся в `lk`, который, следовательно, нужно передавать функции `set_condition_variable()`. К сожалению, при этом возникают новые проблемы: мы передаём ссылку на мьютекс, о времени жизни которого ничего не знаем, другому потоку (тому, который выполняет прерывание), чтобы тот его захватил (внутри `interrupt()`). Но может случиться, что этот поток уже удерживает данный мьютекс, и тогда возникнет взаимоблокировка. К тому же, появляется возможность доступа к уже уничтоженному мьютексу. В общем, это решение не годится. Но если мы не можем *надежно* прерывать ожидание условной переменной, то нечего было и затевать это дело — почти того же самого можно было бы добиться и без специальной функции `interruptible_wait()`. Так какие еще есть варианты? Можно, к примеру, задать таймаут ожидания; использовать вместо `wait()` функцию `wait_for()` с очень коротким таймаутом (скажем, 1 мс). Это ограничивает сверху время до момента, когда поток обнаружит прерывание (с учетом промежутка между тактами часов). Если поступить так, что ожидающий поток будет видеть больше ложных пробуждений из-за срабатывания таймера, но тут уж ничего не попишешь. Такая реализация показана в листинге ниже вместе с соответствующей реализацией `interrupt_flag`.

Листинг 9.11. Реализация `interruptible_wait()` для `std::condition_variable` с таймаутом

```
class interrupt_flag {
 std::atomic<bool> flag;
 std::condition_variable* thread_cond;
```

```
 std::mutex set_clear_mutex;

public:
 interrupt_flag(): thread_cond(0) {}

 void set() {
  flag.store(true, std::memory_order_relaxed);
  std::lock_guard<std::mutex> lk(set_clear_mutex);
  if (thread_cond) {
   thread_cond->notify_all();
  }
 }

 bool is_set() const {
  return flag.load(std::memory_order_relaxed);
 }

 void set_condition_variable(std::condition_variable& cv) {
  std::lock_guard<std::mutex> lk(set_clear_mutex);
  thread_cond = &cv;
 }

 void clear_condition_variable() {
  std::lock_guard<std::mutex> lk(set_clear_mutex);
  thread_cond = 0;
 }

 struct clear_cv_on_destruct {
  ~clear_cv_on_destruct() {
   this_thread_interrupt_flag.clear_condition_variable();
  }
 };
};

void interruptible_wait(std::condition_variable& cv,
 std::unique_lock<std::mutex>& lk) {
 interruption_point();
 this_thread_interrupt_flag.set_condition_variable(cv);
 interrupt_flag::clear_cv_on_destruct guard;
 interruption_point();
 cv.wait_for(lk, std::chrono::milliseconds(1));
 interruption_point();
}
```

Если мы ждем какой-то предикат, то таймаут продолжительностью 1 мс можно полностью скрыть внутри цикла проверки предиката:

```
template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
 std::unique_lock<std::mutex>& lk,
 Predicate pred) {
 interruption_point();
 this_thread_interrupt_flag.set_condition_variable(cv);
 interrupt_flag::clear_cv_on_destruct guard;
 while (!this_thread_interrupt_flag.is_set() && !pred()) {
  cv.wait_for(lk, std::chrono::milliseconds(1));
 }
```

```
  interruption_point();
 }
```

Правда, предикат при этом проверяется чаще, чем необходимо, но зато эту функцию легко использовать вместо простого вызова `wait()`. Легко реализовать и другие варианты функций с таймаутом, например: ждать в течение указанного времени или 1 мс в зависимости от того, что меньше.

Ну хорошо, с ожиданием `std::condition_variable` мы разобрались, а что сказать о `std::condition_variable_any`? Всё точно так же или можно сделать лучше?

9.2.4. Прерывание ожидания `std::condition_variable_any`

Класс `std::condition_variable_any` отличается от `std::condition_variable` тем, что работает с *любым* типом блокировки, а не только с `std::unique_lock<std::mutex>`. Как выясняется, это сильно упрощает дело, так что мы сможем добиться более впечатляющих результатов, чем получилось с `std::condition_variable`. Раз допустим *любой* тип блокировки, то можно написать и свой собственный класс, который захватывает (освобождает) как внутренний мьютекс `set_clear_mutex` в классе `interrupt_flag`, так и блокировку, переданную при вызове `wait()`. Соответствующий код приведён в листинге ниже.

Листинг 9.12. Реализация `interruptible_wait()` для `std::condition_variable_any`

```
class interrupt_flag {
 std::atomic<bool> flag;
 std::condition_variable* thread_cond;
 std::condition_variable_any* thread_cond_any;
 std::mutex set_clear_mutex;

public:
 interrupt_flag():
  thread_cond(0), thread_cond_any(0) {}

 void set() {
  flag.store(true, std::memory_order_relaxed);
  std::lock_guard<std::mutex> lk(set_clear_mutex);
  if (thread_cond) {
   thread_cond->notify_all();
  } else if (thread_cond_any) {
   thread_cond_any->notify_all();
  }
 }

 template<typename Lockable>
 void wait(std::condition_variable_any& cv, Lockable& lk) {
  struct custom_lock {
   interrupt_flag* self;
   Lockable& lk;

   custom_lock(interrupt_flag* self_,
    std::condition_variable_any& cond,
    Lockable& lk_): self(self_), lk(lk_) {
```

```
     self->set_clear_mutex.lock();  ← (1)
     self->thread_cond_any = &cond; ← (2)
    }

    void unlock() { ← (3)
     lk.unlock();
     self->set_clear_mutex.unlock();
    }

    void lock() {
     std::lock(self->set_clear_mutex, lk); ← (4)
    }

    ~custom_lock() {
     self->thread_cond_any = 0; ← (5)
     self->set_clear_mutex.unlock();
    }
   };

   custom_lock cl(this, cv, lk);
   interruption_point();
   cv.wait(cl);
   interruption_point();
  }

  // остальное, как и раньше
 };

 template<typename Lockable>
 void interruptible_wait(std::condition_variable_any& cv,
  Lockable& lk) {
  this_thread_interrupt_flag.wait(cv, lk);
 }
```

Наш класс блокировки должен захватить внутренний мьютекс `set_clear_mutex` на этапе конструирования **(1)** и затем записать в переменную `thread_cond_any` указатель на объект `std::condition_variable_any`, переданный конструктору **(2)**. Ссылка на объект `Lockable` сохраняется для последующего использования; он должен быть уже заблокирован. Теперь проверять, был ли поток прерван, можно, не опасаясь гонки. Если в этой точке флаг прерывания установлен, то это было сделано до захвата мьютекса `set_clear_mutex`. Когда условная переменная вызывает нашу функцию `unlock()` внутри `wait()`, мы разблокируем объект `Lockable` *и внутренний мьютекс* `set_clear_mutex` **(3)**. Это позволяет потокам, которые пытаются нас прервать, захватить `set_clear_mutex` и проверить указатель `thread_cond_any`, *когда мы уже находимся в* `wait()`, но не раньше. Это именно то, чего мы хотели (но не смогли) добиться в случае `std::condition_variable`. После того как `wait()` завершит ожидание (из-за получения сигнала или вследствие ложного пробуждения), она вызовет нашу функцию `lock()`, которая снова захватит внутренний мьютекс `set_clear_mutex` и заблокирует объект `Lockable` **(4)**. Теперь можно еще раз проверить, не было ли прерываний, пока мы находились в `wait()`, и только потом обнулить указатель `thread_cond_any` в деструкторе `custom_lock` **(5)**, где также освобождается `set_clear_mutex`.

9.2.5. Прерывание других блокирующих вызовов

Теперь с прерыванием ожидания условной переменной всё ясно, но как быть с другими блокирующими операциями ожидания: освобождения мьютекса, готовности будущего результата и т.д.? Вообще говоря, приходится прибегать к тому же трюку с таймаутом, который мы использовали для `std::condition_variable`, потому что, если не влезать в код мьютекса или будущего результата, то нет никакого другого способа прервать ожидание, кроме как обеспечить выполнение ожидаемого условия. Но в отличие от условных переменных, мы точно знаем, чего ждем, поэтому можем организовать цикл внутри функции `interruptible_wait()`.

Вот, например, как выглядит перегрузка `interruptible_wait()` для `std::future<>`:

```
template<typename T>
void interruptible_wait(std::future<T>& uf) {
 while (!this_thread_interrupt_flag.is_set()) {
  if (uf.wait_for(lk, std::chrono::milliseconds(1) ==
   std::future_status::ready)
  break;
 }
 interruption_point();
}
```

Здесь мы ждем, пока либо будет установлен флаг прерывания, либо готов будущий результат, но блокирующее ожидание будущего результата продолжается в течение 1 мс на каждой итерации цикла. Это означает, что в среднем запрос на прерывание будет обнаружен с задержкой 0,5 мс, в предположении, что разрешение часов достаточно высокое. Функция `wait_for` обычно ожидает в течение как минимум одного такта, поэтому если такт системных часов составляет 15 мс, то ждать придётся не одну, а 15 мс. Приемлемо это или нет, зависит от конкретных условий. Таймаут при необходимости можно уменьшить (если часы позволяют), но тогда поток будет чаще просыпаться для проверки флага, что увеличит накладные расходы на переключение задач.

На данный момент мы знаем, как можно обнаружить прерывание с помощью функций `interruption_point()` и `interruptible_wait()`, но что с этим потом делать?

9.2.6. Обработка прерываний

С точки зрения прерываемого потока, прерывание — это просто исключение типа `thread_interrupted`, которое, следовательно, можно обработать, как любое другое исключение. В частности, его можно перехватить в стандартном блоке `catch`:

```
try {
 do_something();
} catch (thread_interrupted&) {
 handle_interruption();
}
```

Таким образом, прерывание можно перехватить, каким-то образом обработать и потом спокойно продолжить работу. Если мы так поступим, а потом другой поток еще раз вызовет `interrupt()`, то поток будет снова прерван, когда в очередной раз вызовет функцию `interruption_point()`. Возможно, это и неплохо, если вы выполняете последовательность независимых задач; текущая задача будет прервана, а поток благополучно перейдёт к

следующей задаче в списке.

Поскольку `thread_interrupted` — исключение, то при вызове кода, который может быть прерван, следует принимать все обычные для исключений меры предосторожности, чтобы не было утечки ресурсов, и структуры данных оставались согласованными. Часто желательно завершать поток в случае прерывания, так чтобы исключение можно было просто передать вызывающей функции. Но если позволить исключению выйти за пределы функции потока, переданной конструктору `std::thread`, то будет вызвана функция `std::terminate()`, что приведёт к завершению всей программы. Чтобы не помещать обработчик `catch(thread_interrupted)` в каждую функцию, которая передаётся `interruptible_thread`, можно включить блок `catch` в обертку, служащую для инициализации `interrupt_flag`. Тогда распространять необработанное исключение будет безопасно, так как завершится лишь отдельный поток. Инициализация потока в конструкторе `interruptible_thread` при таком подходе выглядит следующим образом:

```
internal_thread = std::thread([f, &p] {
 p.set_value(&this_thread_interrupt_flag);
  try {
  f();
 } catch(thread_interrupted const&) {}
});
```

А теперь рассмотрим конкретный пример, когда прерывание оказывается полезно.

9.2.7. Прерывание фоновых потоков при выходе из приложения

Представьте себе приложение для поиска в файловой системе настольного ПК. Оно должно не только взаимодействовать с пользователем, но и следить за состоянием файловой системы, обнаруживать изменения и обновлять свой индекс. Обычно такие операции поручаются фоновому потоку, чтобы пользовательский интерфейс мог реагировать на действия пользователя. Фоновый поток должен работать на протяжении всего времени жизни приложения; он запускается на этапе инициализации и трудится, пока приложение не завершится. Обычно это происходит при останове операционной системы, так как приложение должно постоянно поддерживать индекс в актуальном состоянии. Как бы то ни было, когда приложение завершается, надо аккуратно остановить и фоновые потоки, например, прервав их.

В следующем листинге показана возможная реализация управления потоками в такой программе.

Листинг 9.13. Фоновый мониторинг файловой системы

```
std::mutex config_mutex;
std::vector<interruptible_thread> background_threads;

void background_thread(int disk_id) {
 while (true) {
  interruption_point(); ← (1)
  fs_change fsc = get_fs_changes(disk_id); ← (2)
  if (fsc.has_changes()) {
   update_index(fsc); ← (3)
  }
```

```
  }
}

void start_background_processing() {
 background_threads.push_back(
  interruptible_thread(background_thread, disk_1));
 background_threads.push_back(
  interruptible_thread(background_thread, disk_2));
}

int main() {
 start_background_processing(); ← (4)
 process_gui_until_exit(); ← (5)
 std::unique_lock<std::mutex> lk(config_mutex);
 for (unsigned i = 0; i < background_threads.size(); ++i) {
  background_threads[i].interrupt(); ← (6)
 }
 for (unsigned i = 0; i < background_threads.size(); ++i) {
  background_threads[i].join(); ← (7)
 }
}
```

В самом начале запускаются фоновые потоки **(4)**. Затем главный поток продолжает обслуживать пользовательский интерфейс **(5)**. Когда пользователь хочет выйти из приложения, фоновые потоки прерываются **(6)**, после чего главный поток ждет их завершения **(7)**, и только потом выходит сам. Каждый фоновый поток исполняет цикл, в котором следит за изменениями на диске **(2)** и обновляет индекс **(3)**. На каждой итерации цикла поток проверяет, не прервали ли его, вызывая функцию `interruption_point()` **(1)**.

Почему мы прерываем все потоки до того, как начинать ждать их завершения? Почем нельзя прервать один поток, дождаться его, потом прервать следующий и так далее? Все из-за *параллелизма*. Поток не завершается сразу после прерывания, так как должен добраться до очередной точки прерывания, а затем, перед выходом, выполнить все деструкторы и код обработки исключений. Если главный поток будет присоединять прерванные потоки сразу после прерывания, то ему придётся ждать, *хотя в это время он мог бы делать полезную работу* — прерывать другие потоки. Поэтому мы поступаем по-другому — начинаем ждать только тогда, когда больше никакой работы не осталось (все потоки уже прерваны). Заодно это позволяет прерываемым потокам обрабатывать прерывания параллельно, так что общее время завершения, возможно, уменьшится.

Описанный механизм прерывания легко развить, добавив дополнительные прерываемые вызовы или запретив прерывания на определенном участке кода, но это я оставляю читателю в качестве упражнения.

9.3. Резюме

В этой главе мы рассмотрели «продвинутые» способы управления потоками: пулы и прерывание потоков. Мы видели, как использование локальных очередей работ и занимания работ может снизить накладные расходы на синхронизацию и потенциально увеличить пропускную способность пула потоков. Кроме того, мы поняли, что выполнение других выбранных из очереди задач во время ожидания завершения подзадачи устраняет возможность взаимоблокировки.

Мы также познакомились с различными способами прерывания одного потока другим, в частности, с точками прерывания и функциями, которые допускают прерывание во время блокирующего ожидания.

Глава 10.
Тестирование и отладка многопоточных приложений

В этой главе:

- Ошибки, связанные с параллелизмом.
- Поиск ошибок путем тестирования и анализа кода коллегами.
- Разработка тестов для многопоточных приложений.
- Тестирование производительности многопоточных приложений.

До сих пор мы занимались главным образом написанием параллельного кода — описанием имеющихся средств и порядка пользования ими и изучением общей структуры кода. Но у разработки ПО есть не менее важная сторона, о которой я еще не упоминал: тестирование и отладка. Если вы надеетесь найти в этой главе простой рецепт тестирования параллельного кода, то будете жестоко разочарованы. Тестировать и отлаживать параллельные программы *трудно*. Но я все же расскажу о некоторых приёмах, облегчающих эту задачу, а также сформулирую вопросы, над которыми стоит задуматься.

Тестирование и отладка — две стороны одной медали. Вы прогоняете тесты, чтобы найти в программе ошибки, и отлаживаетесь, чтобы эти ошибки устранить. Если повезёт, то придётся устранять только ошибки, найденные вашими собственными тестами, а не конечными пользователями. Но прежде чем приступать непосредственно к вопросам тестирования и отладки, важно понять, какие вообще могут возникать проблемы.

10.1. Типы ошибок, связанных с параллелизмом

В параллельной программе могут быть любые ошибки, в этом отношении она не представляет собой ничего особенного. Однако есть ряд ошибок, непосредственно связанных с параллелизмом, и вот они-то будут нам особенно интересны. Как правило, эти ошибки попадают в одну из двух категорий:

• нежелательное блокирование;

• состояния гонки.

Эти категории очень общие, поэтому давайте немного уточним их. Сначала рассмотрим нежелательное блокирование.

10.1.1. Нежелательное блокирование

Что я понимаю под нежелательным блокированием? Прежде всего, поток считается *заблокированным*, если он не может продолжать выполнение, так как чего-то ждет. Это что-то может быть мьютексом, условной переменной, будущим результатом или завершением ввода/вывода. Это естественный, но не всегда приветствуемый аспект многопоточного кода, потому мы и говорим о проблеме нежелательного блокирования. Тогда возникает следующий вопрос: почему блокирование нежелательно? Обычно потому, что какой-то другой поток ждет результатов от заблокированного потока, чтобы выполнить некоторую операцию. И, значит, этот поток также оказывается заблокированным. На эту тему есть различные вариации.

• *Взаимоблокировка* — в главе 3 мы видели, что взаимоблокировка возникает, когда один поток ждет другого, а тот, в свою очередь, ждет первого. Если потоки взаимно блокируют друг друга, то порученные им задачи вообще никогда не будут выполнены. Наиболее наглядно это проявляется, когда один из таких потоков отвечает за пользовательский интерфейс; в этом случае интерфейс просто перестаёт реагировать на действия пользователя. В других случаях интерфейс реагирует, но какая-то задача не может завершиться, например, поиск не возвращает результатов или документ не печатается.

• *Активная блокировка* — похожа на взаимоблокировку в том смысле, что один поток ждет другого, а тот ждет первого. Но ключевое отличие заключается в том, что здесь мы имеем не блокирующее ожидание, а цикл активной проверки, например спинлок. В серьезных случаях симптомы выглядят так же, как при взаимоблокировке (приложение не может продолжить работу), только программа потребляет много процессорного времени, потому что потоки блокируют друг друга, продолжая работать. В менее серьезных случаях активная блокировка рано или поздно «рассасывается» из-за вероятностной природы планирования, но заблокированная задача испытывает ощутимые задержки, характеризуемые высоким потреблением процессорного времени.

• *Блокировка в ожидании завершения ввода/вывода или поступления данных из внешнего источника* — если поток блокируется в ожидании данных из внешнего источника, то он не может продолжать работу, даже если данные так никогда и не поступят. Поэтому крайне нежелательно, когда такая блокировка происходит в потоке, от работы которого зависят другие потоки.

Вот вкратце описание нежелательного блокирования. А как насчет состояний гонки?

10.1.2. Состояния гонки

Состояния гонки — одна из самых распространенных причин ошибок в многопоточных программах, часто взаимоблокировки и активные блокировки — лишь проявления гонки. Не все состояния гонки проблематичны — гонка возникает всякий раз, как поведение зависит от порядка планирования операций в различных потоках. Многие состояния гонки совершенно безобидны; например, безразлично, какой поток заберет очередную задачу из очереди. Однако же целый ряд связанных с параллелизмом ошибок обусловлен именно гонкой. В частности, гонки нередко приводят к следующим проблемам.

• *Гонка за данными* — это особый тип гонки, который приводит к неопределенному поведению из-за несинхронизированного одновременного доступа к разделяемой ячейке памяти. С этим видом гонок мы познакомились в главе 5 при изучении модели памяти в C++. Обычно гонка за данными возникает вследствие неправильного использования атомарных операций для синхронизации потоков или в результате доступа к разделяемым данным, не защищенного подходящим мьютексом.

• *Нарушение инвариантов* — такие гонки могут проявляться в форме висячих указателей (другой поток уже удалил данные, к которым мы пытаемся обратиться), случайного повреждения памяти (из-за того, что поток читает данные, оказавшиеся несогласованными в результате частичного обновления) и двойного освобождения (например, два потока извлекают из очереди одно и то же значение, и потом оба удаляют ассоциированные с ним данные). Нарушение инварианта может быть связано как с несоблюдением временных соотношений, так и с неправильными значениями. Если требуется, чтобы операции в разных потоках выполнялись в определенном порядке, то некорректная синхронизация может стать причиной гонки, из-за которой требуемый порядок иногда нарушается.

• *Проблемы со временем жизни* — такого рода проблемы можно было бы отнести к нарушению инвариантов, но на самом деле это отдельная категория. Основная проблема в том, что поток живет дольше, чем данные, к которым он обращается, поэтому может попытаться получить доступ к уже удаленным или разрушенным иным способом данным. Не исключено также, что когда-то отведенная под эти данные память уже занята другим объектом. Обычно такие ошибки возникают, когда поток хранит ссылки на локальные переменные, которые вышли из области видимости до завершения функции потока, но это не единственный сценарий. Если время жизни потока и данных, которыми он оперирует, никак не связано, то всегда существует возможность, что данные будут уничтожены до завершения потока, и у функции потока просто «выбьют почву из-под ног». Если вы вручную вызываете `join()`, чтобы дождаться завершения потока, то следите за тем, чтобы вызов `join()` не пропускался из-за исключения. Это простейшая мера безопасности относительно исключений, применяемая к потокам.

Больше всего неприятностей приносят именно проблематичные гонки. Если возникает взаимоблокировка или активная блокировка, то кажется, что приложение зависло — оно либо вообще перестаёт отвечать, либо тратит на выполнение задачи несоразмерно много времени. Зачастую можно подключить к работающему процессу отладчик и понять, какие потоки участвуют в блокировке и какие объекты синхронизации они не поделили. В случае гонок за данными, нарушенных инвариантов или проблем со временем жизни видимые симптомы ошибки (например, произвольные «падения» или неправильный вывод) могут

проявляться где угодно — программа может затереть память, используемую в другой части системы, к которой обращений не будет еще очень долго. Таким образом, ошибка проявляется в коде, совершенно не относящемся к месту ее возникновения, и, возможно, гораздо позже в процессе выполнения программы. Это проклятие всех систем с разделяемой памятью — как бы вы ни пытались ограничить количество данных, доступных потоку, какие бы меры ни принимали для правильной синхронизации, любой поток в состоянии затереть данные, используемые любым другим потоком в том же приложении.

Теперь, когда мы вкратце описали, какие проблемы нас интересуют, посмотрим, как находить проблемные места в коде и исправлять их.

10.2. Методы поиска ошибок, связанных с параллелизмом

В предыдущем разделе мы познакомились с типами ошибок, обусловленных параллелизмом, и тем, как они могут проявляться. Памятуя об этом, мы можем изучить подозрительные участки кода и попытаться понять, есть там ошибки или нет.

Пожалуй, самый прямой и очевидный путь — *посмотреть на код «глазками»*. Несмотря на кажущуюся очевидность, на практике сделать это тщательно весьма трудно. Читая только что написанный вами же код, очень легко увидеть то, что вы собирались написать, а не то, что написано на самом деле. Аналогично, при анализе кода, написанного другим человеком, возникает соблазн быстренько проглядеть текст, проверить его на предмет соблюдения местных стандартов кодирования и отметить проблемы, бросающиеся в глаза. А надо бы потратить время, пройтись по коду мелким гребнем, задуматься над местами, связанными с параллелизмом — да и не связанными тоже (а почему бы и нет, в конце концов, ошибка — она и в Африке ошибка). Чуть ниже мы поговорим о том, что конкретно должно стать предметом таких размышлений.

Но даже после тщательного анализа кода какие-то ошибки могут остаться незамеченными, и в любом случае хотелось бы подтвердить, что код действительно работает — хотя бы ради собственного спокойствия. Поэтому от умозрительного анализа мы перейдём к описанию нескольких способов тестирования многопоточной программы.

10.2.1. Анализ кода на предмет выявления потенциальных ошибок

Я уже упоминал, что анализ многопоточного кода на предмет выявления ошибок, связанных с параллелизмом, надо проводить тщательно, прочёсывая код мелким гребнем. Если возможно, попросите заняться этим кого-нибудь другого. Поскольку этот человек не писал код, то ему придётся думать, как он работает, и это поможет обнаружить скрытые ошибки. Важно, чтобы у рецензента было достаточно времени — нужно не проглядеть код мельком, за пару минут, а тщательно и усидчиво проанализировать. Большинство ошибок, связанных с параллелизмом, поверхностный читатель не увидит — обычно для их проявления нужно редкое сочетание временных соотношений.

Коллега, если вам удастся упросить его проанализировать ваш код, будет смотреть на него свежим взглядом и под другим углом зрения, чем вы сами. Поэтому он может заметить вещи, ускользнувшие от вашего внимания. Если коллег нет, попросите приятеля, можете даже выложить свой код в Интернет (не оскорбляя чувств юристов компании). Но даже если не найдется никого, кто проанализирует ваш код, или если рецензент ничего не обнаружит, все равно не отчаивайтесь — на этом свет клином не сошелся. Для начала имеет смысл на время отложить код — поработать над другой частью программы, книжку почитать, погулять. Во время перерыва вы будете подсознательно обдумывать задачу, заняв сознание чем-то другим. А когда вернетесь к коду, он будет казаться не таким знакомым, и, возможно, вам самому удастся взглянуть на него другими глазами.

Вместо того чтобы обращаться за помощью, можете проанализировать свой код самостоятельно. Например, полезно попытаться *во всех деталях* объяснить кому-нибудь, как он работает. Это даже необязательно должен быть человек — вполне подойдёт плюшевый медвежонок или надувной цыплёнок. Лично мне очень помогает написание подробных

заметок. По ходу объяснения думайте над каждой строкой, рассказывайте, что может произойти, к каким данным происходят обращения и т.д. Задавайте себе вопросы о программе и объясняйте свои ответы. Мне кажется, что это очень действенная методика — задавая себе вопросы и тщательно продумывая ответы, зачастую удается выявить проблемы. Причем задавать вопросы полезно при анализе *любого* кода, а не только своего собственного.

Над какими вопросами следует задуматься при анализе многопоточного кода

Я уже говорил, что рецензенту (неважно, является он автором программы или нет) полезно задавать конкретные вопросы по поводу анализируемой программы. Они позволяют сосредоточиться на деталях кода и выявить потенциальные проблемы. Лично я люблю задавать вопросы, перечисленные ниже, хотя это, конечно, далеко не исчерпывающий список. Вам, возможно, помогут лучше сконцентрироваться совсем другие вопросы.

- Какие данные нужно защищать от одновременного доступа?
- Как вы обеспечиваете защиту этих данных?
- В каком участке программы могут в этот момент находиться другие потоки?
- Какие мьютексы удерживает данный поток?
- Какие мьютексы могут удерживать другие потоки?
- Существуют ли ограничения на порядок выполнения операций в этом и каком-либо другом потоке? Как гарантируется соблюдение этих ограничений?
- Верно ли, что данные, загруженные этим потоком, все еще действительны? Не могло ли случиться, что их изменили другие потоки?
- Если предположить, что другой поток может изменить данные, то к чему это приведёт и как гарантировать, что этого никогда не случится?

Последний вопрос — мой любимый, потому что заставляет думать о взаимосвязях между потоками. Допустив, что в некоторой строке имеется ошибка, вы дальше перевоплощаетесь в сыщика, которому нужно раскрыть преступление. Чтобы убедить себя в отсутствии ошибки, требуется рассмотреть все граничные случаи, приняв во внимание любой возможный порядок операций. Это особенно полезно, если данные в разные моменты времени защищаются разными мьютексами, как, например, обстояло дело в потокобезопасной очереди из главы 6, где мы завели разные мьютексы для головы и хвоста очереди. Чтобы гарантировать безопасность доступа в момент, когда захвачен один мьютекс, нужна уверенность в том, что поток, удерживающий *другой* мьютекс, не будет пытаться получить доступ к тому же элементу. Очевидно, что общедоступные данные, а также данные, на которые программа может получить ссылку или указатель, нужно анализировать особенно пристрастно.

Предпоследний вопрос из списка также важен, потому что касается очень распространенной ошибки: если вы освобождаете, а затем снова захватываете мьютекс, то должны предполагать, что другие потоки могли изменить разделяемые данные. На первый взгляд, очевидно, но если операции с мьютексами не видны — например, потому что скрыты внутри какого-то объекта, — то вы неосознанно допускаете именно эту ошибку В главе 6 мы видели, как это может привести к гонке и ошибкам, когда функции в потокобезопасной структуре данных слишком детализированы. Если для стека, не безопасного относительно потоков, наличие отдельных операций `top()` и `pop()` оправдано, то для стека, к которому

могут одновременно обращаться несколько потоков, это уже не так, потому что между этими двумя вызовами внутренний мьютекс не захвачен, и, значит, какой-то другой поток может модифицировать стек. В главе 6 мы видели, что для решения этой проблемы нужно объединить обе операции в одну — выполняемую под защитой одной и той же блокировки мьютекса. Тем самым опасность гонки устраняется.

Итак, вы проанализировали код (или это сделал кто-то другой). Вы уверены, что в нем нет ошибок. Но критерием истины, как известно, является практика — как можно протестировать код, подтвердив или опровергнув вашу веру в отсутствие ошибок?

10.2.2. Поиск связанных с параллелизмом ошибок путем тестирования

Тестирование однопоточных приложений — процедура сравнительно простая, хотя, возможно, и длительная. Теоретически можно идентифицировать все возможные наборы входных данных (или, но крайней мере, все интересные случаи) и подать их на вход приложения. Если поведение и выходные данные программы совпадают с ожидаемыми, значит, для соответствующего набора входных данных программа работает корректно. Тестировать такие ситуации, как переполнение диска, сложнее, но идея та же самая — подготовить начальные условия и прогнать приложение.

Тестирование многопоточного кода на порядок сложнее, потому что точный порядок выполнения потоков не детерминирован и может изменяться от запуска к запуску. Следовательно, если в коде притаилось какое-то состояние гонки, то даже на одном и том же наборе входных данных программа иногда может работать правильно, а иногда давать ошибку. Наличие потенциальной гонки не означает, что программа будет выдавать ошибку *всегда*, утверждается лишь, что *иногда* она *может* сбоить.

Ввиду трудностей воспроизведения ошибки, внутренне присущих многопоточным программам, вы должны проектировать тесты очень тщательно. Желательно, чтобы каждый тест проверял как можно меньший участок кода, тогда при возникновении ошибки ее будет проще изолировать. Конкретно, проверять правильность работы операций помещения и извлечения элементов в параллельной очереди лучше напрямую, а не путем тестирования всего куска кода, в котором эта очередь используется. Очень помогает еще на этапе проектирования кода думать о том, как он будет тестироваться. См. по этому поводу раздел о тестопригодности ниже в этой главе.

Имеет также смысл устранять из тестов параллелизм, так как это позволяет убедиться, что проблема не связана с параллельным доступом. Если проблема проявляется даже при однопоточной работе, то это самая обычная ошибка, не имеющая отношения к параллелизму. Это особенно важно, когда вы пытаетесь установить причины ошибки, произошедшей «в поле», а не в тестовом окружении. Если ошибка возникает в многопоточной части программы, то это еще не значит, что она как-то связана с параллелизмом. При использовании пулов потоков обычно имеется конфигурационный параметр, определяющий число рабочих потоков. Если вы управляете потоками вручную, то нужно будет модифицировать код, так чтобы в тесте работал только один поток. Как бы то ни было, если удастся воспроизвести ошибку в однопоточном варианте программы, то параллелизм можно исключить из числа возможных причин. С другой стороны, если проблема исчезает при работе в *одноядерной* системе (даже при наличии нескольких одновременно работающих потоков), но появляется в *многоядерной* или *многопроцессорной*, то имеет место состояние

гонки и, возможно, ошибка, связанная с синхронизацией или упорядочением доступа к памяти.

При тестировании параллельного кода важна не только структура самого кода, но и структура теста и тестовой среды. Все в том же примере параллельной очереди необходимо проверить следующие случаи.

• Один поток вызывает `push()` или `pop()` для проверки работоспособности очереди на самом простом уровне.

• Один поток вызывает `push()` для пустой очереди, а второй в это время вызывает `pop()`.

• Несколько потоков вызывают `push()` для пустой очереди.

• Несколько потоков вызывают `push()` для заполненной очереди.

• Несколько потоков вызывают `pop()` для пустой очереди.

• Несколько потоков вызывают `pop()` для заполненной очереди.

• Несколько потоков вызывают `pop()` для частично заполненной очереди, в которой недостаточно элементов для удовлетворения всех потоков.

• Несколько потоков вызывают `push()`, а один вызывает `pop()` для пустой очереди.

• Несколько потоков вызывают `push()`, а один вызывает `pop()` для заполненной очереди.

• Несколько потоков вызывают `push()` и несколько потоков вызывают `pop()` для пустой очереди.

• Несколько потоков вызывают `push()` и несколько потоков вызывают `pop()` для заполненной очереди.

Проверив все эти и другие случаи, вы затем должны учесть дополнительные параметры тестовой среды.

• Что понимается под «несколькими потоками» в каждом случае (3, 4, 1024)?

• Достаточно ли в системе процессорных ядер, чтобы каждый поток работал на отдельном ядре?

• Какова архитектура процессора, на котором будет прогоняться тест?

• Как обеспечить подходящее планирование для циклов «while» в тестах?

В зависимости от ситуации может быть необходимо принять во внимание и другие факторы. Из четырех приведённых выше аспектов тестовой среды первый и последний относятся к структуре самого теста (и рассматриваются в разделе 10.2.5), а оставшиеся два — к физической тестовой системе. Сколько потоков использовать, определяется конкретной программой, но способов структурировать тесты для получения нужного планирования потоков существует несколько. Прежде чем рассматривать их, поговорим о том, как следует проектировать код, чтобы его было легко тестировать.

10.2.3. Проектирование с учетом тестопригодности

Тестировать многопоточный код трудно, поэтому вы должны сделать все возможное, чтобы облегчить эту задачу. И едва ли не самое важное — проектировать код с учетом тестопригодности. Для однопоточных программ на эту тему написано немало, и многие рекомендации применимы и к многопоточному случаю. Вообще говоря, код проще тестировать, если он написан с соблюдением следующих принципов.

• Обязанности всех функций и классов четко очерчены.

• Каждая функция короткая и решает ровно одну задачу.

• Тесты способны полностью контролировать окружение тестируемого кода.

• Код, выполняющий конкретную тестируемую операцию, находится приблизительно в одном месте, а не разбросан но всей системе.

• Автор сначала думал о том, как будет тестировать код, а только потом приступал к его написанию.

Все это остается в силе и для многопоточного кода. Я бы даже сказал, что думать о тестопригодности многопоточной программы даже важнее, чем однопоточной, поскольку тестировать ее гораздо труднее. Очень важен последний пункт: даже если вы не считаете нужным писать тесты раньше кода, все равно стоит заранее подумать о том, как вы будете тестировать — какие входные данные использовать, при каких условиях могут проявиться проблемы, какие способы обращения к коду могут выявить потенциальные проблемы и т.д.

Один из лучших способов проектирования параллельного кода, пригодного для тестирования, — устранить параллелизм. Если программу удается разбить на части, которые отвечают за взаимодействие потоков, и части, которые оперируют данными внутри одного потока, то проблема сильно упрощается. Части, оперирующие данными, к которым может обращаться только один поток, можно тестировать, применяя хорошо известные методы. А трудный для тестирования параллельный код, который отвечает за взаимодействие потоков и должен гарантировать, что в каждый момент времени только один поток обращается к конкретному блоку данных, теперь оказывается гораздо короче и обозримее.

Например, приложение, спроектированное в виде многопоточного конечного автомата, можно разбить на несколько частей. Логику управления состояниями в каждом потоке, отвечающую за правильность переходов и операций для любого возможного набора входных событий, можно тестировать независимо, применяя стандартные методы для однопоточного случая. При этом входные событие, которые реально должны были бы поступать от других потоков, будет поставлять тестовая среда. После этого независимо можно протестировать основной код конечного автомата и код маршрутизации сообщений и убедиться, что события доставляются нужному потоку в правильном порядке; при этом специально для тестов будет написана простая логика работы в каждом состоянии.

Или, если получится разбить программу на несколько блоков вида *читать разделяемые данные / преобразовать данные / обновить разделяемые данные*, то части *преобразовать данные* можно будет протестировать с помощью стандартных методов, поскольку это обычный однопоточный код. И трудная задача тестирования многопоточных преобразований будет сведена к тестированию чтения и обновления разделяемых данных, что гораздо проще.

Нужно обращать особое внимание на библиотечные вызовы, в которых могут использоваться внутренние переменные для хранения состояния, поскольку эти переменные становятся разделяемыми, если к библиотечным функциям обращаются сразу несколько потоков. Проблема в том, что сразу не очевидно, что код обращается к разделяемым данным. Впрочем, со временем вы запоминаете такие функции, потому что они, словно болячка, постоянно напоминают о себе. Тогда можно либо добавить подходящую защиту и синхронизацию, либо пользоваться другими функциями, безопасными для доступа из нескольких потоков.

Проектирование многопоточного кода с учетом тестопригодности не сводится к структурированию программы с целью уменьшить объем кода, имеющего дело с параллелизмом, и внимательному обращению с библиотечными функциями. Полезно также помнить о вопросах, которые вы задаете себе при анализе кода (см. раздел 10.2.1). Они, правда, не имеют прямого отношения к тестированию и тестопригодности, но, постоянно

держа в уме вопросы тестирования, вы будете принимать более правильные проектные решения, которые затем это самое тестирование облегчат.

Итак, мы поговорили о том, как проектировать код с учетом тестопригодности и но возможности отделять «параллельные» части (например, потокобезопасные контейнеры и логику событий конечного автомата) от «однопоточных» (которые все же могут взаимодействовать с другими потоками при посредстве параллельных частей). А теперь рассмотрим некоторые приёмы тестирования параллельного кода.

10.2.4. Приемы тестирования многопоточного кода

Вы уже продумали сценарий, который собираетесь тестировать, и написали код, подвергающий тестируемые функции испытаниям. Как обеспечить произвольный потенциально проблематичный порядок планирования, чтобы ошибки вылезли на свет? Есть несколько способов, начиная с тестирования грубой силой, или нагрузочного тестирования.

Тестирование грубой силой

Идея тестирования грубой силой заключается в том, чтобы подвергать программу большой нагрузке и наблюдать за появлением ошибок. Обычно это означает, что код прогоняется многократно и, возможно, с большим количеством потоков. Если некоторая ошибка возникает только при определенном порядке планирования потоков, то чем дольше вы будете гонять код, тем больше будет вероятность ее проявления. Если тест прогоняется только один раз и при этом проходит, то появляется некоторая уверенность в его правильности. Если тест проходит десять раз подряд, то эта уверенность возрастает. Ну а если и после миллиарда прогонов ошибок не было, то доверие к коду становится еще сильнее.

Степень доверия к результатам, конечно, зависит от объема кода, проверяемого каждым тестом. Если тесты очень детальные, как, например, рассмотренные выше для потокобезопасной очереди, то такое тестирование грубой силой порождает высокую степень доверия к программе. С другой стороны, если тестированию подвергаются крупные участки кода, то количество возможных вариантов порядка планирования настолько велико, что и после миллиарда прогонов теста уверенность в правильности кода слабенькая.

Недостаток метода грубой силы в том, что он может вселять ложную уверенность. Если тест написан таким образом, что проблематичные условия просто не могут возникнуть, то прогонять его можно сколь угодно долго, и всякий раз он будет проходить, хотя стоит условиям чуть-чуть измениться, как сразу возникнет ошибка. Наихудший вариант такого развития событий возникает, когда система, на которой производится тестирование, настроена так, что проблематичные условия в принципе невозможны. Это бывает, когда производственная система отличается от тестовой, и конкретное сочетание оборудования и операционной системы не дает материализоваться условиям, при которых возникает ошибка.

Классический пример — тестирование многопоточного приложения на однопроцессорной машине. Поскольку все потоки исполняются единственным процессором, работа программы автоматически сериализуется, а разнообразные состояния гонки и проблемы перебрасывания кэша, которые могли бы наблюдаться в

многопроцессорной системе, вообще невозможны. Но это еще не все — процессоры с разной архитектурой предоставляют различные средства синхронизации и упорядочения доступа к памяти. Например, в процессорах x86 и x86-64 атомарные операции загрузки одинаковы вне зависимости от того, помечены они признаком `memory_order_relaxed` или `memory_order_seq_cst` (см. раздел 5.3.3). Это означает, что код, написанный в предположении ослабленного упорядочения, может работать на машинах с архитектурой x86, но откажет на машине с системой команд, допускающей более точное управление порядком доступа к памяти, например, SPARC.

Если приложение должно быть переносимо на разные архитектуры, то и тестировать его следует на машинах, представляющих каждую возможную архитектуру. Поэтому я и включил архитектуру процессора в список факторов, которые нужно учитывать при тестировании (см. раздел 10.2.2).

Применяя тестирование грубой силой, важно всеми силами избегать условий, способных породить ложную уверенность. Для этого необходимо тщательно планировать тесты, уделяя внимание не только тому, какие участки кода подвергать тестированию, но также проектированию стенда (test harness) и выбору тестовой среды. Вы должны постараться протестировать как можно больше путей выполнения кода и взаимодействий между потоками. При этом еще необходимо знать, *какие именно* варианты протестированы, а *какие остались не протестированными*.

Хотя метод грубой силы дает некоторую уверенность в правильности кода, гарантировать обнаружение всех ошибок он не может. Однако существует методика, которая *гарантированно* находит проблемы, если только у вас есть время применить ее к своему коду и подходящее программное обеспечение. Я называю ее *комбинаторным имитационным тестированием*.

Комбинаторное имитационное тестирование

Название не вполне внятное, поэтому объясню, что я имею в виду. Идея в том, чтобы прогонять код под управлением специальной программы, которая *имитирует* реальную среду. Вам, наверное, доводилось слышать о программах, которые запускают несколько виртуальных машин на одном физическом компьютере, причём характеристики каждой виртуальной машины и оборудования эмулируются программным супервизором. Здесь всё похоже, только вместо эмулирования системы имитационное ПО записывает последовательности операций доступа к данным, захвата блокировок и атомарных операций в каждом потоке. Затем она применяет правила модели памяти, определенные в C++, чтобы повторить прогон при любой допустимой *комбинации* операций и таким образом выявить гонки и взаимоблокировки.

Хотя такое исчерпывающее тестирование всех комбинаций гарантированно обнаружит все проблемы, на поиск которых система рассчитана, но для любой программы, кроме самых тривиальных, оно займет чудовищно много времени, потому что количество комбинаций экспоненциально увеличивается с ростом числа потоков и операций, выполняемых в каждом потоке. Лучше применять эту методику к детальным тестам отдельных частей кода, а не ко всему приложению. Очевидный недостаток заключается том, что необходимо специальное имитационное ПО, способное обработать встречающиеся в вашей программе операции.

Таким образом, мы располагаем методикой, подразумевающей многократный прогон тестов при обычных условиях, которая, однако, может пропускать некоторые ошибки, и методикой, предполагающий запуск в специально созданных условиях, которая найдет ошибки с гораздо большей вероятностью. А есть ли еще какие-нибудь варианты?

Третий способ — воспользоваться библиотекой, которая сама обнаруживает проблемы, возникающие при прогоне тестов.

Обнаружение возникающих во время тестирования проблем с помощью специальной библиотеки

Этот вариант не обеспечивает исчерпывающего покрытия, которое дает комбинаторное имитационное тестирование, но все же позволяет выявить многие проблемы за счет специальных реализаций таких библиотечных примитивов синхронизации, как мьютексы, блокировки и условные переменные. Например, часто требуется, чтобы любой доступ к некоторым разделяемым данным, производился, когда захвачен определенный мьютекс. Если бы была возможность проверить, какие мьютексы захвачены в момент доступа к этим данным, то можно было бы сообщить об ошибке в случае, когда нужный мьютекс не захвачен. Пометив разделяемые данные определенным образом, мы смогли бы сообщить библиотеке, что проверять.

Такая реализация библиотеки могла бы также записывать последовательность захватов в случае, когда некоторый поток одновременно удерживает более одного мьютекса. Если другой поток попытается захватить те же мьютексы в другом порядке, то будет зарегистрирована *потенциальная* взаимоблокировка, даже если при реальном прогоне теста она не возникала.

Для тестирования многопоточного кода могла бы быть полезна и специальная библиотека другого рода, в которой реализации таких примитивов, как мьютексы и условные переменные, позволяют автору теста управлять тем, какой поток захватит блокировку, если ее ожидают несколько потоков, или какой из потоков, ожидающих условную переменную, будет разбужен вызовом `notify_one()`. Это дало бы возможность настраивать конкретные сценарии и проверять, что код работает в соответствии с ожиданиями.

Некоторые из описанных средств тестирования следовало бы включать в стандартную библиотеку C++, а другие можно было бы реализовать на основе стандартной библиотеке как часть тестового стенда.

Обсудив различные способы исполнения тестового кода, посмотрим, как можно структурировать код для достижения желаемого порядка планирования потоков.

10.2.5. Структурирование многопоточного тестового кода

В разделе 10.2.2 я говорил о том, что нужно придумать, как обеспечить надлежащий порядок планирования для циклов «while» в тестах. Сейчас самое время поговорить о возникающих здесь вопросах.

Основная проблема — организовать набор потоков таким образом, чтобы каждый исполнял выбранный фрагмент кода в указанный вами момент времени. В простейшем случае потоков всего два, но решение легко обобщается и на большее число. На первом этапе

нужно определиться, как устроен каждый тест:

• код общей настройки, исполняемый в самом начале;

• потоковый код настройки, исполняемый в каждом потоке;

• содержательный код, исполняемый в параллельно работающих потоках;

• код, исполняемый по завершении параллельного исполнения; может включать утверждения о состоянии программы.

Для определённости рассмотрим пример из списка в разделе 10.2.2: один поток вызывает `push()` для пустой очереди, а второй в это время вызывает `pop()`.

Код *общей* настройки очевиден: надо создать очередь. В потоке, исполняющем `pop()`, нет *потокового* кода настройки. Потоковый код настройки для потока, исполняющего `push()`, зависит от интерфейса очереди и типа сохраняемого в ней объекта. Если конструировать сохраняемый объект дорого или память для него должна выделяться из кучи, то лучше сделать это в потоковом коде настройки, чтобы не оказывать влияния на сам тест. С другой стороны, если в очереди хранятся всего лишь значения типа `int`, то мы ничего не выиграем от их конструирования в коде настройки. Собственно тестируемый код тоже прост — вызвать `push()` в одном потоке и `pop()` в другом. А вот как быть с кодом, «исполняемым по завершении»?

В данном случае всё зависит от того, что должна делать функция `pop()`. Если предполагается, что она блокирует поток до появления данных в очереди, то, очевидно, мы ожидаем, что будут возвращены данные, переданные функции `push()`, и что очередь в итоге окажется пустой. Если же `pop()` *не* блокирует поток и может вернуть управление, даже когда очередь пуста, то требуется проверить два возможных исхода: либо `pop()` вернула данные, переданные `push()`, и очередь пуста, либо `pop()` известила об отсутствии данных и в очереди есть один элемент. Истинно должно быть ровно одно утверждение; чего мы точно не хотим, так это ситуации, когда `pop()` говорит «нет данных», но очередь пуста, или когда `pop()` вернула значение, а очередь все равно *не* пуста. Для упрощения теста предположим, что функция `pop()` блокирующая. Тогда в завершающем коде должно быть утверждение вида «извлеченное значение совпадает с помещённым и очередь пуста».

Определившись со структурой кода, мы должны постараться, чтобы все работало в соответствии с планом. Один из путей - воспользоваться набором объектов `std::promise`, обозначающих, что все готово. Каждый поток устанавливает обещание, сообщая, что он готов, а затем ждет (копии) будущего результата `std::shared_future`, полученного из третьего объекта `Std::promise`; главный поток ждет обещаний от всех потоков, а затем запускает потоки, устанавливая `go`. Тем самым гарантируется, что каждый поток запущен и находится в точке, непосредственно предшествующей коду, который должен выполняться параллельно; весь потоковый код настройки должен завершиться до установки обещания `go`. Наконец, главный поток ждет завершения других потоков и проверяет получившееся состояние. Мы также должны принять во внимание исключения и гарантировать, что ни один поток не будет ждать сигнала `go`, который никогда не поступит. В листинге ниже приведён один из возможных способов структурирования этого теста.

Листинг 10.1. Пример теста, проверяющего параллельное выполнение функций очереди `push()` и `pop()`

```
    void test_concurrent_push_and_pop_on_empty_queue() {
     threadsafe_queue<int> q; ← (1)
```

```
 std::promise<void> go, push_ready, pop_ready;← (2)
 std::shared_future<void>
  ready(go.get_future()); ← (3)

 std: :future<void> push_done; ← (4)
 std::future<int> pop_done;

 try {
  push_done = std::async(std::launch::async, ← (5)
   [&q, ready, &push_ready]() {
    push_ready.set_value();
    ready.wait();
    q.push(42);
   }
  );
  pop_done = std::async(std::launch::async, ← (6)
   [&q, ready, &pop_ready]() {
    pop_ready.set_value();
    ready.wait();
    return q.pop(); ←(7)
   }
  );
  push_ready.get_future().wait(); ← (8)
  pop_ready.get_future().wait();
  go.set_value(); ← (9)

  push_done.get(); ← (10)
  assert(pop_done.get() == 42); ← (11)
  assert(q.empty());
 } catch (...) {
  go.set_value(); ← (12)
  throw;
 }
}
```

Структура кода в точности соответствует описанной выше. Сначала, в коде общей настройки, мы создаем пустую очередь **(1)**. Затем создаем все объекты-обещания для сигналов `ready` (готово) **(2)** и получаем `std::shared_future` для сигнала `go` **(3)**. После этого создаются будущие результаты, означающие, что потоки завершили исполнение **(4)**. Они должны быть созданы вне блока `try`, чтобы сигнал `go` можно было установить в случае исключения, не ожидая завершения потоков (что привело бы к взаимоблокировке — вещь, абсолютно недопустимая в тесте).

Внутри блока `try` мы затем можем создать потоки **(5)**, **(6)** — использование `std::launch::async` гарантирует, что каждая задача работает в отдельном потоке. Отметим, что благодаря использованию `std::async` обеспечить безопасность относительно исключений проще, чем в случае простого `std::thread`, потому что деструктор будущего результата присоединит поток. В переменных, захваченных лямбда-функцией, хранится ссылка на очередь, соответствующее обещание для подачи сигнала о готовности, а также копия будущего результата `ready`, полученного из обещания `go`.

Как было описано выше, каждая задача устанавливает свой сигнал `ready`, а затем ждет общего сигнала `ready`, прежде чем начать исполнение тестируемого кода. Главный поток делает всё наоборот — ждет сигналов от обоих потоков **(8)**, а затем сигнализирует им о том, что можно переходить к исполнению тестируемого кода **(9)**.

Напоследок главный поток вызывает функцию `get()` обоих будущих результатов, возвращенных асинхронными вызовами, чтобы дождаться завершения задач **(10)**, **(11)** и проверить получившееся состояние. Отметим, что задача *pop* возвращает извлеченное из очереди значение в будущем результате **(7)**, чтобы мы могли проверить его в утверждении **(11)**.

В случае исключения мы устанавливаем сигнал `go`, чтобы не оказалось висячего потока, и возбуждаем исключение повторно **(12)**. Будущие результаты, соответствующие обеим задачам **(4)**, были объявлены последними, поэтому уничтожаются первыми, и их деструкторы ждут завершения задач, если они еще не завершились.

Хотя служебного кода многовато для тестирования двух простых вызовов, что-то в этом роде все равно необходимо, чтобы проверить именно то, что мы хотим проверить. Например, запуск потока занимает довольно много времени, поэтому если бы мы не заставили потоки ждать сигнала `go`, то поток, помещающий данные, вполне мог бы завершиться еще до запуска потока, извлекающего данные, а это шло бы вразрез с целью данного теста. Благодаря использованию будущих результатов мы можем быть уверены, что оба потока запущены и блокированы в ожидании одного и того же будущего. Как только это будущее наступит, оба потока начнут работать. Привыкнув к этой структуре, вы без труда напишете и другие тесты. Продемонстрированный принцип без труда обобщается на случай, когда в каком-то тесте требуется более двух потоков.

До сих пор мы говорили о *корректности* многопоточного кода. Это, конечно, самая важная, но не единственная цель тестирования. Существенна также его *производительность*, и далее мы займемся этим вопросом.

10.2.6. Тестирование производительности многопоточного кода

Одна из основных причин распараллеливания кода — задействовать многоядерные процессоры для повышения производительности программы. Поэтому проверять, что производительность действительно возросла, так же важно, как при использовании других методов оптимизации.

Когда распараллеливание применяется ради повышения производительности, особый интерес представляет *масштабируемость* — мы хотим, чтобы на машине с 24 ядрами код выполнялся примерно в 24 раза быстрее или обрабатывал в 24 раза больше данных, чем на машине с одним ядром, — при прочих равных условиях. Мы не хотим, чтобы на двухъядерной машине код исполнялся в два раза быстрее, а на 24-ядерной — медленнее. В разделе 8.4.2 мы видели, что если значительная часть программы работает в одном потоке, то выигрыш от распараллеливания надает. Поэтому еще до начала тестирования стоит критически проанализировать общую структуру программы, чтобы понять, можно ли рассчитывать на 24-кратное ускорение или вследствие того, что большая часть программы работает последовательно, коэффициент ускорения вряд ли превысит 3.

В предыдущих главах было показано, что конкуренция между процессорами за доступ к

структуре данных может весьма негативно сказаться на производительности. Программа, которая хорошо масштабируется, когда число процессоров мало, может повести себя никуда не годно при увеличении их числа из-за возрастания конкуренции.

Следовательно, при тестировании производительности многопоточной программы лучше замерять результаты в максимально широком спектре конфигураций, чтобы можно было составить целостное представление о масштабируемости. Как минимум, следует прогнать тесты на однопроцессорной машине и на машине с максимальным числом процессорных ядер, которое вы можете себе позволить.

10.3. Резюме

В этой главе мы рассмотрели различные виды ошибок, связанных с параллелизмом, — от взаимоблокировок и активных блокировок до гонок за данными и других проблематичных состояний гонки. Были описаны различные методы поиска ошибок. Я сформулировал вопросы, над которыми следует поразмыслить, дал рекомендации по написанию тестопригодного кода и рассказал о том, как структурировать тесты для параллельных программ. И, наконец, мы затронули вопрос о некоторых служебных компонентах, которые могут оказать помощь в процессе тестирования.

Приложение A.
Краткий справочник по некоторым конструкциям языка C++

Новый стандарт C++ отнюдь не исчерпывается поддержкой параллелизма; в нем появилось немало других языковых средств и новых библиотек. В этом приложении я вкратце расскажу о тех новых возможностях, которые используются в библиотеке многопоточности и встречаются в этой книге. За исключением модификатора `thread_local` (рассматриваемого в разделе A.8), все они не имеют прямого отношения к параллелизму, однако важны и (или) полезны для написания многопоточного кода. Я ограничился лишь теми конструкциями, которые либо необходимы (например, ссылки на *r*-значения), либо делают код проще и яснее. Поначалу разобраться в программе, где применяются эти конструкции, будет трудно, но, познакомившись с ними поближе, вы согласитесь, что, вообще говоря, включающий их код проще, а не сложнее для понимания. По мере распространения C++11 описываемые средства будут встречаться в программах все чаще.

А теперь, без дальнейших предисловий, начнем с изучения *ссылок на r-значения* — средства, которое широко используется в библиотеке Thread Library для передачи владения (потоками, блокировками и вообще всем на свете) от одного объекта другому.

A.1. Ссылки на *r*-значения

Всякий, кто программировал на C++, знаком со ссылками; в C++ ссылки служат для создания альтернативного имени существующего объекта. Любой доступ к объекту по ссылке, в том числе для модификации, приводит к манипуляциям с исходным объектом. Например:

```
int var = 42;     | Создаем ссылку
int& ref = var;↵┘ на var
ref = 99;         | В результате присваивания ссылке
assert (var == 99);↵┘ изменен оригинал
```

Ссылки, к которым мы все давно привыкли, являются *ссылками на l-значения.* Термин *l-значение* появился еще в языке C и обозначает любую конструкцию, которая может находиться в левой части выражения присваивания, — именованные объекты, объекты, созданные в стеке или в куче, или члены других объектов, то есть сущности, расположенные по определенному адресу в памяти. Термин *r-значение* также происходит из C и обозначает конструкции, которые могут находиться только в правой части выражения присваивания, — например, литералы и временные объекты. Ссылки на *l*-значения можно связать только с *l*-значениями, но не с *r*-значениями. Так, невозможно написать

```
int& i = 42;
```

потому что 42 — это *r*-значение. Впрочем, это не совсем верно; всегда разрешалось связывать *r*-значение с константной ссылкой на *l*-значение:

```
int const& i = 42;
```

Однако в стандарте это исключение сделано сознательно задолго до появления ссылок на *r*-значения, и смысл его в том, чтобы разрешить передавать временные объекты функциям, принимающим ссылки. Благодаря этому механизму становятся возможны неявные преобразования, например, можно написать:

```
void print(std::string const& s);
print("hello");
```

Как бы то ни было, в стандарте C++11 официально введены ссылки на *r-значения*, которые связываются *только* с *r*-значениями, но не с *l*-значениями, и объявляются с помощью двух знаков амперсанда:

```
int&& i = 42;
int j = 42;
int&& k = j;
```

Таким образом, функцию можно перегрузить в зависимости от того, являются параметры *l*-значениями или *r*-значениями, — один вариант будет принимать ссылку на *l*-значение, другой — на *r*-значение. Эта возможность — краеугольный камень *семантики перемещения*.

A.1.1. Семантика перемещения

r-значения — это обычно временные объекты, поэтому их можно спокойно модифицировать; если известно, что параметр функции — r-значение, то его можно использовать как временную память, то есть «позаимствовать» его содержимое без ущерба для корректности программы. Это означает, что вместо *копирования* параметра, являющегося

r-значением, мы можем просто *переместить* его содержимое. В случае больших динамических структур это позволяет сэкономить на выделении памяти и оставляет простор для оптимизации.

Рассмотрим функцию, которая принимает в качестве параметра `std::vector<int>` и хочет иметь его внутреннюю копию для модификации, так чтобы не затрагивать оригинал. Раньше мы для этого должны были принимать параметр как `const`-ссылку на *l*-значение и делать внутреннюю копию:

```
void process_copy(std::vector<int> const& vec_) {
 std::vector<int> vec(vec_);
 vec.push_back(42);
}
```

При этом функция может принимать как *l*-значения, так и *r*-значения, но копирование производится всегда. Однако, если добавить перегруженный вариант, который принимает ссылку на *r*-значение, то в этом случае можно будет избежать копирования, поскольку нам точно известно, что оригинал разрешается модифицировать:

```
void process_copy(std::vector<int>&& vec) {
 vec.push_back(42);
}
```

Если функция является конструктором класса, то можно умыкнуть содержимое *r*-значения и воспользоваться им для создания нового экземпляра. Рассмотрим класс, показанный в листинге ниже. В конструкторе по умолчанию он выделяет большой блок памяти, а в деструкторе освобождает его.

Листинг A.1. Класс с перемещающим конструктором

```
class X {
private:
 int* data;

public:
 X() : data(new int[1000000]) {}

 ~X() {
  delete [] data;
 }

 X(const X& other) : ← (1)
  data(new int[1000000]) {
  std::copy(other.data, other.data + 1000000, data);
 }

 X(X&& other) : ← (2)
  data(other.data) {
  other.data = nullptr;
 }
};
```

Копирующий конструктор **(1)** определяется как обычно: выделяем новый блок памяти и копируем в него данные. Но теперь у нас есть еще один конструктор, который принимает ссылку на *r*-значение **(2)**. Это *перемещающий конструктор*. В данном случае мы копируем только *указатель* на данные, а в объекте `other` остается нулевой указатель. Таким образом,

мы обошлись без выделения огромного блока памяти и сэкономили время на копировании данных из *r*-значения.

В классе `X` перемещающий конструктор — всего лишь оптимизация, но в ряде случаев такой конструктор имеет смысл определять, даже когда копирующий конструктор не предоставляется. Например, идея `std::unique_ptr<>` в том и заключается, что любой ненулевой экземпляр является единственным указателем на свой объект, поэтому копирующий конструктор лишен смысла. Однако же перемещающий конструктор позволяет передавать владение указателем от одного объекта другому, поэтому `std::unique_ptr<>` можно использовать в качестве возвращаемого функцией значения — указатель перемещается, а не копируется.

Чтобы явно переместить значение из именованного объекта, который больше заведомо не будет использоваться, мы можем привести его к типу *r*-значения либо с помощью `static_cast<X&&>`, либо путем вызова функции `std::move()`:

```
X x1;
X x2 = std::move(x1);
X x3 = static_cast<X&&>(x2);
```

Это особенно удобно, когда требуется переместить значение параметра в локальную переменную или переменную-член без копирования, потому что хотя параметр, являющийся ссылкой на *r*-значение, и может связываться с *r*-значениями, но внутри функции он трактуется как *l*-значение:

```
void do_stuff(X&& x_) {
 X a(x_); ← Копируется
 X b(std::move(x_)); ← Перемещается
}                    | r-значение связывается
do_stuff(X());←┘ со ссылкой на r -значение
X x;                 | Ошибка, l -значение нельзя связывать
do_stuff(x);←┘ со ссылкой на r -значение
```

Семантика перемещения сплошь и рядом используется в библиотеке Thread Library — и в случаях, когда копирование не имеет смысла, но сами ресурсы можно передавать, и как оптимизация, чтобы избежать дорогостоящего копирования, когда исходный объект все равно будет уничтожен. Один пример мы видели в разделе 2.2, где `std::move()` использовалась для передачи экземпляра `std::unique_ptr<>` новому потоку, а второй — в разделе 2.3, когда рассматривали передачу владения потоком от одного объекта `std::thread` другому.

Ни один из классов `std::thread`, `std::unique_lock<>`, `std::future<>`, `std::promise<>`, `std::packaged_task<>` не допускает копирования, но в каждом из них имеется перемещающий конструктор, который позволяет передавать ассоциированный ресурс другому экземпляру и возвращать объекты этих классов из функций. Объекты классов `std::string` и `std::vector<>` можно копировать, как и раньше, но дополнительно они обзавелись перемещающими конструкторами и перемещающими операторами присваивания, чтобы избежать копирования данных из *r*-значений.

Стандартная библиотека C++ никогда не делает ничего с объектом, который был явно перемещён в другой объект, кроме его уничтожения или присваивания *ему* значения (путем копирования или, что более вероятно, перемещения). Однако рекомендуется учитывать в инвариантах класса состояние перемещен-из. Например, экземпляр `std::thread`, содержимое которого перемещено, эквивалентен объекту `std::thread`, сконструированному

по умолчанию, а экземпляр `std::string`, бывший источником перемещения, все же находится в согласованном состоянии, хотя не дается никаких гарантий относительно того, что это за состояние (в терминах длины строки или содержащихся в ней символов).

A.1.2. Ссылки на *r*-значения и шаблоны функций

Еще один нюанс имеет отношение к использованию ссылок на *r*-значения в качестве параметров шаблона функции: если параметр функции — ссылка на *r*-значение типа параметра шаблона, механизм автоматического выведения типа аргумента шаблона заключает, что тип — это ссылка на *l*-значение, если функции передано *l*-значение, или обычный не-ссылочный тип, если передано *r*-значение. Фраза получилась довольно запутанной, поэтому приведём пример. Рассмотрим такую функцию:

```
template<typename T>
void foo(T&& t) {}
```

Если при вызове передать ей *r*-значение, как показано ниже, то в качестве `T` выводится тип этого значения:

```
foo(42);
foo(3.14159);
foo(std::string());
```

Но если вызвать `foo`, передав *l-значение*, то механизм выведения типа решит, что `T` — ссылка на *l*-значение:

```
int i = 42;
foo(i);
```

Поскольку объявлено, что параметр функции имеет тип `T&&`, то получается, что это ссылка на ссылку, и такая конструкция трактуется как обычная одинарная ссылка. Таким образом, сигнатура функции `foo<int&>()` такова:

```
void foo<int&>(int& t);
```

Это позволяет одному шаблону функции принимать параметры, являющиеся как *l*-, так и *r*-значениями. В частности, это используется в конструкторе `std::thread` (см. разделы 2.1 и 2.2), чтобы в случае, когда переданный допускающий вызов объект является *r*-значением, его можно было бы не копировать, а переместить во внутреннюю память.

A.2. Удаленные функции

Иногда операция копирования класса лишена смысла. Типичный пример — `std::mutex`. Действительно, что должно было бы получиться в результате копирования мьютекса? Другой пример — `std::unique_lock<>`, экземпляр этого класса является единственным владельцем удерживаемой им блокировки. Честное копирование в этом случае означало бы, что у блокировки два владельца, а это противоречит определению. Передача владения, описанная в разделе A.1.2, имеет смысл, но это не копирование. Уверен, вы назовете и другие примеры.

Стандартная идиома предотвращения копирования класса хорошо известна — объявить копирующий конструктор и копирующий оператор присваивания закрытыми и не предоставлять их реализации. Если теперь какой-нибудь внешний по отношению к классу код попытается скопировать объект такого класса, то произойдёт ошибка на этапе компиляции, а если то же самое попытается сделать член класса или его друг, — то ошибка на этапе компоновки (так как реализации отсутствуют):

```
class no_copies {
public:
 no_copies(){}

private:
 no_copies(no_copies const&); ← Реализаций нет
 no_copies& operator=(no_copies const&);
};

no_copies a; ← Не компилируется
no_copies b(a);
```

Комитет, разрабатывавший стандарт C++11, конечно, знал об этой идиоме, но счел ее не совсем честным приёмом. Поэтому было решено предоставить более общий механизм, применимый и к другим случаям: объявить функцию *удаленной*, включив в ее объявление конструкцию `= delete`. Тогда класс `no_copies` можно записать в виде:

```
class no_copies {
public:
 no_copies() {}
 no_copies(no_copies const&) = delete;
 no_copies& operator=(no_copies const&) = delete;
};
```

Это гораздо нагляднее и четко выражает намерения автора. Кроме того, компилятор может в этом случае выдать более понятное сообщение об ошибке, и к тому же при попытке скопировать объект внутри функции-члена класса ошибка произойдёт уже на этапе компиляции, а не компоновки.

Если, удалив копирующие конструктор и оператор присваивания, вы явно напишете перемещающие конструктор и оператор присваивания, то класс будет допускать только перемещение — как, например, `std::thread` и `std::unique_lock<>`. В следующем листинге приведен пример такого класса.

Листинг A.2. Простой тип, допускающий только перемещение

```
class move_only {
 std::unique_ptr<my_class> data;
```

```
public:
 move_only(const move_only&) = delete;
 move_only(move_only&& other):
  data(std::move(other.data)) {}
 move_only& operator=(const move_only&) = delete;
 move_only& operator=(move_only&& other) {
  data = std::move(other.data);
  return *this;
 }
};
```

```
move_only m1;          | Ошибка, копирующий конструктор объявлен
move_only m2(m1);←┘    | удаленным
move_only m3(std::move(m1));←┐ правильно, имеется переме-
                       | щающий конструктор
```

Объекты, допускающие только перемещение, можно передавать функциям в качестве параметров и возвращать из функций, но если вы захотите переместить содержимое *l*-значения, то должны будете выразить свое намерение явно, воспользовавшись функцией `std::move()` или оператором `static_cast<T&&>`.

Спецификатор `= delete` можно задать для любой функции, а не только для копирующего конструктора и оператора присваивания. Тем самым вы ясно даете понять, что функция отсутствует. Но это еще не все — удаленная функция участвует в разрешении перегрузки, как любая другая, и вызывает ошибку компиляции, только если будет выбрана. Этим можно воспользоваться для исключения некоторых перегруженных вариантов. Например, если функция принимает параметр типа `short`, то сужение типа `int` можно предотвратить, написав перегруженный вариант, который принимает `int`, и объявив его удаленным:

```
void foo(short);
void foo(int) = delete;
```

Любую попытку вызвать `foo` с параметром типа `int` компилятор встретит в штыки, так что вызывающей программе придётся явно привести параметр к типу `short`:

```
foo(42); ← Ошибка, перегрузка для int удалена
foo((short)42); ← Правильно
```

A.3. Умалчиваемые функции

Если механизм удаленных функций позволяет явно объявить, что функция не реализована, то назначение умалчиваемых (defaulted) функций прямо противоположное - это средство указать, что компилятор должен автоматически сгенерировать реализацию функции «по умолчанию». Разумеется, это можно делать только для функций, которые компилятор и так генерирует: конструкторов, деструкторов, копирующих и перемещающих конструкторов, копирующих и перемещающих операторов присваивания.

Зачем это может понадобиться? Есть несколько причин.

• *Чтобы изменить видимость функции.* По умолчанию генерируемые компилятором функции открыты. Если требуется, чтобы они были защищенными или даже закрытыми, то писать их придётся самостоятельно. Но объявив функцию умалчиваемой, вы можете заставить компилятор сгенерировать ее и одновременно изменить уровень доступа.

• *Для документирования.* Если сгенерированной компилятором версии достаточно, то имеет смысл так прямо и сказать. Тогда всякий, кто впоследствии будет читать код, поймёт, что это сделано намеренно.

• *Чтобы заставить компилятор сгенерировать функцию, которую в противном случае он не стал бы генерировать.* Обычно это касается конструкторов по умолчанию, которые автоматически генерируются, только если нет ни одного определенного пользователем конструктора. Если вы хотите, например, определить свой копирующий конструктор, то, объявив конструктор по умолчанию умалчиваемым, заставите компилятор сгенерировать его.

• *Чтобы сделать деструктор виртуальным и при этом генерируемым компилятором.*

• *Чтобы сгенерировать специальный вариант копирующего конструктора, например, принимающий параметр по неконстантной ссылке* (по умолчанию генерируется конструктор, принимающий константную ссылку).

• *Чтобы воспользоваться специальными свойствами сгенерированных компилятором функций, который теряются, если вы сами пишете реализацию.* Подробнее об этом чуть ниже.

Умалчиваемые функции объявляются путем добавления спецификатора `= default`, например:

```
class Y {
private:
 Y() = default; ← Изменяем видимость

public:
 Y(Y&) = default; ← Принимаем не-const ссылку
 T& operator=(const Y&) = default;←┐ объявляем умалчиваемой
                                   │ для документирования
protected:
 virtual ~Y() = default; ← Изменяем видимость и добавляем virtual
};
```

Выше я упомянул, что сгенерированные компилятором функции обладают специальными свойствами, которые невозможно получить от версии, написанной пользователем. Самое существенное отличие заключается в том, что сгенерированная компилятором функция может быть *тривиальной*. Отсюда вытекает ряд следствий.

• Объекты с тривиальными копирующим конструктором, копирующим оператором присваивания и деструктором можно копировать с помощью `memcpy` или `memmove`.

• Литеральные типы, используемые в `constexpr`-функциях (см. раздел А.4) обязаны обладать тривиальными конструктором, копирующим конструктором и деструктором.

• Классы с тривиальными конструктором по умолчанию, копирующим конструктором, копирующим оператором присваивания и деструктором можно использовать в объединении (`union`), в котором определены пользовательские конструктор и деструктор.

• Классы с тривиальными конструктором копирующим оператором присваивания можно использовать вместе с шаблонным классом `std::atomic<>` (см. раздел 5.2.6), то есть передавать значения такого типа атомарным операциям.

Одного объявления функции со спецификатором `= default` недостаточно, чтобы сделать ее тривиальной, для этого класс должен удовлетворять всем прочим условиям, при которых соответствующая функция будет тривиальной. Однако явно написанная пользователем функция не будет тривиальной *никогда*.

Второе различие между классами с функциями, сгенерированными компилятором и написанными пользователем, заключается в том, что класс без написанных пользователем конструкторов может быть *агрегатным* и, стало быть, допускать инициализацию с помощью агрегатного инициализатора:

```
struct aggregate {
 aggregate() = default;
 aggregate(aggregate const&) = default;
 int a;
 double b;
};
aggregate x={42, 3.141};
```

В данном случае `x.a` инициализируется значением `42`, а `x.b` — значением `3.141`.

Третье различие малоизвестно и относится только к конструктору по умолчанию, да и то лишь в классах, удовлетворяющих определенному условию. Рассмотрим такой класс:

```
struct X {
 int a;
};
```

Если экземпляр класса `X` создается без инициализатора, то содержащееся в нем значение (`a`) типа `int` *инициализируется по умолчанию*. Если у объекта статический класс памяти, то значение инициализируется нулем, в противном случае начальное значение произвольно, что может привести к неопределённому поведению, если программа обращается к объекту раньше, чем ему будет присвоено значение:

`X x1;` ← **значение x1.a не определено**

С другой стороны, если инициализировать экземпляр `X` путем явного вызова конструктора по умолчанию, то он получит значение 0:

`X x2 = X();` ← **x2.a == 0**

Это странное свойство распространяется также на базовые классы и члены классов. Если в классе имеется сгенерированный компилятором конструктор по умолчанию, и каждый член самого класса и всех его базовых классов также имеет сгенерированный компилятором конструктор по умолчанию, то переменные-члены самого класса и его базовых классов, принадлежащие встроенным типам, также будут иметь неопределенное значение или будут инициализированы нулями в зависимости от того, вызывался ли явно для внешнего класса его конструктор по умолчанию.

У этого замысловатого и потенциально чреватого ошибками правила есть тем не менее применения, а, если вы пишете конструктор по умолчанию самостоятельно, то это свойство утрачивается; данные-члены (например, `a`) либо всегда инициализируются (коль скоро вы указали значение или явно вызвали конструктор по умолчанию), либо вообще не инициализируются (если вы этого не сделали):

```
X::X() : a() {}    ← всегда a == 0
X::X() : a(42) {}  ← всегда a == 42
X::X() {}          ← (1)
```

Если инициализация `a` при конструировании `X` не производится (как в третьем примере **(1)**), то `a` остается неинициализированным для нестатических экземпляров `X` и инициализируется нулем для экземпляров `X` со статическим временем жизни.

Обычно, если вы вручную напишете хотя бы один конструктор, то компилятор не станет генерировать конструктор по умолчанию. Стало быть, если он вам все-таки нужен, его придётся написать самостоятельно, а тогда это странное свойство инициализации теряется. Однако явно объявив конструктор умалчиваемым, вы можете заставить компилятор сгенерировать конструктор по умолчанию и сохранить это свойство:

```
X::X() = default;
```

Это свойство используется в атомарных типах (см. раздел 5.2), в которых конструктор по умолчанию явно объявлен умалчиваемым. У таких типов начальное значение не определено, если только не выполняется одно из следующих условий: (a) задан статический класс памяти (тогда значение инициализируется нулем); (b) для инициализации нулем явно вызван конструктор по умолчанию; (c) вы сами явно указали начальное значение. Отметим, что в атомарных типах конструктор для инициализации значением объявлен как `constexpr` (см. раздел A.4), чтобы разрешить статическую инициализацию.

A.4. constexpr-функции

Целые литералы, например `42`, — это *константные выражения*. Равно как и простые арифметические выражения, например `23*2-4`. Частью константного выражения могут быть также `const`-переменные любого целочисленного типа, которые сами инициализированы константным выражением:

```
const int i = 23;
const int two_i = i * 2;
const int four = 4;
const int forty_two = two_i - four;
```

Помимо использования константных выражений для инициализации переменных, которые могут использоваться в других константных выражениях, есть ряд случаев, где разрешается применять *только* константные выражения.

• Задание границ массива:

```
int bounds = 99;     | Ошибка, bounds — не константное
int array[bounds];↵┘ | выражение
const int bounds2 = 99; | Правильно, bounds2 — константное
int array2[bounds2];  ↵┘ | выражение
```

• Задание значения параметра шаблона, не являющего типом:

```
template<unsigned size>
struct test {};   | Ошибка, bounds —
                  | не константное
test<bounds> is;↵┘ | выражение
test<bounds2 > ia2;↵┐ Правильно, bounds2 —
                     | константное выражение
```

• Задание непосредственно в определении класса инициализатора для переменной-члена класса целочисленного типа со спецификаторами `static const`:

```
class X {
 static const int the_answer = forty_two;
};
```

• Употребление в инициализаторах встроенных типов или агрегатов, применяемых для статической инициализации:

```
struct my_aggregate {
 int a;
 int b;
};
static my_aggregate ma1 =| Статическая
 { forty_two, 123 };    ↵┘ инициализация
int dummy = 257;                          | Динамическая
static my_aggregate ma2 = {dummy, dummy};↵┘ инициализация
```

Такая статическая инициализация полезна для предотвращения зависимости от порядка инициализации и состояний гонки.

Всё это не ново и было описано еще в стандарте C++ 1998 года. Но в новом стандарте появилось и дополнение в части константных выражений — ключевое слово `constexpr`.

Ключевое слово `constexpr` применяется главным образом как модификатор функции. Если параметр и возвращаемое функцией значение удовлетворяют определенным условиям,

а тело функции достаточно простое, то в ее объявлении можно указать `constexpr` и использовать функцию в константных выражениях. Например:

```
constexpr int square(int x) {
 return x*x;
}
int array[square(5)];
```

В этом случае массив `array` будет содержать 25 значений, потому что функция `square` объявлена как `constexpr`. Конечно, из того, что функцию *можно* использовать в константном выражении, еще не следует, что любой случай ее использования автоматически будет константным выражением:

```
int dummy = 4;                    (1) Ошибка, dummy — не константное
int array[square(dummy)];↵┘       выражение
```

В этом примере `dummy` не является константным выражением **(1)**, поэтому не является таковым и `square(dummy)`. Это обычный вызов функции, и, следовательно, для задания границ массива array его использовать нельзя.

A.4.1. `constexpr` и определенные пользователем типы

До сих пор мы употребляли в примерах только встроенные типы — такие, как `int`. Но в новом стандарте C++ допускаются константные выражения любого типа, удовлетворяющего требованиям, предъявляемым к *литеральному типу*. Чтобы тип класса можно было считать литеральным, должны быть выполнены все следующие условия:

• в классе должен существовать тривиальный копирующий конструктор;

• в классе должен существовать тривиальный деструктор;

• все нестатические переменные-члены данного класса и его базовых классов должны иметь тривиальный тип;

• в классе должен существовать либо тривиальный конструктор по умолчанию, либо `constexpr`-конструктор, отличный от копирующего конструктора.

О `constexpr`-конструкторах мы поговорим чуть ниже. А пока обратимся к классам с тривиальным конструктором по умолчанию. Пример такого класса приведён ниже:

```
class CX {
private:
 int a;
 int b;

public:
 CX() = default; ←(1)
 CX(int a_, int b_) : ←(2)
  a(a_), b(b_) {}

 int get_a() const {
  return a;
 }

 int get_b() const {
  return b;
 }
```

```
 int foo() const {
  return a + b;
 }
};
```

Здесь мы явно объявили конструктор по умолчанию **(1)** *умалчиваемым* (см. раздел А.3), чтобы сохранить его тривиальность, несмотря на наличие определённого пользователем конструктора **(2)**. Таким образом, этот тип удовлетворяет всем требованиям к литеральному типу и, значит, его можно использовать в константных выражениях. К примеру, можно написать `constexpr`-функцию, которая создает новые экземпляры этого класса:

```
constexpr CX create_cx() {
 return CX();
}
```

Можно также написать простую `constexpr`-функцию, которая копирует свой параметр:

```
constexpr CX clone(CX val) {
 return val;
}
```

Но это практически и всё, что можно сделать, — `constexpr`-функции разрешено вызывать только другие `constexpr`-функции. Тем не менее, допускается применять спецификатор `constexpr` к функциям-членам и конструкторам CX:

```
class CX {
private:
 int a;
 int b;

public:
 CX() = default;
 constexpr CX(int a_, int b_): a(a_), b(b_) {}

 constexpr int get_a() const { ← (1)
  return a;
 }

 constexpr int get_b() { ← (2)
  return b;
 }

 constexpr int foo() {
  return a + b;
 }
};
```

Отметим, что теперь квалификатор `const` в функции `get_a()` **(1)** избыточен, потому что он и так подразумевается ключевым словом `constexpr`. Функция `get_b()` достаточно «константная» несмотря на то, что квалификатор `const` опущен **(2)**. Это дает возможность строить более сложные `constexpr`-функции, например:

```
constexpr CX make_cx(int a) {
 return CX(a, 1);
}

constexpr CX half_double(CX old) {
 return CX(old.get_a()/2, old.get_b()*2);
}
```

```
constexpr int foo_squared(CX val) {
 return square(val.foo());
}

int array[foo_squared(
 half_double(make_cx(10)))]; ← 49 элементов
```

Всё это, конечно, интересно, но уж слишком много усилий для того, чтобы всего лишь вычислить границы массива или значение целочисленной константы. Основное же достоинство константных выражений и `constexpr`-функций в контексте пользовательских типов заключается в том, что объекты литерального типа, инициализированные константным выражением, инициализируются статически и, следовательно, не страдают от проблем, связанных с зависимостью от порядка инициализации и гонок.

```
CX si = half_double(CX(42, 19));
```

Это относится и к конструкторам. Если конструктор объявлен как `constexpr`, а его параметры — константные выражения, то такая инициализация считается *константной инициализацией* и происходит на этапе статической инициализации. *Это одно из наиболее важных изменений в стандарте C++11 с точки зрения параллелизма*: разрешив статическую инициализацию для определенных пользователем конструкторов, мы предотвращаем состояния гонки во время инициализации, поскольку объекты гарантированно инициализируются до начала выполнения программы.

Особенно существенно это для таких классов, как `std::mutex` (см. раздел 3.2.1) и `std::atomic<>` (см. раздел 5.2.6), поскольку иногда мы хотим, чтобы некий глобальный объект синхронизировал доступ к другим переменным, но так, чтобы не было гонок при доступе к нему самому. Это было бы невозможно, если бы конструктор мьютекса мог стать жертвой гонки, поэтому конструктор по умолчанию в классе `std::mutex` объявлен как `constexpr`, чтобы инициализация мьютекса всегда производилась на этапе статической инициализации.

A.4.2. `constexpr`-объекты

До сих пор мы говорили о применении `constexpr` к функциям. Но этот спецификатор можно применять и к объектам. Чаще всего, так делают для диагностики; компилятор проверяет, что объект инициализирован константным выражением, `constexpr`-конструктором или агрегатным инициализатором, составленным из константных выражений. Кроме того, объект автоматически объявляется как `const`:

```
constexpr int i = 45;← Правильно
constexpr std::string s("hello");←¬ Ошибка, std::string —
int foo();                          | не литеральный тип
constexpr int j = foo();← Ошибка, foo() не объявлена как constexpr
```

A.4.3. Требования к `constexpr`-функциям

Чтобы функцию можно было объявить как `constexpr`, она должна удовлетворять нескольким требованиям. Если эти требования не выполнены, компилятор сочтет наличие

спецификатора `constexpr` ошибкой. Требования таковы:

• все параметры должны иметь литеральный тип;

• возвращаемое значение должно иметь литеральный тип;

• тело функции может содержать только предложение `return` и ничего больше;

• выражение в предложении `return` должно быть константным;

• любой конструктор или оператор преобразования, встречающийся в выражении для вычисления возвращаемого значения, должен быть объявлен как `constexpr`.

На самом деле, это вполне понятные требования: у компилятора должна быть возможность встроить вызов функции в константное выражение, и при этом оно должно остаться константным. Кроме того, запрещается что-либо изменять; `constexpr`-функции являются *чистыми*, то есть не имеют побочных эффектов.

К `constexpr`-функциям, являющимся членами класса, предъявляются дополнительные требования:

• `constexpr` функции-члены не могут быть виртуальными;

• класс, членом которого является функция, должен иметь литеральный тип.

Для `constexpr`-конструкторов действуют другие правила:

• тело конструктора должно быть пустым;

• все базовые классы должны быть инициализированы;

• все нестатические данные-члены должны быть инициализированы;

• все выражения, встречающиеся в списке инициализации членов, должны быть константными;

• конструкторы, выбранные для инициализации данных-членов и базовых классов, должны быть `constexpr`-конструкторами;

• все конструкторы и операторы преобразования, используемые для конструирования данных-членов и базовых классов в соответствующем выражении инициализации, должны быть объявлены как `constexpr`.

Это тот же набор правил, что и для функций, с тем отличием, что возвращаемого значения нет, а, значит, нет и предложения `return`. Вместо возврата значения конструктор инициализирует базовые классы и данные-члены в списке инициализации членов. Тривиальные копирующие конструкторы неявно объявлены как `constexpr`.

A.4.4. `constexpr` и шаблоны

Спецификатор `constexpr` в объявлении шаблона функции или функции-члене шаблонного класса игнорируется, если типы параметров и возвращаемого значения для данной конкретизации шаблона не являются литеральными. Это позволяет писать шаблоны функций, которые становятся `constexpr`-функциями, если параметры шаблона имеют подходящие типы, и обычными встраиваемыми функциями в противном случае. Например:

```
template<typename T>
constexpr T sum(T a, T b) {
 return a + b;
}                                  | Правильно, sum<int>
constexpr int i = sum(3, 42);↵┘ constexpr
std::string s =
 sum(std::string("hello"),     | Правильно, но sum<std::string>
```

```
        std::string(" world"));↵
```
Не constexpr

Функция должна удовлетворять также всем остальным требованиям, предъявляемым к `constexpr`-функциям. Нельзя включить в тело шаблона функции, объявленного как `constexpr`, несколько предложений только потому, что это шаблон; компилятор сочтет это ошибкой.

A.5. Лямбда-функции

Лямбда-функции — одно из самых интересных новшеств в стандарте C++11, потому что они позволяют существенно упростить код и исключить многие стереотипные конструкции, которые применяются при написании объектов, допускающих вызов. Синтаксис лямбда-функций в C++11 позволяет определить функцию в той точке выражения, где она необходима. Это отличное решение, например, для передачи предикатов функциям ожидания из класса `std::condition_variable` (как в примере из раздела 4.1.1), потому что дает возможность кратко выразить семантику в терминах доступных в данной точке переменных, а не запоминать необходимое состояние в переменных-членах класса с оператором вызова.

В простейшем случае *лямбда-выражение* определяет автономную функцию без параметров, которая может пользоваться только глобальными переменными и функциями. У нее даже нет возвращаемого значения. Такое лямбда-выражение представляет собой последовательность предложений, заключенных в фигурные скобки, которым предшествуют квадратные скобки (так называемый *лямбда-интродуктор*):

```
[] { ← Лямбда-выражение начинается с []
 do_stuff();       | Конец определения
 do_more_stuff();  | лямбда-выражения
} ();              ←┘ и его вызов
```

В данном случае лямбда-выражение сразу вызывается, потому что за ним следуют круглые скобки, однако это необычно. Ведь если вы хотите вызывать его напрямую, то можно было бы вообще обойтись без лямбда-выражения и записать составляющие его предложения прямо в коде. Чаще лямбда-выражение передаётся в шаблон функции, который принимает допускающий вызов объект в качестве одного из параметров. Но тогда ему, скорее всего, нужны параметры или возвращаемое значение или то и другое вместе. Если лямбда-функция принимает параметры, то их можно указать после лямбда-интродуктора с помощью списка параметров, как для обычной функции. Так, в следующем примере мы выводим все элементы вектора на `std::cout`, разделяя их символами новой строки:

```
std::vector<int> data = make_data();
std::for_each(data.begin(), data.end(),
 [](int i){std::cout << i << "\n";});
```

С возвращаемыми значениями всё почти так же просто. Если тело лямбда-функции состоит из единственного предложения `return`, то тип возвращаемого ей значения совпадает с типом возвращаемого выражения. Например, такую простую лямбда-функцию можно было бы использовать для проверки флага, ожидаемого условной переменной `std::condition_variable` (см. раздел 4.1.1).

Листинг A.4. Простая лямбда-функция с выводимым типом возвращаемого значения

```
std::condition_variable cond;
 bool data_ready;
 std::mutex m;

void wait_for_data() {
 std::unique_lock<std::mutex> lk(m);
 cond.wait(lk, []{return data_ready;}); ← (1)
}
```

Тип значения, возвращаемого лямбда-функцией, которая передана `cond.wait()` **(1)**, выводится из типа переменной `data_ready`, то есть совпадает с `bool`. Когда условная переменная получает сигнал, она вызывает эту лямбда-функцию, захватив предварительно мьютекс, и `wait()` возвращает управление, только если `data_ready` равно `true`.

Но что если невозможно написать тело лямбда-функции, так чтобы оно содержало единственное предложение `return`? В таком случае тип возвращаемого значения следует задать явно. Это *можно* сделать и тогда, когда тело функции состоит из единственного предложения `return`, но *обязательно*, если тело более сложное. Для задания типа возвращаемого значения нужно поставить после списка параметров функции стрелку (`->`), а за ней указать тип. Если лямбда-функция не имеет параметров, то список параметров (пустой) все равно необходим, иначе задать тип возвращаемого значения невозможно. Таким образом, предикат, проверяемый условной переменной, можно записать так:

```
cond.wait(lk, []()->bool{ return data_ready; });
```

Лямбда-функции с явно заданным типом возвращаемого значения можно использовать, например, для записи сообщений в журнал или для более сложной обработки:

```
cond.wait(lk, []()->bool {
 if (data_ready) {
  std::cout << "Данные готовы" << std::endl;
  return true;
 } else {
  std::cout <<
   "Данные не готовы, продолжаю ждать" << std::endl;
  return false;
 }
});
```

Даже такие простые лямбда-функции весьма полезны и существенно упрощают код, но их истинная мощь проявляется, когда требуется запомнить локальные переменные.

A.5.1. Лямбда-функции, ссылающиеся на локальные переменные

Лямбда-функции с *лямбда-интродуктором* вида `[]` не могут ссылаться на локальные переменные из объемлющей области видимости; им разрешено использовать только глобальные переменные и то, что передано в параметрах. Чтобы получить доступ к локальной переменной, ее нужно *захватить* (capture). Проще всего захватить все переменные в локальной области видимости, указав лямбда-интродуктор вида `[=]`. Теперь лямбда-функция может получить доступ к *копиям* локальных переменных на тот момент, когда эта функция была создана.

Рассмотрим этот механизм на примере следующей простой функции:

```
std::function<int(int)> make_offseter(int offset) {
 return [=](int j){return offset+j;};
}
```

При каждом вызове `make_offseter` с помощью обертки `std::function<>` создается новый содержащий лямбда-функцию объект. Возвращенная функция добавляет указанное смещение к любому переданному ей параметру. Например, следующая программа

```
int main() {
 std::function<int(int)> offset_42 = make_offseter(42);
 std::function<int(int)> offset_123 = make_offseter(123);
 std::cout <<
  offset_42(12) << "," << offset_123(12) << std::endl;
```

```
 std::cout <<
  offset_42(12) << "," << offset_123(12) << std::endl;
}
```

два раза выведет числа `54, 135`, потому что функция, возвращенная после первого обращения к `make_offseter`, всегда добавляет 42 к переданному ей аргументу Напротив, функция, возвращенная после второго обращения к `make_offseter`, добавляет к своему аргументу 123. Это самый безопасный вид захвата локальных переменных — все значения копируются, поэтому лямбда-функцию можно вернуть и вызывать вне контекста функции, в которой она была создана. Но это не единственно возможное решение, можно захватывать локальные переменные и по ссылке. В таком случае попытка вызвать лямбда-функцию после того, как переменные, на которые указывают ссылки, были уничтожены в результате выхода из области видимости объемлющей их функции или блока, приведёт к неопределённому поведению, точно так же, как обращение к уничтоженной переменной в любом другом случае.

Лямбда-функция, захватывающая все локальные переменные по ссылке, начинается интродуктором `[&]`:

```
int main() {
 int offset = 42;                          ← (1)
 std::function<int(int)> offset_a =
  [&](int j){return offset + j;};          ← (2)
 offset = 123;                             ← (3)
 std::function<int(int)> offset_b =
  [&](int j){return offset + j;};          ← (4)
 std::cout <<
  offset_a(12) << "," << offset_b(12) << std::endl; ← (5)
 offset = 99;                                       ← (6)
 std::cout <<
  offset_a(12) << "," << offset_b(12) << std::endl; ← (7)
}
```

Если функция `make_offseter` из предыдущего примера захватывала копию смещения `offset`, то функция `offset_a` в этом примере, начинающаяся интродуктором `[&]`, захватывает `offset` по ссылке **(2)**. Неважно, что начальное значение `offset` было равно 42 **(1)**; результат вызова `offset_a(12)` зависит от текущего значения `offset`. Значение `offset` было изменено на 123 **(3)** перед порождением второй (идентичной) лямбда-функции `offset_b` **(4)**, но эта вторая функция снова производит захват по ссылке, поэтому результат, как и прежде, зависит от текущего значения `offset`.

Теперь при печати первой строки **(5)**, `offset` всё еще равно 123, поэтому печатаются числа `133, 135`. Однако к моменту печати второй строки **(7)** `offset` стало равно 99 **(6)**, поэтому печатается `111, 111`. И `offset_a`, и `offset_b` прибавляют текущее значение `offset` (99) к переданному аргументу (12).

Но ведь это C++, поэтому вам не обязательно выбирать между всем или ничем; вполне можно захватывать одни переменные по значению, а другие по ссылке. Более того, можно даже указывать, какие именно переменные захватить. Нужно лишь изменить лямбда-интродуктор. Если требуется *скопировать* все видимые переменные, кроме одной-двух, то воспользуйтесь интродуктором `[=]`, но после знака равенства перечислите переменные, захватываемые по ссылке, предпослав им знаки амперсанда. В следующем примере

печатается `1239`, потому что переменная `i` копируется в лямбда-функцию, а `j` и `k` захватываются по ссылке:

```
int main() {
 int i=1234, j=5678, k=9;
 std::function<int()> f=[=,&j,&k] {return i+j+k;};
 i = 1;
 j = 2;
 k = 3;
 std::cout << f() << std::endl;
}
```

Можно поступить и наоборот — по умолчанию захватывать по ссылке, но некоторое подмножество переменных копировать. В таком случае воспользуйтесь интродуктором `[&]`, а после знака амперсанда перечислите переменные, захватываемые по значению. В следующем примере печатается `5688`, потому что `i` захватывается по ссылке, а `j` и `k` копируются:

```
int main() {
 int i=1234, j=5678, k= 9;
 std::function<int()> f=[&,j,k] {return i+j+k;};
 i = 1;
 j = 2;
 k = 3;
 std::cout << f() << std::endl;
}
```

Если требуется захватить только именованные переменные, то можно опустить знак `=` или `&` и просто перечислить захватываемые переменные, предпослав знак амперсанда тем, что должны захватываться по ссылке, а не по значению. В следующем примере печатается `5682`, потому что `i` и `k` захвачены по ссылке, а `j` скопирована

```
int main() {
 int i=1234, j=5678, k=9;
 std::function<int()> f=[&i, j, &k] {return i+j+k;};
 i = 1;
 j = 2;
 k = 3;
 std::cout << f() << std::endl;
}
```

Последний способ заодно гарантирует, что захвачены только необходимые переменные, потому что ссылка на локальную переменную, отсутствующую в списке захвата, приведёт к ошибке компиляции. Выбирая этот вариант, нужно соблюдать осторожность при доступе к членам класса, если лямбда-функция погружена в функцию-член класса. Члены класса нельзя захватывать непосредственно; если к ним необходим доступ из лямбда-функции, то необходимо захватить указатель `this`, включив его в список захвата. В следующем примере лямбда-функция захватывает `this` для доступа к члену класса `some_data`:

```
struct X {
 int some_data;

 void foo(std::vector<int>& vec) {
  std::for_each(vec.begin(), vec.end(),
   [this](int& i){ i += some_data; });
 }
};
```

В контексте параллелизма лямбда-функции особенно полезны для задания предикатов

функции `std::condition_variable::wait()` (см. раздел 4.1.1) и в сочетании с `std::packaged_task<>` (раздел 4.2.1) или пулами потоков для упаковки небольших задач. Их можно также передавать конструктору `std::thread` в качестве функций потока (раздел 2.1.1) и в качестве исполняемой функции в таких параллельных алгоритмах, как `parallel_for_each()` (раздел 8.5.1).

A.6. Шаблоны с переменным числом параметров

Функции с переменным числом параметров, например `printf`, используются уже давно, а теперь появились и шаблоны с переменным числом параметров (variadic templates). Такие шаблоны применяются во многих местах библиотеки C++ Thread Library. Например, конструктор `std::thread` для запуска потока (раздел 2.1.1) — это шаблон функции с переменным числом параметров, а `std::packaged_task<>` (раздел 4.2.2) — шаблон класса с переменным числом параметров. С точки зрения пользователя, достаточно знать, что шаблон принимает неограниченное количество параметров, но если вы хотите написать такой шаблон или просто любопытствуете, как это работает, то детали будут небезынтересны.

При объявлении шаблонов с переменным числом параметров, по аналогии с обычными функциями, употребляется многоточие (. . .) в списке параметров шаблона:

```
template<typename ... ParameterPack>
class my_template {};
```

Переменное число параметров допустимо и в частичных специализациях шаблона, даже если основной шаблон содержит фиксированное число параметров. Например, основной шаблон `std::packaged_task<>` (раздел 4.2.1) — это простой шаблон с единственным параметром:

```
template<typename FunctionType>
class packaged_task;
```

Однако этот основной шаблон нигде не конкретизируется, а служит лишь основой для частичных специализаций:

```
template<typename ReturnType, typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

Именно внутри частичной специализации и содержится реальное определение класса; в главе 4 мы видели, что для объявления задачи, которая принимает параметры типа `std::string` и `double` и возвращает результат в виде объекта `std::future<int>`, можно написать `std::packaged_task<int(std::string, double)>`.

На примере этого объявления демонстрируются два дополнительных свойства шаблонов с переменным числом параметров. Первое сравнительно простое: разрешается в одном объявлении задавать как обычные параметры шаблона (скажем `ReturnType`), так и переменные (`Args`). Второе свойство — это использование `Args...` в списке аргументов специализации шаблона для обозначения того, что здесь должны быть перечислены фактические типы, подставляемые вместо `Args` в точке конкретизации шаблона. На самом деле, поскольку это частичная специализация, то работает она, как сопоставление с образцом; типы, встречающиеся в контексте конкретизации, запоминаются как `Args`. Переменное множество параметров `Args` называется *пакетом параметров* (parameter pack), а конструкция `Args...` — расширением пакета.

Как и для обычных функций с переменным числом параметров, переменная часть может быть как пустым списком, так и содержать много элементов. Например, в конкретизации `std::packaged_task<my_class()>` параметром `ReturnType` является `my_class`, а пакет параметров `Args` пуст. С другой стороны, в конкретизации `std::packaged_task<void(int, double, my_class&, std::string*)>` параметр `ReturnType` — это `void`, и `Args` — список, состоящий из элементов `int`, `double`, `my_class&`, `std::string*`.

A.6.1. Расширение пакета параметров

Мощь шаблонов с переменным числом параметров связана с тем, что можно делать при расширении пакета, — мы отнюдь не ограничены простым расширением списка типов. Прежде всего, расширение пакета можно использовать всюду, где требуется список типов, например, в качестве списка аргументов другого шаблона:

```
template<typename ... Params>
struct dummy {
 std::tuple<Params...> data;
};
```

В данном случае единственная переменная-член `data` представляет собой конкретизацию `std::tuple<>`, содержащую все заданные типы, то есть в классе `dummy<int, double, char>` имеется член типа `std::tuple<int, double, char>`. Расширение пакета можно комбинировать с обычными типами:

```
template<typename ... Params>
struct dummy2 {
 std::tuple<std::string, Params...> data;
};
```

На этот раз класс `tuple` имеет дополнительный (первый) член типа `std::string`. Есть еще одна красивая возможность: разрешается определить образец, в который будут подставляться все элементы расширения пакета. Для этого в конце образца размещается многоточие `...`, обозначающее расширение пакета. Например, вместо кортежа элементов тех типов, которые перечислены в пакете параметров, можно создать кортеж указателей на такие типы или даже кортеж интеллектуальных указателей `std::unique_ptr<>` на них:

```
template<typename ... Params>
struct dummy3 {
 std::tuple<Params* ...> pointers;
 std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

Типовое выражение может быть сколь угодно сложным при условии, что в нем встречается пакет параметров и после него находится многоточие `...`, обозначающее расширение. Во время расширения пакета параметров каждый элемент пакета подставляется в типовое выражение и порождает соответственный элемент в результирующем списке. Таким образом, если пакет параметров `Params` содержит типы `int`, `int`, `char`, то расширение выражения `std::tuple<std::pair<std::unique_ptr<Params>, double> ... >` дает `std::tuple<std::pair<std::unique_ptr<int>, double>, std::pair<std::unique_ptr<int>, double>, std::pair<std::unique_ptr<char>, double>>`. Если расширение пакета используется в качестве списка аргументов шаблона, то шаблон не обязан иметь переменные параметры, но если таковых действительно нет, то размер пакета должен быть в точности равен количеству требуемых параметров шаблона:

```
template<typename ... Types>
struct dummy4 {
 std::pair<Types...> data;
};                          | Правильно, данные имеют
dummy4<int, char> a;←┘ вид std::pair<int, char>
dummy4<int> b; ← Ошибка, нет второго типа
dummy4<int, int, int> c;← Ошибка, слишком много типов
```

Еще один способ применения расширения пакета — объявление списка параметров

функции:

```
template<typename ... Args>
void foo(Args ... args);
```

При этом создается новый пакет параметров `args`, являющийся списком параметров функции, а не списком типов, и его можно расширить с помощью ..., как и раньше. Теперь для объявления параметров функции можно использовать образец, в который производится подстановка типов из расширения пакета, — точно так же, как при подстановке расширения пакета в образец в других местах. Например, вот как это применяется в конструкторе `std::thread`, чтобы все аргументы функции принимались по ссылке на *r*-значение (см. раздел А.1):

```
template<typename CallableType, typename ... Args>
thread::thread(CallableType&& func, Args&& ... args);
```

Теперь пакет параметров функции можно использовать для вызова другой функции, указав расширение пакета в списке аргументов вызываемой функции. Как и при расширении типов, образец можно использовать для каждого выражения в результирующем списке аргументов. Например, при работе со ссылками на r-значения часто применяется идиома, заключающаяся в использовании `std::forward<>` для сохранения свойства «является r-значением» переданных функции аргументов:

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args) {
 foo(std::forward<ArgTypes>(args)...);
}
```

Отметим, что в этом случае расширение пакета содержит как пакет типов `ArgTypes`, так и пакет параметров функции `args`, а многоточие расположено после всего выражения в целом. Если вызвать `bar` следующим образом:

```
int i;
bar(i, 3.141, std::string("hello "));
```

то расширение примет такой вид:

```
template<>
void bar<int&, double, std::string>(
 int& args_1,
 double&& args_2,
 std::string&& args_3) {
 foo(std::forward<int&>(args_1),
 std::forward<double>(args_2),
 std::forward<std::string>(args_3));
}
```

и, следовательно, первый аргумент правильно передается функции `foo` как ссылка на *l*-значение, а остальные — как ссылки на *r*-значения.

И последнее, что можно сделать с пакетом параметров, — это узнать его размер с помощью оператора `sizeof...`. Это совсем просто: `sizeof...(p)` возвращает число элементов в пакете параметров `p`. Неважно, является ли `p` пакетом параметров-типов или пакетом аргументов функции, — результат будет одинаковый. Это, пожалуй, единственный случай, где пакет параметров употребляется без многоточия, поскольку многоточие уже является частью оператора `sizeof...`. Следующая функция возвращает число переданных ей аргументов:

```
template<typename ... Args>
unsigned count_args(Args ... args) {
 return sizeof... (Args);
}
```

Как и для обычного оператора `sizeof`, результатом `sizeof...` является константное выражение, которое, следовательно, можно использовать для задания границ массива и т.п.

A.7. Автоматическое выведение типа переменной

C++ — статически типизированный язык: тип любой переменной известен на этапе компиляции. Более того, программист обязан указать тип каждой переменной. В некоторых случаях имена оказываются очень громоздкими, например:

```
std::map<std::string, std::unique_ptr<some_data>> m;
std::map<std::string, std::unique_ptr<some_data>>::iterator
iter = m.find("my key");
```

Традиционно для решения этой проблемы использовались псевдонимы типов (`typedef`), позволяющие сократить длину идентификатора типа и избавиться от потенциальных проблем несовместимости типов. Этот способ работает и в C++11, но появился и новый: если переменная инициализируется в объявлении, то в качестве ее типа можно указать `auto`. Тогда компилятор автоматически выведет тип переменной из типа инициализатора. Следовательно, приведенный выше пример итератора можно записать и так:

```
auto iter = m.find("my key");
```

Спецификатор `auto` необязательно употреблять изолированно; его можно использовать в сочетании с другими спецификаторами для объявления `const`-переменных, а также указателей и ссылок. Вот несколько примеров объявления переменных с помощью `auto` и дополнительных конструкций:

```
auto i = 42;         // int
auto& j = i;         // int&
auto const k = i;    // int const
auto* const p = &i; // int * const
```

Правила выведения типа переменной основаны на правилах, применяемых в другом месте языка, где выводятся типы: параметры шаблонов функций. В объявлении вида

```
Какое-то-типовое-выражение-включающее-auto
var = some-expression;
```

переменная `var` имеет тот же тип, который был бы выведен, если бы она встречалась в качестве параметра шаблона функции, объявленного с таким же типовым выражением, только `auto` заменяется именем типового параметра шаблона:

```
template<typename T>
void f(type-expression var);
f(some-expression);
```

Это означает, что тип массива сводится к указателю, а ссылки опускаются, если только в типовом выражении переменная явно не объявлена как ссылка. Например:

```
int some_array[45];
auto p = some_array; // int*
int& r = *p;
auto x = r;          // int
auto& y = r;         // int&
```

Это позволяет существенно упростить объявление переменных, особенно в случаях, когда полный идентификатор типа очень длинный или даже неизвестен (например, тип результата вызова функции в шаблоне).

A.8. Поточно-локальные переменные

У поточно-локальной переменной имеется отдельный экземпляр в каждом потоке программы. Для объявления поточно-локальной переменной служит ключевое слово `thread_local`. Поточно-локальными могут быть переменные с областью видимости пространства имен, статические члены классов и локальные переменные. Говорят, что они имеют *потоковое время жизни* (thread storage duration):

```
thread_local int x;←┐ Поточно-локальная переменная в
                    | области видимости пространства
                    | имен
class X                                 | Поточно-локальная
{                                       | статическая пере-
 static thread_local std::string s;←┘   менная-член класса
};
                                        | Необходимо
static thread_local std::string X::s;←┘ определение X::s

void foo() {
                                        | Поточно-локальная
 thread_local std::vector<int> v;←┘     локальная переменная
}
```

Поточно-локальные переменные в области видимости пространства имен и поточно-локальные статические члены класса конструируются раньше первого использования переменной в той же единице трансляции, но *насколько раньше* не оговаривается. В одних реализациях поточно-локальные переменные могут конструироваться при запуске потока, в других — непосредственно перед первым использованием в каждом потоке, в третьих — еще в какой-то момент. Возможен и смешанный подход в зависимости от контекста. На самом деле, если ни одна из поточно-локальных переменных в данной единице трансляции не используется, то не гарантируется, что они вообще будут сконструированы. Это позволяет динамически загружать модули, содержащие поточно-локальные переменные — они будут сконструированы в данном потоке при первом обращении потока к переменной из динамически загруженного модуля.

Поточно-локальные переменные, объявленные внутри функции, инициализируются, когда поток управления впервые проходит через объявление переменной в данном потоке. Если функция в данном потоке не вызывалась, то объявленные в ней поточно-локальные переменные не будут сконструированы. Точно такое же поведение характерно для локальных статических переменных, только в этом случае оно применяется в каждом потоке по отдельности.

У поточно-локальных переменных есть и другие общие черты со статическими переменными — они инициализируются нулями перед последующей инициализацией (например, динамической) и, если конструктор поточно-локальной переменной возбуждает исключение, то вызывается функция `std::terminate()`, которая аварийно завершает приложение.

Деструкторы всех поточно-локальных переменных, сконструированных в данном потоке, вызываются после возврата из функции потока в порядке, обратном

конструированию. Поскольку порядок инициализации не определён, то необходимо гарантировать отсутствие взаимозависимостей между деструкторами таких переменных. Если деструктор поточно-локальной переменной возбуждает исключение, то вызывается функция `std::terminate()`, как и при конструировании.

Поточно-локальные переменные, принадлежащие некоторому потоку, уничтожаются также в случае, когда *данный поток* вызывает `std::exit()` или возвращается из `main()` (что эквивалентно вызову `std::exit()` со значением, которое вернула `main()`). Если другие потоки продолжают работать, когда приложение завершается, то деструкторы принадлежащих им поточно-локальных переменных *не* вызываются.

Хотя поточно-локальные переменные, принадлежащие разным потокам, имеют разные адреса, все же можно получать обычный указатель на такую переменную. Этот указатель адресует объект в том потоке, где указатель был получен, и, следовательно, его можно использовать для предоставления доступа к этому объекту из других потоков. Попытка доступа к уже уничтоженному объекту является неопределенным поведением (как всегда), потому, передавая указатель на поточно-локальную переменную в другой поток, следите за тем, чтобы он не разыменовывался после завершения потока-владельца.

A.9. Резюме

В этом приложении мы смогли лишь пробежаться по верхам новых языковых средств, появившихся в стандарте C++11, поскольку нас интересовали лишь те возможности, которые активно используются в библиотеке Thread Library. Из других средств стоит отметить статические утверждения, строго типизированные перечисления, делегирующие конструкторы, поддержку Unicode, псевдонимы шаблонов, новую универсальную последовательность инициализации, а также ряд других, более мелких изменений. Подробное описание всех новых средств выходит далеко за рамки этой книги и, пожалуй, заслуживает отдельного тома. Лучший из существующих на данный момент обзор всего множества изменений, наверное, приведён в FAQ'e по C++11[21] Бьярна Страуструпа, хотя популярные учебники по C++ в скором времени, вероятно, будут переизданы с учетом всех новшеств.

Я надеюсь, что в этом кратком введении в новые языковые средства мне все же удалось объяснить, как они соотносятся с библиотекой Thread Library, и что с его помощью вы сможете писать и понимать многопоточный код, в котором эти средства используются. Но не забывайте, что это отнюдь не полный справочник и не учебник по использованию новых возможностей языка. Если вы намерены активно пользоваться ими, то рекомендую приобрести достаточно подробную книгу, которая позволит задействовать их в полной мере.

Приложение B. Краткое сравнение библиотек для написания параллельных программ

Поддержка параллелизма и многопоточности в языках программирования и библиотеках не является чем-то новым, хотя включение ее в стандарт C++ — действительно новшество. Например, в Java многопоточность поддерживалась уже в самой первой версии; на платформах, согласованных со стандартом POSIX, имеется интерфейс из языка C к средствам многопоточности, предоставляемым операционной системой, а язык Erlang поддерживает параллелизм на основе передачи сообщений. Существуют даже библиотеки классов для C++, например Boost, которые обертывают программный интерфейс к средствам многопоточности, существующим на данной платформе (будь то интерфейс POSIX C или нечто иное) и тем самым предоставляют переносимый интерфейс для всех поддерживаемых платформ.

Для тех, кто уже имеет опыт написания многопоточных приложений и хотел бы воспользоваться им для разработки программ на C++ с применением новых возможностей, в этом приложении проводится сравнение средств, имеющихся в Java, POSIX C, C++ с библиотекой Boost Thread Library и C++11. Даются также перекрестные ссылки на соответствующие главы этой книги.

Средство	**Java**	**Posix C**
Запуск потоков	Класс `java.lang.thread`	Тип `pthread_t` и соответствующие функции API: `pthread_create()`, `pthread_detach()`, `pthread_join()`
Взаимное исключение	Блоки `synchronized`	Тип `pthread_mutex_t` и соответствующие функции API: `pthread_mutex_lock()` `pthread_mutex_unlock()` и другие
Ожидание предиката	Методы `wait()` и `notify()` класса `java.lang.Object`, используемые внутри блоков `synchronized`	Тип `pthread_cond_t` и соответствующие функции API: `pthread_cond_wait()`, `pthread_cond_timed_wait(` и другие
Атомарные операции и модель памяти с учетом параллелизма	`volatile`-переменные, типы в пакете `java.util.concurrent.atomic`	Отсутствует
Потокобезопасные контейнеры	Контейнеры в пакете `java.util.concurrent`.	Отсутствует

Будущие результаты	Интерфейс `java.util.concurrent.future` и ассоциированные с ним классы	Отсутствует
Пулы потоков	Класс `java.util.concurrent.ThreadPoolExecutor`	Отсутствует
Прерывание потока	Метод `interrupt()` класса `java.lang.Thread`	`pthread_cancel()`

Приложение C.
Каркас передачи сообщений и полный пример программы банкомата

В разделе 4.1 мы познакомились с каркасом передачи сообщений между потоками, продемонстрировав его на примере программы банкомата. В этом приложении приводится полный код примера, включая и код каркаса передачи сообщений.

В листинге C.1 показан код очереди сообщений. Сообщения хранятся в списке и представлены указателями на базовый класс. Сообщения конкретного типа обрабатываются шаблонным классом, производным от этого базового класса. В момент помещения сообщения в очередь конструируется подходящий экземпляр обертывающего класса и сохраняется указатель на него; операция извлечения возвращает именно этот указатель. Поскольку в классе `message_base` нет функций-членов, извлекающий поток должен привести указатель к нужному типу `wrapped_message<T>`, прежде чем сможет получить хранящееся сообщение.

Листинг C.1. Простая очередь сообщений

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

namespace messaging
{                                    | Базовый класс
struct message_base {↵               | элементов очереди
 virtual ~message_base() {}
};

template <typename Msg>              | Для каждого типа сообщений
struct wrapped_message:↵             | имеется специализация
message_base {
 Msg contents;
 explicit wrapped_message(Msg const& contents_):
  contents(contents_) {}
};

                                     | Наша очередь
class queue↵                         | сообщений
{                                                    | В настоящей
 std::mutex m;                                       | очереди хранят-
 std::condition_variable c;                          | ся указатели на
 std::queue<std::shared_ptr<message_base> > q;↵      | message_base

public:
 template<typename T>                                | Обернуть добав-
```

```
 void push(T const& msg)                    | ленное сообще-
 {                                          | ние и сохранить
  std::lock_guard<std::mutex> lk(m);        | указатель
  q.push( ↵┘
   std::make_shared<wrapped_message<T> >(msg));
  c.notify_all();
 }

 std::shared_ptr<message_base> wait_and_pop()| Блокирует до
 {                                           | появления в
  std::unique_lock<std::mutex> lk(m);        | очереди хотя бы
  c.wait(lk, [&]{ return !q.empty(); });  ↵┘ одного элемента
  auto res = q.front();
  q.pop();
  return res;
 }
};
}
```

Отправкой сообщений занимается объект класса `sender`, показанного в листинге С.2. Это не более чем тонкая обертка вокруг очереди сообщений, которая позволяет только добавлять сообщения. При копировании экземпляров `sender` копируется только указатель на очередь, а не сама очередь.

Листинг С.2. Класс `sender`

```
namespace messaging {
 class sender {| sender обертывает указатель
 queue* q;    ↵┘ на очередь

public:       | У сконструированного по умолчанию
 sender() :↵┘ sender'а нет очереди
 q(nullptr) {}


                                  | Разрешаем конструирование
 explicit sender(queue* q_):↵┘ из указателя на очередь
  q(q_) {}

 template<typename Message>
 void send(Message const& msg) {
  if (q){          | Отправка сообщения сводится
   q->push(msg);↵┘ к помещению его в очередь
  }
 }
};
}
```

Получение сообщений несколько сложнее. Мы не только должны дождаться появления сообщения в очереди, но еще и проверить, совпадает ли его тип с одним из известных нам типов, и вызвать соответствующий обработчик. Эта процедура начинается в классе `receiver`,

показанном в листинге ниже.

Листинг С.3. Класс `receiver`

```
namespace messaging {
class receiver {
 queue q; ← receiver владеет очередью

public:                    | Разрешить неявное преобразование в объект
 operator sender() {↵      | sender, ссылающийся на эту очередь
  return sender(&q);
 }

                           | При обращении к функции ожидания
 dispatcher wait() {↵      | очереди создается диспетчер
  return dispatcher(&q);
 }
};
}
```

Если `sender` только ссылается на очередь сообщений, то `receiver` ей владеет. Мы можем получить объект `sender`, ссылающийся на очередь, воспользовавшись неявным преобразованием. Процедура диспетчеризации сообщения начинается с обращения к функции `wait()`. При этом создается объект `dispatcher`, ссылающийся на очередь, которой владеет `receiver`. Класс `dispatcher` показан в следующем листинге; как видите, содержательная работа производится в его *деструкторе*. В данном случае работа состоит в ожидании сообщения и его диспетчеризации.

Листинг С.4. Класс `dispatcher`

```
namespace messaging {
class close_queue {}; ← Сообщение о закрытии очереди

class dispatcher {
 queue* q;                                  | Экземпляры
 bool chained;                              | диспетчера нельзя
                                            | копировать
 dispatcher(dispatcher const&)=delete;↵
 dispatcher& operator=(dispatcher const&)=delete;
 template<
  typename Dispatcher,| Разрешить экземплярам
  typename Msg,       | TemplateDispatcher доступ
  typename Func>    ↵ | к закрытым частям класса
 friend class TemplateDispatcher;
 void wait_and_dispatch()
 {          (1) В цикле ждем и диспетчеризуем
  for (;;) {↵ сообщения
   auto msg = q->wait_and_pop();
   dispatch(msg);
  }
 }                  (2) dispatch() смотрит, не пришло ли
```

```
                    | сообщение close_queue, и, если
 bool dispatch (←┘ да, возбуждает исключение
  std::shared_ptr<message_base> const& msg) {
  if (dynamic_cast<wrapped_message<close_queue>*>(msg.get())) {
   throw close_queue();
  }
  return false;
 }

public:                              | Экземпляры диспетчера
 dispatcher(dispatcher&& other):←┘ можно перемещать
  q(other.q), chained(other.chained) {| Объект-источник не должен
  other.chained = true;               ←┘ ждать сообщений
 }

 explicit dispatcher(queue* q_): q(q_), chained(false) {}

 template<typename Message, typename Func>
 TemplateDispatcher<dispatcher, Message, Func>
 handle(Func&& f)←┐ Сообщения конкретного типа
 {                 (3) обрабатывает TemplateDispatcher
  return TemplateDispatcher<dispatcher, Message, Func>(
   q, this, std::forward<Func>(f));
 }

 ~dispatcher() noexcept(false)←┐ Деструктор может
 {                               (4) возбудить исключение
  if (!chained) {
   wait_and_dispatch();
  }
 }
};
}
```

Экземпляр `dispatcher`, возвращенный функцией `wait()`, немедленно уничтожается, так как является временным объектом, и, как уже было сказало, вся работа выполняется в его деструкторе. Деструктор вызывает функцию `wait_and_dispatch()`, которая в цикле **(1)** ожидает сообщения и передает его функции `dispatch()`. Сама функция `dispatch()` **(2)** проста, как правда: она проверяет, не получено ли сообщение типа `close_queue`, и, если так, то возбуждает исключение; в противном случае возвращает `false`, извещая, что сообщение не обработало. Именно из-за исключения `close_queue` деструктор и помечен как `noexcept(false)` **(4)**; без этой аннотации действовала бы подразумеваемая спецификация исключений для деструктора — `noexcept(true)`, означающая, что исключения не допускаются, и тогда исключение `close_queue` привело бы к завершению программы.

Но просто вызывать функцию `wait()` особого смысла не имеет — как правило, нам нужно обработать полученное сообщение. Для этого предназначена функция-член `handle()` **(3)**. Это шаблон, и тип сообщения нельзя вывести, поэтому необходимо явно указать, сообщение какого типа обрабатывается, и передать функцию (или допускающий вызов объект) для его обработки. Сама функция `handle()` передает очередь, текущий объект

dispatcher и функцию-обработчик новому экземпляру шаблонного класса TemplateDispatcher, который обрабатывает сообщения указанного типа. Код этого класса показан в листинге С.5. Именно поэтому мы проверяем флаг chained в деструкторе перед тем, как приступить к ожиданию сообщения; он не только предотвращает ожидание объектами, содержимое которых перемещено, но и позволяет передать ответственность за ожидание новому экземпляру TemplateDispatcher.

Листинг С.5. Шаблон класса TemplateDispatcher

```
namespace messaging {
template<
 typename PreviousDispatcher, typename Msg, typename Func>
class TemplateDispatcher {
 queue* q;
 PreviousDispatcher* prev;
 Func f;
 bool chained;

 TemplateDispatcher(TemplateDispatcher const&) = delete;
 TemplateDispatcher& operator=(
  TemplateDispatcher const&) = delete;

 template<
  typename Dispatcher, typename OtherMsg, typename OtherFunc>
 friend class TemplateDispatcher;
 void wait_and_dispatch()
 {
  for (;;) {
   auto msg = q->wait_and_pop();
   if (dispatch(msg))
   break;
  }
 }

 bool dispatch(std::shared_ptr<message_base> const& msg) {
  if (wrapped_message<Msg>* wrapper =
   dynamic_cast<wrapped_message<Msg>*>(
   msg.get())) {
   f(wrapper->contents);
   return true;
  }
  else {
   return prev->dispatch(msg);
  }
 }

public:
 TemplateDispatcher(TemplateDispatcher&& other):
  q(other.q), prev(other.prev), f(std::move(other.f)),
  chained(other.chained) {
  other.chained = true;
```

Все конкретизации TemplateDispatcher дружат между собой

Если мы обработали сообщение выходим (1) из цикла

Проверяем тип сообщения и вызываем (2) функцию

Вызываем предыдущий (3) диспетчер в цепочке

```
  }

  TemplateDispatcher(
   queue* q_, PreviousDispatcher* prev_, Func&& f_):
   q(q_), prev(prev_), f(std::forward<Func>(f_)), chained(false)
  {
   prev_->chained = true;
  }

  template<typename OtherMsg, typename OtherFunc>
  TemplateDispatcher<TemplateDispatcher, OtherMsg, OtherFunc>
  handle(OtherFunc&& of)
  {
   return TemplateDispatcher<
    TemplateDispatcher, OtherMsg, OtherFunc>(
     q, this, std::forward<OtherFunc>(of));
  }

  ~TemplateDispatcher() noexcept(false)
  {
   if (!chained) {
    wait_and_dispatch();
   }
  }
};
}
```

Дополнительные обработчики
(4) можно связать в цепочку

Деструктор снова помечен как
(5) noexcept(false)

Шаблон класса `TemplateDispatcher<>` устроен по образцу класса `dispatcher` и почти ничем не отличается от него. В частности, деструктор тоже вызывает `wait_and_dispatch()`, чтобы дождаться сообщения.

Поскольку мы не возбуждаем исключения, если сообщение обработало, то теперь в цикле **(1)** нужно проверять, обработали мы сообщение или нет. Обработка прекращается, как только сообщение успешно обработало, чтобы в следующий раз можно было ждать очередного набора сообщений. Если найдено соответствие указанному типу сообщения, то вызывается предоставленная функция **(2)**, а не возбуждается исключение (хотя функция-обработчик может и сама возбудить исключение). Если же соответствие не найдено, то мы передаем сообщение предыдущему диспетчеру в цепочке **(3)**. В самом первом экземпляре это будет объект `dispatcher`, но если в функции `handle()` **(4)** вызовы сцеплялись, чтобы можно было обработать несколько типов сообщений, то предыдущим диспетчером может быть ранее созданный экземпляр `TemplateDispatcher<>`, который в свою очередь передаст сообщение предшествующему ему диспетчеру в цепочке, если не сможет обработать его сам. Поскольку любой обработчик может возбудить исключение (в том числе и обработчик самого первого объекта `dispatcher`, если встретит сообщение `close_queue`), то деструктор снова необходимо снабдить аннотацией `noexcept(false)` **(5)**.

Этот простенький каркас позволяет помещать в очередь сообщения любого типа, а затем на принимающем конце отбирать те из них, которые мы можем обработать. Кроме того, он позволяет передавать ссылку на очередь, чтобы в нее можно было добавлять новые сообщения, оставляя при этом прижимающий конец недоступным извне.

И чтобы закончить пример из главы 4, в листинге С.6 приведён код сообщений, в

листингах C.7, C.8 и C.9 — различные конечные автоматы, а в листинге C.10 — управляющая программа.

Листинг C.6. Сообщения банкомата

```
struct withdraw {
 std::string account;
 unsigned amount;
 mutable messaging::sender atm_queue;

 withdraw(std::string const& account_,
  unsigned amount_, messaging::sender atm_queue_):
   account(account_), amount(amount_), atm_queue(atm_queue_) {}
};

struct withdraw_ok {};

struct withdraw_denied {};

struct cancel_withdrawal {
 std::string account;
 unsigned amount;

 cancel_withdrawal(std::string const& account_,
  unsigned amount_):
   account(account_), amount(amount_) {}
};

struct withdrawal_processed {
 std::string account;
 unsigned amount;

 withdrawal_processed(std::string const& account_,
  unsigned amount_):
   account(account_), amount(amount_) {}
};

struct card_inserted {
 std::string account;

 explicit card_inserted(std::string const& account_):
  account(account_) {}
};

struct digit_pressed {
 char digit;

 explicit digit_pressed(char digit_):
  digit(digit_) {}
};

struct clear_last_pressed {};

struct eject_card {};
```

```
struct withdraw_pressed {
 unsigned amount;

 explicit withdraw_pressed(unsigned amount_):
  amount(amount_) {}

};

struct cancel_pressed {};

struct issue_money {
 unsigned amount;
 issue_money(unsigned amount_):
  amount(amount_) {}
};

struct verify_pin {
 std::string account;
 std::string pin;
 mutable messaging::sender atm_queue;

 verify_pin(std::string const& account_, std::string const& pin_,
  messaging::sender atm_queue_):
   account(account_), pin(pin_), atm_queue(atm_queue_) {}
};

struct pin_verified {};

struct pin_incorrect {};

struct display_enter_pin {};

struct display_enter_card {};

struct display_insufficient_funds {};

struct display_withdrawal_cancelled {};

struct display_pin_incorrect_message {};

struct display_withdrawal_options (};

struct get_balance {
 std::string account;
 mutable messaging::sender atm_queue;

 get_balance(
  std::string const& account_, messaging::sender atm_queue_):
   account(account_), atm_queue(atm_queue_) {}
};

struct balance {
```

```
 unsigned amount;

 explicit balance(unsigned amount_):
  amount(amount_) {}
};

struct display_balance {
 unsigned amount;

 explicit display_balance(unsigned amount_):
  amount(amount_) {}
};

struct balance_pressed {};
```

Листинг С.7. Конечный автомат банкомата

```
class atm {
 messaging::receiver incoming;
 messaging::sender bank;
 messaging::sender interface_hardware;
 void (atm::*state)();
 std::string account;
 unsigned withdrawal_amount;
 std::string pin;

 void process_withdrawal() {
  incoming.wait().handle<withdraw_ok>(
   [&](withdraw_ok const& msg) {
    interface_hardware.send(
     issue_money(withdrawal_amount));
    bank.send(
     withdrawal_processed(account, withdrawal_amount));
    state = &atm::done_processing;
   }
  ).handle<withdraw_denied>(
   [&](withdraw_denied const& msg) {
    interface_hardware.send(display_insufficient_funds());
    state = &atm::done_processing;
   }
  ).handle<cancel_pressed>(
   [&](cancel_pressed const& msg) {
    bank.send(
     cancel_withdrawal(account, withdrawal_amount));
    interface_hardware.send(
     display_withdrawal_cancelled());
    state = &atm::done_processing;
   }
  );
 }

 void process_balance() {
  incoming.wait().handle<balance>(
   [&](balance const& msg) {
    interface_hardware.send(display_balance(msg.amount));
    state = &atm::wait_for_action;
```

```
  }
 ).handle<cancel_pressed>(
  [&](cancel_pressed const& msg) {
   state = &atm::done_processing;
  }
 );
}

void wait_for_action() {
 interface_hardware.send(display_withdrawal_options());
 incoming.wait().handle<withdraw_pressed>(
  [&](withdraw_pressed const& msg) {
   withdrawal_amount = msg.amount;
   bank.send(withdraw(account, msg.amount, incoming));
   state = &atm::process_withdrawal;
  }
 ).handle<balance_pressed>(
  [&](balance_pressed const& msg) {
   bank.send(get_balance(account, incoming));
   state = &atm::process_balance;
  }
 ).handle<cancel_pressed>(
  [&](cancel_pressed const& msg) {
   state = &atm::done_processing;
  }
 );
}

void verifying_pin() {
 incoming.wait().handle<pin_verified>(
  [&](pin_verified const& msg) {
   state = &atm::wait_for_action;
  }
 ).handle<pin_incorrect>(
  [&](pin_incorrect const& msg) {
   interface_hardware.send(
    display_pin_incorrect_message());
   state = &atm::done_processing;
  }
 ).handle<cancel_pressed>(
  [&](cancel_pressed const& msg) {
   state = &atm::done_processing;
  }
 );
}

void getting_pin() {
 incoming.wait().handle<digit_pressed>(
  [&](digit_pressed const& msg) {
   unsigned const pin_length = 4;
   pin += msg.digit;
   if (pin.length() == pin_length) {
    bank.send(verify_pin(account, pin, incoming));
    state = &atm::verifying_pin;
   }
  }
 ).handle<clear_last_pressed>(
```

```
    [&](clear_last_pressed const& msg) {
     if (!pin.empty()) {
      pin.pop_back();
     }
    }
   ).handle<cancel_pressed>(
    [&](cancel_pressed const& msg) {
     state = &atm::done_processing;
    }
   );
  }

  void waiting_for_card() {
   interface_hardware.send(display_enter_card());
   incoming.wait().handle<card_inserted>(
    [&](card_inserted const& msg) {
     account = msg.account;
     pin = "";
     interface_hardware.send(display_enter_pin());
     state = &atm::getting_pin;
    }
   );
  }

  void done_processing() {
   interface_hardware.send(eject_card());
   state = &atm::waiting_for_card;
  }

  atm(atm const&) = delete;
  atm& operator=(atm const&) = delete;

 public:
  atm(messaging::sender bank_,
   messaging::sender interface_hardware_):
    bank(bank_), interface_hardware(interface_hardware_) {}

  void done() {
   get_sender().send(messaging::close_queue());
  }

  void run() {
   state = &atm::waiting_for_card;
   try {
    for (;;) {
     (this->*state)();
    }
   } catch(messaging::close_queue const&) {
   }
  }

  messaging::sender get_sender() {
   return incoming;
  }
 };
```

Листинг С.8. Конечный автомат банка

```
class bank_machine {
 messaging::receiver incoming;
 unsigned balance;

public:
 bank_machine():
  balance(199) {}

 void done() {
  get_sender().send(messaging::close_queue());
 }

 void run() {
  try {
   for (;;) {
    incoming.wait().handle<verify_pin>(
     [&](verify_pin const& msg) {
      if (msg.pin == "1937") {
       msg.atm_queue.send(pin_verified());
      } else {
       msg.atm_queue.send(pin_incorrect());
      }
     }
    ).handle<withdraw>(
     [&](withdraw const& msg) {
      if (balance >= msg.amount) {
       msg.atm_queue.send(withdraw_ok());
       balance -= msg.amount;
      } else {
       msg.atm_queue.send(withdraw_denied());
      }
     }
    ).handle<get_balance>(
     [&](get_balance const& msg) {
      msg.atm_queue.send(::balance(balance));
     }
    ).handle<withdrawal_processed>(
     [&](withdrawal_processed const& msg) {
     }
    ).handle<cancel_withdrawal>(
     [&](cancel_withdrawal const& msg) {
     }
    );
   }
  } catch(messaging::close_queue const&) {
  }
 }

 messaging::sender get_sender() {
  return incoming;
 }
};
```

Листинг С.9. Конечный автомат пользовательского интерфейса

```
class interface_machine {
 messaging::receiver incoming;

public:
 void done() {
  get_sender().send(messaging::close_queue());
 }

 void run() {
  try {
   for (;;) {
    incoming.wait().handle<issue_money> (
     [&](issue_money const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout << "Issuing "
                 << msg.amount << std::endl;
      }
     }
    ).handle<display_insufficient_funds>(
     [&](display_insufficient_funds const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout << "Insufficient funds" << std::endl;
      }
     }
    ).handle<display_enter_pin>(
     [&](display_enter_pin const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout
        << "Please enter your PIN (0-9)" << std::endl;
      }
     }
    ).handle<display_enter_card>(
     [&](display_enter_card const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout << "Please enter your card (I)"
                 << std::endl;
      }
     }
    ).handle<display_balance>(
     [&](display_balance const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout
        << "The balance of your account is "
        << msg.amount << std::endl;
      }
     }
    ).handle<display_withdrawal_options>(
     [&](display_withdrawal_options const& msg) {
      {
       std::lock_guard<std::mutex> lk(iom);
       std::cout << "Withdraw 50? (w)" << std::endl;
```

```
        std::cout << "Display Balance? (b)"
                  << std::endl;
        std::cout << "Cancel? (c) " << std::endl;
       }
      }
     ).handle<display_withdrawal_cancelled>(
      [&](display_withdrawal_cancelled const& msg) {
       {
        std::lock_guard<std::mutex> lk(iom);
        std::cout << "Withdrawal cancelled"
                  << std::endl;
       }
      }
     ).handle<display_pin_incorrect_message>(
      [&](display_pin_incorrect_message const& msg) {
       {
        std::lock_guard<std::mutex> lk(iom);
        std::cout << "PIN incorrect" << std::endl;
       }
      }
     ).handle<eject_card>(
      [&](eject_card const& msg) {
       {
        std::lock_guard<std::mutex> lk(iom);
        std::cout << "Ejecting card" << std::endl;
       }
      }
     );
    }
   } catch (messaging::close_queue&) {
   }
  }

  messaging::sender get_sender() {
   return incoming;
  }
};
```

Листинг С.10. Управляющая программа

```
int main() {
 bank_machine bank;
 interface_machine interface_hardware;

 atm machine(bank.get_sender(), interface_hardware.get_sender());

 std::thread bank_thread(&bank_machine::run, &bank);
 std::thread if_thread(&interface_machine::run,
  &interface_hardware);
 std::thread atm_thread(&atm::run, &machine);

 messaging::sender atmqueue(machine.get_sender());
 bool quit_pressed = false;

 while (!quit_pressed) {
  char c = getchar();
  switch(c) {
```

```
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9':
    atmqueue.send(digit_pressed(c));
    break;
   case 'b':
    atmqueue.send(balance_pressed());
    break;
   case 'w':
    atmqueue.send(withdraw_pressed(50));
    break;
   case 'c':
    atmqueue.send(cancel_pressed());
    break;
   case 'q':
    quit_pressed = true;
    break;
   case 'i':
    atmqueue.send(card_inserted("acc1234"));
    break;
   }
  }
  bank.done();
  machine.done();
  interface_hardware.done();
  atm_thread.join();
  bank_thread.join();
  if_thread.join();
 }
```

Приложение D
Справочник по библиотеке C++ Thread Library

D.1. Заголовок <chrono>

В заголовке <chrono> объявлены классы для представления моментов времени, интервалов и часов, которые служат источником объектов time_point. В каждом классе часов имеется статическая переменная-член is_steady, показывающая, являются ли данные часы *стабильными*. Стабильными называются часы, которые ходят с постоянной частотой и не допускают подведения. Единственные гарантированно стабильные часы представлены классом std::chrono::steady_clock.

Содержимое заголовка

```
namespace std {

namespace chrono {

template<typename Rep, typename Period = ratio<1>>
class duration;

template<
 typename Clock,
 typename Duration = typename Clock::duration>
class time_point;

class system_clock;
class steady_clock;
typedef unspecified-clock-type high_resolution_clock;

}

}
```

D.1.1. Шаблон класса std::chrono::duration

Шаблон класса Std::chrono::duration предназначен для представления интервалов. Параметры шаблона Rep и Period — это соответственно тип данных для хранения значения интервала и конкретизация шаблона класса std::ratio, которая задает промежуток времени (в виде долей секунды) между последовательными «тиками». Например, std::chrono::duration<int, std::milli> определяет количество миллисекунд, представимое значением типа int, Std::chrono::duration<short, std::ratio<1, 50>> — количество пятидесятых долей секунды, представимое значением типа short, а std::chrono::duration<long long, std::ratio<60, 1>> — количество минут, представимое значением типа long long.

Определение класса

```
template <class Rep, class Period = ratio<1> >
class duration {
public:
 typedef Rep rep;
 typedef Period period;

 constexpr duration() = default;
```

```
 ~duration() = default;

 duration(const duration&) = default;
 duration& operator=(const duration&) = default;

 template <class Rep2>
 constexpr explicit duration(const Rep2& r);

 template <class Rep2, class Period2>
 constexpr duration(const duration<Rep2, Period2>& d);

 constexpr rep count() const;
 constexpr duration operator+() const;
 constexpr duration operator-() const;
 duration& operator++();
 duration operator++(int);
 duration& operator--();
 duration operator--(int);
 duration& operator+=(const duration& d);
 duration& operator-=(const duration& d);
 duration& operator*=(const rep& rhs);
 duration& operator/=(const rep& rhs);
 duration& operator%=(const rep& rhs);
 duration& operator%=(const duration& rhs);
 static constexpr duration zero();
 static constexpr duration min();
 static constexpr duration max();
};

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator !=(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
 const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
```

```
constexpr bool operator>=(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);

template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(
 const duration<Rep, Period>& d);
```

Требования

`Rep` должен быть встроенным числовым типом или определенным пользователем типом со свойствами числа. `Period` должен быть конкретизацией шаблона `std::ratio<>`.

STD::CHRONO::DURATION::REP, TYPEDEF

Это псевдоним типа для хранения числа тиков в значении `duration`.

Объявление

```
typedef Rep rep;
```

`rep` — тип значения, используемого для хранения внутреннего представления объекта `duration`.

STD::CHRONO::DURATION::PERIOD, TYPEDEF

Это псевдоним типа для конкретизации шаблона класса `std::ratio`, которая задает количество долей секунды, представляемых счетчиком интервала. Например, если `period` — это `std::ratio<1, 50>`, то объект `duration`, для которого `count()` равно *N*, представляет *N* пятидесятых долей секунды.

Объявление

```
typedef Period period;
```

STD::CHRONO::DURATION , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует экземпляр `std::chrono::duration` со значением по умолчанию.

Объявление

```
constexpr duration() = default;
```

Результат

Внутреннее значение `duration` (типа `rep`) инициализируется значением по умолчанию.

STD::CHRONO::DURATION , КОНВЕРТИРУЮЩИЙ КОНСТРУКТОР ИЗ ЗНАЧЕНИЯ СЧЕТЧИКА

Конструирует экземпляр `std::chrono::duration` с заданным значением счетчика.

Объявление

```
template <class Rep2>
constexpr explicit duration(const Rep2& r);
```

Результат

Внутреннее значение объекта `duration` инициализируется значением `static_cast<rep>(r)`.

Требования

Этот конструктор участвует в разрешении перегрузки, только если `Rep2` может быть неявно преобразован в `Rep`, и либо `Rep` — тип с плавающей точкой, либо `Rep2` *не* является типом с плавающей точкой.

Постусловие

```
this->count() == static_cast<rep>(r)
```

STD::CHRONO::DURATION , КОНВЕРТИРУЮЩИЙ КОНСТРУКТОР ИЗ ДРУГОГО ЗНАЧЕНИЯ STD::CHRONO::DURATION

Конструирует экземпляр `std::chrono::duration`, масштабируя значение счетчика другого объекта `std::chrono::duration`.

Объявление

```
template <class Rep2 , class Period2>
constexpr duration(const duration<Rep2, Period2>& d);
```

Результат

Внутреннее значение объекта `duration` инициализируется значением `duration_cast<duration<Rep, Period>>(d).count()`.

Требования

Этот конструктор участвует в разрешении перегрузки, только если `Rep` — тип с плавающей точкой, либо `Rep2` *не* является типом с плавающей точкой, и `Period2` — целое кратное `Period` (то есть `ratio_divide<Period2, Period>::den == 1`). Это позволяет избежать случайного обрезания (и, значит, потери точности) при сохранении интервала с меньшим периодом в переменной, представляющий интервал с большим периодом.

Постусловие

```
this->count() == duration_cast<duration<Rep, Period>>(d).count()
```

Примеры

```
duration< int, ratio<1, 1000>> ms(5); ←
```
5 миллисекунд

```
duration<int, ratio<1,1>> s(ms);←┐
```
Ошибка: нельзя
| сохранить мс как
| целые секунды

```
duration<double, ratio<1, 1>> s2(ms);←┐
```
Правильно:
| s2.count() == 0.005

```
duration<int, ratio<1, 1000000>> us(ms);←┐
```
Правильно:
| us.count() == 5000

STD::CHRONO::DURATION::COUNT , ФУНКЦИЯ-ЧЛЕН

Получает значение интервала.

Объявление

```
constexpr rep count() const;
```

Возвращаемое значение

Внутреннее значение объекта `duration` в виде значения типа `rep`.

STD::CHRONO::DURATION::OPERATOR+ , УНАРНЫЙ ОПЕРАТОР ПЛЮС

Пустая операция, возвращает копию `*this`.

Объявление

```
constexpr duration operator+() const;
```

Возвращаемое значение

```
*this
```

STD::CHRONO::DURATION::OPERATOR- , УНАРНЫЙ ОПЕРАТОР МИНУС

Возвращает интервал, в котором значение `count()` противоположно значению `this->count()`.

Объявление

```
constexpr duration operator-() const;
```

Возвращаемое значение

```
duration(-this->count());
```

STD::CHRONO::DURATION::OPERATOR++ , ОПЕРАТОР ПРЕДИНКРЕМЕНТА

Инкрементирует внутренний счетчик.

Объявление

```
duration& operator++();
```

Результат

```
++this->internal_count;
```

Возвращаемое значение

```
*this
```

STD::CHRONO::DURATION::OPERATOR++ , ОПЕРАТОР ПОСТИНКРЕМЕНТА

Инкрементирует внутренний счетчик и возвращает то значение `*this`, которое предшествовало выполнению операции.

Объявление

```
duration operator++(int);
```

Результат

```
duration temp(*this);
++(*this);
return temp;
```

STD::CHRONO::DURATION::OPERATOR-- , ОПЕРАТОР ПРЕДЕКРЕМЕНТА

Декрементирует внутренний счетчик.

Объявление

```
duration& operator--();
```

Результат

```
--this->internal_count;
```

Возвращаемое значение

```
*this
```

STD::CHRONO::DURATION::OPERATOR-- , ОПЕРАТОР ПОСТДЕКРЕМЕНТА

Декрементирует внутренний счетчик и возвращает то значение `*this`, которое предшествовало выполнению операции.

Объявление

```
duration operator--(int);
```

Результат

```
duration temp(*this);
--(*this);
return temp;
```

STD::CHRONO::DURATION::OPERATOR+= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Прибавляет счетчик другого объекта `duration` к внутреннему счетчику `*this`.

Объявление

```
duration& operator+=(duration const& other);
```

Результат

```
internal_count += other.count();
```

Возвращаемое значение

```
*this
```

STD::CHRONO::DURATION::OPERATOR-= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Вычитает счетчик другого объекта `duration` из внутреннего счетчика `*this`.

Объявление

```
duration& operator-=(duration const& other);
```

Результат

```
internal_count-=other.count();
```

Возвращаемое значение

```
*this
```

STD::CHRONO::DURATION::OPERATOR*= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Умножает внутренний счетчик `*this` на заданное значение.

Объявление

```
duration& operator*=(rep const& rhs);
```
Результат
```
internal_count*=rhs;
```
Возвращаемое значение
```
*this
```
STD::CHRONO::DURATION::OPERATOR/= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Делит внутренний счетчик `*this` на заданное значение.

Объявление
```
duration& operator/=(rep const& rhs);
```
Результат
```
internal_count/=rhs;
```
Возвращаемое значение
```
*this
```
STD::CHRONO::DURATION::OPERATOR%= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Записывает во внутренний счетчик `*this` остаток от его деления на заданное значение.

Объявление
```
duration& operator%=(rep const& rhs);
```
Результат
```
internal_count%=rhs;
```
Возвращаемое значение
```
*this
```
STD::CHRONO::DURATION::OPERATOR%= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Записывает во внутренний счетчик `*this` остаток от его деления на счетчик в другом объекте `duration`.

Объявление
```
duration& operator%=(duration const& rhs);
```
Результат
```
internal_count %= rhs.count();
```
Возвращаемое значение
```
*this
```
STD::CHRONO::DURATION::ZERO , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Возвращает объект `duration`, представляющий значение нуль.

Объявление
```
constexpr duration zero();
```
Возвращаемое значение
```
duration(duration_values<rep>::zero());
```
STD::CHRONO::DURATION::MIN , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Возвращает объект `duration`, представляющий минимально возможное для данной конкретизации значение.

Объявление
```
constexpr duration min();
```
Возвращаемое значение
```
duration(duration_values<rep>::min());
```
STD::CHRONO::DURATION::MAX , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Возвращает объект `duration`, представляющий максимально возможное для данной конкретизации значение.

Объявление
```
constexpr duration max();
```

Возвращаемое значение

```
duration(duration_values<rep>::max());
```

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ НА РАВЕНСТВО

Сравнивает два объекта `duration` на равенство, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразована в другую или они являются различными представлениями `duration`, но каждая может быть неявно преобразована в другую, то выражение построено некорректно.

Результат

Если `CommonDuration` — синоним `std::common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type`, то `lhs==rhs` возвращает `CommonDuration(lhs).count() == CommonDuration(rhs).count()`.

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ НА НЕРАВЕНСТВО

Сравнивает два объекта `duration` на неравенство, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразовала в другую или они являются различными представлениями `duration`, но каждая может быть неявно преобразовала в другую, то выражение построено некорректно.

Возвращаемое значение

```
!(lhs == rhs)
```

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ МЕНЬШЕ

Проверяет, что один объект `duration` меньше другого, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразована в другую или они являются различными представлениями `duration`, по каждая может быть неявно преобразована в другую, то выражение построено некорректно.

Результат

Если `CommonDuration` — синоним `std::common_type< duration< Rep1, Period1>, duration< Rep2, Period2>>::type`, то `lhs<rhs` возвращает `CommonDuration(lhs).count() < CommonDuration(rhs).count()`.

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ БОЛЬШЕ

Проверяет, что один объект `duration` больше другого, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразована в другую или они являются различными представлениями `duration`, но каждая может быть неявно преобразована в другую, то выражение построено некорректно.

Возвращаемое значение

```
!((rhs<lhs)||(rhs==lhs))
```

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ МЕНЬШЕ ИЛИ РАВНО

Проверяет, что один объект `duration` меньше или равен другому, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразовала в другую или они являются различными представлениями `duration`, но каждая может быть неявно преобразована в другую, то выражение построено некорректно.

Возвращаемое значение

```
!(rhs>lhs)
```

STD::CHRONO::DURATION , ОПЕРАТОР СРАВНЕНИЯ БОЛЬШЕ ИЛИ РАВНО

Проверяет, что один объект `duration` больше или равен другому, даже если они имеют разные представления и (или) периоды.

Объявление

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
 const duration<Rep1, Period1>& lhs,
 const duration<Rep2, Period2>& rhs);
```

Требования

Либо для `lhs` определено неявное преобразование в `rhs`, либо наоборот. Если ни одна из частей не может быть неявно преобразована в другую или они являются различными представлениями `duration`, но каждая может быть неявно преобразована в другую, то выражение построено некорректно.

Возвращаемое значение

```
!(lhs<rhs)
```

STD::CHRONO::DURATION_CAST , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Явно преобразует объект `std::chrono::duration` в заданную конкретизацию `std::chrono::duration`.

Объявление

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(
 const duration<Rep, Period>& d);
```

Требования

`ToDuration` должен быть конкретизацией `std::chrono::duration`.

Возвращаемое значение

Значение `d`, преобразованное к типу интервала, заданного параметром `ToDuration`. При выполнении операции минимизируется потеря точности в результате преобразования интервалов с разными масштабами и типами представления.

D.1.2. Шаблон класса std::chrono::time_point

Шаблон класса `std::chrono::time_point` представляет момент времени, измеренный по конкретным часам. Задается в виде интервала, прошедшего с момента *эпохи* данных часов. Параметр шаблона `Clock` задает часы (у разных часов должны быть разные типы), а параметр `Duration` — тип для измерения интервала от эпохи, который должен быть конкретизацией шаблона `std::chrono::duration`. По умолчанию `Duration` совпадает с подразумеваемым типом интервала, определенным в `Clock`.

Определение класса

```
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
 typedef Clock clock;
 typedef Duration duration;
 typedef typename duration::rep rep;
 typedef typename duration::period period;

 time_point();
 explicit time_point(const duration& d);

 template <class Duration2>
 time_point(const time_point<clock, Duration2>& t);

 duration time_since_epoch() const;

 time_point& operator+=(const duration& d);
 time_point& operator-=(const duration& d);

 static constexpr time_point min();
 static constexpr time_point max();
};
```

STD::CHRONO::TIME_POINT , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `time_point`, представляющий эпоху часов `Clock`; внутренний интервал инициализируется значением `Duration::zero()`.

Объявление

```
time_point();
```

Постусловие

Для сконструированного по умолчанию объекта `tp` типа `time_point` имеет место равенство `tp.time_since_epoch() == tp::duration::zero()`.

STD::CHRONO::TIME_POINT , КОНСТРУКТОР ИЗ ИНТЕРВАЛА

Конструирует объект `time_point`, представляющий заданный интервал от эпохи часов `Clock`.

Объявление

```
explicit time_point(const duration& d);
```

Постусловие

Для объекта `tp` типа `time_point`, созданного конструктором `tp(d)` из некоторого интервала `d`, имеет место равенство `tp.time_since_epoch() == d`.

STD::CHRONO::TIME_POINT , КОНВЕРТИРУЮЩИЙ КОНСТРУКТОР

Конструирует объект `time_point` из другого объекта `time_point` с таким же типом `Clock`, по другим типом `Duration`.

Объявление

```
template <class Duration2>
time_point(const time_point<clock, Duration2>& t);
```

Требования

Для типа `Duration2` должно существовать неявное преобразование в тип `Duration`.

Результат

Эквивалентно выражению `time_point(t.time_since_epoch())`.

Значение, возвращенное функцией `t.time_since_epoch()` неявно преобразуется в объект типа `Duration`, который сохраняется в новом объекте типа `time_point`.

STD::CHRONO::TIME_POINT::TIME_SINCE_EPOCH , ФУНКЦИЯ-ЧЛЕН

Возвращает интервал от эпохи часов для данного объекта типа `time_point`.

Объявление

```
duration time_since_epoch() const;
```

Возвращаемое значение

Значение `duration`, хранящееся в `*this`.

STD::CHRONO::TIME_POINT::OPERATOR+= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Прибавляет указанный интервал `duration` к значению, хранящемуся в данном объекте `time_point`.

Объявление

```
time_point& operator+=(const duration& d);
```

Результат

Прибавляет `d` к внутреннему интервалу `*this`, эквивалентно `this->internal_duration += d`.

Возвращаемое значение

```
*this
```

STD::CHRONO::TIME_POINT::OPERATOR-= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Вычитает указанный интерфейс `duration` из значения, хранящегося в данном объекте `time_point`.

Объявление

```
time_point& operator-=(const duration& d);
```

Результат

Вычитает `d` из внутреннего интервала `*this`, эквивалентно `this->internal_duration -= d`.

Возвращаемое значение

`*this`

`STD::CHRONO::TIME_POINT::MIN` , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Получает объект `time_point`, представляющий минимально возможное для данного типа значение.

Объявление

```
static constexpr time_point min();
```

Возвращаемое значение

`time_point(time_point::duration::min())` (см. 11.1.1.15)

`STD::CHRONO::TIME_POINT::MAX`, СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Получает объект `time_point`, представляющий максимально возможное для данного типа значение.

Объявление

```
static constexpr time_point max();
```

Возвращаемое значение

`time_point(time_point::duration::max())` (см. 11.1.1.16)

D.1.3. Класс `std::chrono::system_clock`

Класс `std::chrono::system_clock` предоставляет средства для получения времени от системных часов реального времени. Текущее время возвращает функция `std::chrono::system_clock::now()`. Объекты класса `std::chrono::system_clock::time_point` можно преобразовывать в тип `time_t` с помощью функции `std::chrono::system_clock::to_time_t()` и получать из этого типа с помощью функции `std::chrono::system_clock::to_time_point()`. Системные часы не *стабильны*, поэтому последующее обращение к `std::chrono::system_clock::now()` может вернуть момент времени, более ранний, чем при предыдущем обращении (например, если часы операционной системы были подведены вручную или автоматически синхронизировались с внешним источником времени).

Определение класса

```
class system_clock {

public:
 typedef unspecified-integral-type rep;
 typedef std::ratio<unspecified, unspecified> period;
 typedef std::chrono::duration<rep, period> duration;
 typedef std::chrono::time_point<system_clock> time_point;
 static const bool is_steady = unspecified;

 static time_point now() noexcept;

 static time_t to_time_t(const time_point& t) noexcept;
 static time_point from_time_t(time_t t) noexcept;
```

```
};
```

STD::CHRONO::SYSTEM_CLOCK::REP , TYPEDEF

Псевдоним целочисленного типа, используемого для хранения количества тиков в интервале `duration`.

Объявление

```
typedef unspecified-integral-type rep;
```

STD::CHRONO::SYSTEM_CLOCK::PERIOD , TYPEDEF

Псевдоним типа для конкретизации шаблонного класса `std::ratio`, которая определяет наименьшее число секунд (или долей секунды) между различающимися значениями `duration` или `time_point`. Псевдоним `period` определяет *точность* часов, а не частоту тактов.

Объявление

```
typedef std::ratio<unspecified, unspecified> period;
```

STD::CHRONO::SYSTEM_CLOCK::DURATION , TYPEDEF

Конкретизация шаблонного класса `std::chrono::duration`, в которой может храниться разность между любыми двумя моментами времени, полученными от системных часов реального времени.

Объявление

```
typedef std::chrono::duration<
 std::chrono::system_clock::rep,
 std::chrono::system_clock::period> duration;
```

STD::CHRONO::SYSTEM_CLOCK::TIME_POINT , TYPEDEF

Конкретизация шаблонного класса `std::chrono::time_point`, в которой могут храниться моменты времени, полученные от системных часов реального времени.

Объявление

```
typedef std::chrono::time_point<std::chrono::system_clock> time_point;
```

STD::CHRONO::SYSTEM_CLOCK::NOW , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Получает текущее время от системных часов реального времени.

Объявление

```
time_point now() noexcept;
```

Возвращаемое значение

Экземпляр `time_point`, представляющий текущее время по системным часам реального времени.

Исключения

Возбуждает исключение `std::system_error` в случае ошибки.

`STD::CHRONO::SYSTEM_CLOCK::TO_TIME_T`, СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Преобразует объект типа `time_point` в объект типа `time_t`.

Объявление

```
time_t to_time_t(time_point const& t) noexcept;
```

Возвращаемое значение

Экземпляр `time_t`, представляющий тот же момент времени, что и `t`, округленный или обрезанный до секунд.

Исключения

Возбуждает исключение `std::system_error` в случае ошибки.

STD::CHRONO::SYSTEM_CLOCK::FROM_TIME_T , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Преобразует объект типа `time_t` в объект типа `time_point`.

Объявление

```
time_point from_time_t(time_t const& t) noexcept;
```

Возвращаемое значение

Экземпляр `time_point`, представляющий тот же момент времени, что и `t`.

Исключения

Возбуждает исключение `std::system_error` в случае ошибки.

D.1.4. Класс `std::chrono::steady_clock`

Класс `std::chrono::steady_clock` предоставляет доступ к системным стабильным часам. Текущее время возвращает функция `std::chrono::steady_clock::now()`. Не существует фиксированного соотношения между значениями, возвращаемыми `std::chrono::steady_clock::now()` и показаниями часов реального времени. Стабильные часы не могут «идти в обратную сторону», поэтому если некое обращение к функции `std::chrono::steady_clock::now()` происходит-раньше другого обращения к ней же, то второе обращение должно вернуть момент времени, больший или равным первому. Часы ходят с частотой, настолько близкой к постоянной, насколько это возможно.

Определение класса

```
class steady_clock {
public:
 typedef unspecified-integral-type rep;
 typedef std::ratio<
  unspecified, unspecified> period;
 typedef std::chrono::duration<rep, period> duration;
 typedef std::chrono::time_point<steady_clock>
 time_point;
 static const bool is_steady = true;

 static time_point now() noexcept;
};
```

`STD::CHRONO::STEADY_CLOCK::REP` , TYPEDEF

Псевдоним целочисленного типа, используемого для хранения количества тиков в интервале `duration`.

Объявление

```
typedef unspecified-integral-type rep;
```

`STD::CHRONO::STEADY_CLOCK::PERIOD` , TYPEDEF

Псевдоним типа для конкретизации шаблонного класса `std::ratio`, которая определяет наименьшее число секунд (или долей секунды) между различающимися значениями `duration` или `time_point`. Псевдоним `period` определяет *точность* часов, а не частоту тактов.

Объявление

```
typedef std::ratio<unspecified, unspecified> period;
```

`STD::CHRONO::STEADY_CLOCK::DURATION` , TYPEDEF

Конкретизация шаблонного класса `std::chrono::duration`, в которой может храниться разность между любыми двумя моментами времени, полученными от системных стабильных часов.

Объявление

```
typedef std::chrono::duration<
 std::chrono::steady_clock::rep,
```

```
 std::chrono::steady_clock::period> duration;
```

STD::CHRONO::STEADY_CLOCK::TIME_POINT , TYPEDEF

Конкретизация шаблонного класса `std::chrono::time_point`, в которой могут храниться моменты времени, полученные от системных стабильных часов.

Объявление

```
typedef std::chrono::time_point<std::chrono::steady_clock>
time_point;
```

STD::CHRONO::STEADY_CLOCK::NOW , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Получает текущее время от системных стабильных часов.

Объявление

```
time_point now() noexcept;
```

Возвращаемое значение

Экземпляр `time_point`, представляющий текущее время но системным стабильным часам.

Исключения

Возбуждает исключение `std::system_error` в случае ошибки.

Синхронизация

Если одно обращение к `std::chrono::steady_clock::now()` происходит-раньше другого, то момент времени `time_point`, возвращенный при первом обращении, меньше или равен моменту времени `time_point`, возвращенному при втором обращении.

D.1.5. Псевдоним типа `std::chrono::high_resolution_clock`

Класс `std::chrono::high_resolution_clock` предоставляет доступ к системным часам максимально высокого разрешения. Как и для всех остальных часов, текущее время можно получить от функции `std::chrono::high_resolution_clock::now()`. Имя `std::chrono::high_resolution_clock` может быть псевдонимом класса `std::chrono::system_clock` или класса `std::chrono::steady_clock`, либо отдельным типом.

Хотя тип `std::chrono::high_resolution_clock` дает самое высокое разрешение среди всех входящих в библиотеку часов, обращение к функции `std::chrono::high_resolution_clock::now()` все же занимает конечное время. Поэтому, пытаясь хронометрировать очень короткие операции, учитывайте накладные расходы на вызов этой функции.

Определение класса

```
class high_resolution_clock {
public:
 typedef unspecified-integral-type rep;
 typedef std::ratio<
  unspecified, unspecified> period;
 typedef std::chrono::duration<rep, period> duration;
 typedef std::chrono::time_point<
  unspecified> time_point;
 static const bool is_steady = unspecified;

 static time_point now() noexcept;
};
```

D.2. Заголовок <condition_variable>

Заголовок <condition_variable> предоставляет доступ к условным переменным. Это базовый механизм синхронизации, который позволяет блокировать поток до получения уведомления о том, что выполнено некоторое условие или истек таймаут.

Содержимое заголовка

```
namespace std {
enum class cv_status { timeout, no_timeout };

class condition_variable;
class condition_variable_any;
}
```

D.2.1. Класс std::condition_variable

Класс std::condition_variable позволяет потоку ждать выполнения условия.

Экземпляры этого класса не удовлетворяют концепциям CopyAssignable, CopyConstructible, MoveAssignable, MoveConstructible.

Определение класса

```
class condition_variable {
public:
 condition_variable();
 ~condition_variable();

 condition_variable(condition_variable const&) = delete;
 condition_variable& operator=(
  condition_variable const&) = delete;

 void notify_one() noexcept;
 void notify_all() noexcept;

 void wait(std::unique_lock<std::mutex>& lock);

 template <typename Predicate>
 void wait(std::unique_lock<std::mutex>& lock, Predicate pred);

 template <typename Clock, typename Duration>
 cv_status wait_until(
  std::unique_lock<std::mutex>& lock,
  const std::chrono::time_point<Clock, Duration>& absolute_time);

 template <typename Clock, typename Duration, typename Predicate>
 bool wait_until(
  std::unique_lock<std::mutex>& lock,
  const std::chrono::time_point<Clock, Duration>& absolute_time,
  Predicate pred);

 template <typename Rep, typename Period>
 cv_status wait_for(
  std::unique_lock<std::mutex>& lock,
  const std::chrono::duration<Rep, Period>& relative_time);
```

```
 template <typename Rep, typename Period, typename Predicate>
 bool wait_for(
  std::unique_lock<std::mutex>& lock,
  const std::chrono::duration<Rep, Period>& relative_time,
  Predicate pred);
};

void notify_all_at_thread_exit(
 condition_variable&, unique_lock<mutex>);
```

STD::CONDITION_VARIABLE , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект типа `std::condition_variable`.

Объявление

```
condition_variable();
```

Результат

Конструирует объект типа `std::condition_variable`.

Исключения

Исключение типа `std::system_error`, если сконструировать условную переменную не получилось.

STD::CONDITION_VARIABLE , ДЕСТРУКТОР

Уничтожает объект `std::condition_variable`.

Объявление

```
~condition_variable();
```

Предусловия

Не существует потоков, заблокированных по `*this` в обращениях к `wait()`, `wait_for()` или `wait_until()`.

Результат

Уничтожает `*this`.

Исключения

Нет.

STD::CONDITION_VARIABLE::NOTIFY_ONE , ФУНКЦИЯ-ЧЛЕН

Пробуждает один из потоков, ожидающих `std::condition_variable`.

Объявление

```
void notify_one() noexcept;
```

Результат

Пробуждает один из потоков, ожидающих `*this`, в точке вызова. Если таких потоков нет, функция не имеет никакого эффекта.

Исключения

Исключение типа `std::system_error`, если действие не выполнено.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

STD::CONDITION_VARIABLE::NOTIFY_ALL , ФУНКЦИЯ-ЧЛЕН

Пробуждает все потоки, ожидающие `std::condition_variable`.

Объявление

```
void notify_all() noexcept;
```

Результат

Пробуждает все потоки, ожидающие `*this`, в точке вызова. Если таких потоков нет, функция не имеет никакого эффекта.

Исключения

Исключение типа `std::system_error`, если действие не выполнено.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE::WAIT` , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная `std::condition_variable` не получит сигнал в результате обращения к `notify_one()` или `notify_all()` либо не произойдёт ложное пробуждение.

Объявление

```
void wait(std::unique_lock<std::mutex>& lock);
```

Предусловия

Значение `lock.owns_lock()` равно `true`, и блокировкой владеет вызывающий поток.

Результат

Атомарно разблокирует предоставленный объект `lock` и блокирует поток, пока он не будет разбужен обращением к `notify_one()` или `notify_all()` из другого потока либо не произойдёт ложное пробуждение. Перед возвратом управления из `wait()` объект `lock` снова блокируется.

Исключения

Исключение типа `std::system_error`, если действие не выполнено. Если объект `lock` был разблокирован при обращении к `wait()`, он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

> **Примечание**. Ложное пробуждение означает, что поток, вызвавший `wait()`, может быть разбужен, даже если ни один другой поток не обращался к `notify_one()` или `notify_all()`. Поэтому рекомендуется использовать перегруженный вариант `wait()`, который принимает предикат. Если это нежелательно, то рекомендуется вызывать `wait()` в цикле, где проверяется предикат, ассоциированный с условной переменной.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE::WAIT` , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная `std::condition_variable` не получит сигнал в результате обращения к `notify_one()` или `notify_all()` и при этом предикат равен `true`.

Объявление

```
template<typename Predicate>
void wait(std::unique_lock<std::mutex>& lock, Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимо и возвращать значение, преобразуемое в тип `bool`. Значение `lock.owns_lock()` должно быть равно `true`, и владельцем блокировки `lock` должен быть поток, вызвавший `wait()`.

Результат

Эквивалентно циклу

```
while (!pred()) {
 wait(lock);
}
```

Исключения

Исключение, возбужденное в результате обращения к `pred`, или `std::system_error`, если действие не выполнено.

> **Примечание**. Возможность ложного пробуждения означает, что функция `pred` может вызываться несколько раз (сколько именно, не определено). При любом вызове `pred` мьютекс, на который ссылается объект `lock`, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления `(bool)pred()` является `true`.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE::WAIT_FOR` , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная `std::condition_variable` не получит сигнал в результате обращения к `notify_one()` или `notify_all()`, либо не истечет таймаут, либо не произойдёт ложное пробуждение.

Объявление

```
template<typename Rep, typename Period>
cv_status wait_for(
std::unique_lock<std::mutex>& lock,
std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

Значение `lock.owns_lock()` равно `true`, и блокировкой владеет вызывающий поток.

Результат

Атомарно разблокирует предоставленный объект `lock` и блокирует поток, пока он не будет разбужен обращением к `notify_one()` или `notify_all()` из другого потока, либо не истечет таймаут, заданный аргументом `relative_time`, либо не произойдёт ложное пробуждение. Перед возвратом управления из `wait_for()` объект `lock` снова блокируется.

Возвращаемое значение

`std::cv_status::no_timeout`, если поток был разбужен в результате обращения к `notify_one()` или `notify_all()` либо ложного пробуждения. В противном случае `std::cv_status::timeout`.

Исключения

Исключение типа `std::system_error`, если действие не выполнено. Если объект `lock` был разблокирован при обращении к `wait_for()`, он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

> **Примечание**. Ложное пробуждение означает, что поток, вызвавший `wait_for()`, может быть разбужен, даже если ни один другой поток не обращался к `notify_one()` или `notify_all()`. Поэтому рекомендуется использовать перегруженный вариант `wait_for()`, который принимает предикат. Если это нежелательно, то рекомендуется вызывать `wait_for()` в цикле, где проверяется предикат, ассоциированный с условной переменной. При этом необходимо следить, не истек ли таймаут. Во многих случаях предпочтительнее использовать функцию `wait_until()`. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE::WAIT_FOR` , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная `std::condition_variable` не получит сигнал в результате обращения к `notify_one()` или `notify_all()` и при этом предикат равен `true`, либо не истечет указанный таймаут.

Объявление

```
template<typename Rep, typename Period, typename Predicate>
bool wait_for(
 std::unique_lock<std::mutex>& lock,
 std::chrono::duration<Rep, Period> const& relative_time,
 Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимо и возвращать значение, преобразуемое в тип `bool`. Значение `lock.owns_lock()` должно быть равно `true`, и владельцем блокировки `lock` должен быть поток, вызвавший `wait_for()`.

Результат

Эквивалентно следующему коду:

```
internal_clock::time_point end =
 internal_clock::now() + relative_time;
while (!pred()) {
 std::chrono::duration<Rep, Period> remaining_time =
  end-internal_clock::now();
 if (wait_for(lock, remaining_time) == std::cv_status::timeout)
  return pred();
}
return true;
```

Возвращаемое значение

`true`, если последнее обращение к `pred()` вернуло `true`; `false`, если истекло время,

заданное в аргументе relative_time и обращение к pred() вернуло false.

Примечание. Возможность ложного пробуждения означает, что функция pred может вызываться несколько раз (сколько именно, не определено). При любом вызове pred мьютекс, на который ссылается объект lock, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления (bool)pred() является true или истекло время, заданное в аргументе relative_time. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

Исключения

Исключение, возбужденное в результате обращения к pred, или std::system_error, если действие не выполнено.

Синхронизация

Обращения к функциям notify_one(), notify_all(), wait(), wait_for() и wait_until() одного и того же объекта std::condition_variable сериализуются. Обращение к notify_one() или notify_all() будит только потоки, запущенные *до* этого обращения.

STD::CONDITION_VARIABLE::WAIT_UNTIL , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная std::condition_variable не получит сигнал в результате обращения к notify_one() или notify_all() либо не будет достигнут указанный момент времени, либо не произойдёт ложное пробуждение.

Объявление

```
template<typename Clock, typename Duration>
cv_status wait_until(
 std::unique_lock<std::mutex>& lock,
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

Значение lock.owns_lock() равно true, и владельцем блокировки lock является вызывающий поток.

Результат

Атомарно разблокирует предоставленный объект lock и блокирует поток, пока он не будет разбужен обращением к notify_one() или notify_all() из другого потока, либо функция Clock::now() не вернет время, большее или равное absolute_time, либо не произойдёт ложное пробуждение. Перед возвратом управления из wait_until() объект lock снова блокируется.

Возвращаемое значение

std::cv_status::no_timeout, если поток был разбужен в результате обращения к notify_one() или notify_all() либо ложного пробуждения. В противном случае std::cv_status::timeout.

Исключения

Исключение типа std::system_error, если действие не выполнено. Если объект lock был разблокирован при обращении к wait_for(), он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

Примечание. Ложное пробуждение означает, что поток, вызвавший

wait_until(), может быть разбужен, даже если ни один другой поток не обращался к notify_one() или notify_all(). Поэтому рекомендуется использовать перегруженный вариант wait_until(), который принимает предикат. Если это нежелательно, то рекомендуется вызывать wait_until() в цикле, где проверяется предикат, ассоциированный с условной переменной. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула false, то значение, возвращенное Clock::now(), больше или равно absolute_time в точке, где поток разблокировался.

Синхронизация

Обращения к функциям notify_one(), notify_all(), wait(), wait_for() и wait_until() одного и того же объекта std::condition_variable сериализуются. Обращение к notify_one() или notify_all() будит только потоки, запущенные *до* этого обращения.

STD::CONDITION_VARIABLE::WAIT_UNTIL , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная std::condition_variable не получит сигнал в результате обращения к notify_one() или notify_all(), и при этом предикат равен true, либо не будет достигнут указанный момент времени.

Объявление

```
template<typename Clock, typename Duration, typename Predicate>
bool wait_until(
 std::unique_lock<std::mutex>& lock,
 std::chrono::time_point<Clock, Duration> const& absolute_time,
 Predicate pred);
```

Предусловия

Выражение pred() должно быть допустимо и возвращать значение, преобразуемое в тип bool. Значение lock.owns_lock() должно быть равно true, и владельцем блокировки lock должен быть поток, вызвавший wait_until().

Результат

Эквивалентно следующему коду:

```
while (!pred()) {
 if (wait_until(lock, absolute_time) == std::cv_status::timeout)
  return pred();
}
return true;
```

Возвращаемое значение

true, если последнее обращение к pred() вернуло true; false, если функция Clock::now() вернула время, большее или равное absolute_time, и обращение к pred() вернуло false.

Примечание. Возможность ложного пробуждения означает, что функция pred может вызываться несколько раз (сколько именно, не определено). При любом вызове pred мьютекс, на который ссылается объект lock, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления (bool)pred() является true или функция Clock::now() вернула

время, больше или равное absolute_time. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула false, то значение, возвращенное Clock::now(), больше или равно absolute_time в точке, где поток разблокировался.

Исключения

Исключение, возбужденное в результате обращения к pred, или std::system_error, если действие не выполнено.

Синхронизация

Обращения к функциям notify_one(), notify_all(), wait(), wait_for() и wait_until() одного и того же объекта std::condition_variable сериализуются. Обращение к notify_one() или notify_all() будит только потоки, запущенные *до* этого обращения.

STD::NOTIFY_ALL_AT_THREAD_EXIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Пробуждает все потоки, ожидающие std::condition_variable, при завершении текущего потока.

Объявление

```
void notify_all_at_thread_exit(
 condition_variable& cv, unique_lock<mutex> lk);
```

Предусловия

Значение lock.owns_lock() равно true, и владельцем блокировки lock является вызывающий поток. Функция lk.mutex() должна возвращать такое же значение, как для любого объекта блокировки, передаваемого функциям-членам wait(), wait_for() или wait_until() объекта cv из одновременно ожидающих потоков.

Результат

Передает владение мьютексом, захваченным lk, внутреннему объекту и планирует отправку уведомления условной переменной cv при завершении вызывающего потока. Уведомление эквивалентно выполнению следующего кода:

```
lk.unlock();
cv.notify_all();
```

Исключения

Возбуждает исключение std::system_error, если действие не выполнено.

Примечание. Блокировка удерживается до завершения потока, поэтому необходимо предпринимать меры для предотвращения взаимоблокировки. Рекомендуется завершать вызывающий поток как можно раньше и не выполнять в нем никаких блокирующих операций.

Пользователь должен следить за тем, чтобы ожидающий поток не сделал ошибочного предположения о том, что в момент его пробуждения данный поток уже завершен, — в частности, из-за возможности ложного пробуждения. Для этого можно проверять в ожидающем потоке предикат, который может быть сделан истинным только уведомляющим потоком, причём это должно делаться под защитой мьютекса, который не освобождается до вызова notify_all_at_thread_exit.

D.2.2. Класс `std::condition_variable_any`

Класс `std::condition_variable_any` позволяет потоку ждать выполнения условия. Если объект `std::condition_variable` можно использовать только с блокировкой типа `std::unique_lock<std::mutex>`, то `std::condition_variable_any` допустимо использовать с блокировкой *любого* типа, удовлетворяющего требованиям концепции `Lockable`.

Экземпляры `std::condition_variable_any` не удовлетворяют концепциям `CopyAssignable`, `CopyConstructible`, `MoveAssignable`, `MoveConstructible`.

Определение класса

```
class condition_variable_any {
public:
 condition_variable_any();
 ~condition_variable_any();

 condition_variable_any(
  condition_variable_any const&) = delete;
 condition_variable_any& operator=(
  condition_variable_any const&) = delete;

 void notify_one() noexcept;
 void notify_all() noexcept;

 template<typename Lockable>
 void wait(Lockable& lock);

 template <typename Lockable, typename Predicate>
 void wait(Lockable& lock, Predicate pred);

 template <typename Lockable, typename Clock, typename Duration>
 std::cv_status wait_until(
  Lockable& lock,
  const std::chrono::time_point<Clock, Duration>& absolute_time);

 template <
  typename Lockable, typename Clock,
  typename Duration, typename Predicate>
 bool wait_until(
  Lockable& lock,
  const std::chrono::time_point<Clock, Duration>& absolute_time,
  Predicate pred);

 template <typename Lockable, typename Rep, typename Period>
 std::cv_status wait_for(
  Lockable& lock,
  const std::chrono::duration<Rep, Period>& relative_time);

 template <
  typename Lockable, typename Rep,
  typename Period, typename Predicate>
 bool wait_for(
  Lockable& lock,
  const std::chrono::duration<Rep, Period>& relative_time,
```

```
    Predicate pred);
};
```

STD::CONDITION_VARIABLE_ANY , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект типа `std::condition_variable_any`.

Объявление

```
condition_variable_any();
```

Результат

Конструирует объект типа `std::condition_variable_any`.

Исключения

Исключение типа `std::system_error`, если сконструировать условную переменную не получилось.

STD::CONDITION_VARIABLE_ANY , ДЕСТРУКТОР

Уничтожает объект `Std::condition_variable_any`.

Объявление

```
~condition_variable_any();
```

Предусловия

Не существует потоков, заблокированных по `*this` в обращениях к `wait()`, `wait_for()` или `wait_until()`.

Результат

Уничтожает `*this`.

Исключения

Нет.

STD::CONDITION_VARIABLE_ANY::NOTIFY_ONE , ФУНКЦИЯ-ЧЛЕН

Пробуждает один из потоков, ожидающих `std::condition_variable_any`.

Объявление

```
void notify_one() noexcept;
```

Результат

Пробуждает один из потоков, ожидающих `*this`, в точке вызова. Если таких потоков нет, функция не имеет никакого эффекта.

Исключения

Исключение типа `std::system_error`, если действие не выполнено.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

STD::CONDITION_VARIABLE_ANY::NOTIFY_ALL , ФУНКЦИЯ-ЧЛЕН

Пробуждает все потоки, ожидающие `std::condition_variable_any`.

Объявление

```
void notify_all() noexcept;
```

Результат

Пробуждает все потоки, ожидающие `*this`, в точке вызова. Если таких потоков нет, функция не имеет никакого эффекта.

Исключения

Исключение типа `std::system_error`, если действие не выполнено.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE::WAIT` , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная `std::condition_variable_any` не получит сигнал в результате обращения к `notify_one()` или `notify_all()` либо не произойдёт ложное пробуждение.

Объявление

```
template<typename Lockable>
void wait(Lockable& lock);
```

Предусловия

Тип `Lockable` удовлетворяет требованиям концепции `Lockable` и `lock` владеет блокировкой.

Результат

Атомарно разблокирует предоставленный объект `lock` и блокирует поток, пока он не будет разбужен обращением к `notify_one()` или `notify_all()` из другого потока либо не произойдёт ложное пробуждение. Перед возвратом управления из `wait()` объект `lock` снова блокируется.

Исключения

Исключение типа `std::system_error`, если действие не выполнено. Если объект `lock` был разблокирован при обращении к `wait()`, он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

> **Примечание**. Ложное пробуждение означает, что поток, вызвавший `wait()`, может быть разбужен, даже если ни один другой поток не обращался к `notify_one()` или `notify_all()`. Поэтому рекомендуется использовать перегруженный вариант `wait()`, который принимает предикат. Если это нежелательно, то рекомендуется вызывать `wait()` в цикле, где проверяется предикат, ассоциированный с условной переменной.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE_ANY::WAIT` , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная `std::condition_variable_any` получит сигнал в результате обращения к `notify_one()` или `notify_all()` и при этом предикат равен `true`.

Объявление

```
template<typename Lockable, typename Predicate>
void wait(Lockable& lock, Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимо и возвращать значение, преобразуемое в тип `bool`. Тип `Lockable` удовлетворяет требованиям концепции `Lockable` и `lock` владеет

блокировкой.

Результат

Эквивалентно циклу

```
while (!pred()) {
 wait(lock);
}
```

Исключения

Исключение, возбужденное в результате обращения к `pred`, или `std::system_error`, если действие не выполнено.

> **Примечание**. Возможность ложного пробуждения означает, что функция `pred` может вызываться несколько раз (сколько именно, не определено). При любом вызове `pred` мьютекс, на который ссылается объект `lock`, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления `(bool)pred()` является `true`.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE_ANY::WAIT_FOR` , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная `std::condition_variable_any` получит сигнал в результате обращения к `notify_one()` или `notify_all()`, либо истечет таймаут, либо произойдёт ложное пробуждение.

Объявление

```
template<typename Lockable, typename Rep, typename Period>
std::cv_status wait_for(
 Lockable& lock,
 std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

Тип `Lockable` удовлетворяет требованиям концепции `Lockable` и `lock` владеет блокировкой.

Результат

Атомарно разблокирует предоставленный объект `lock` и блокирует поток, пока он не будет разбужен обращением к `notify_one()` или `notify_all()` из другого потока, либо не истечет таймаут, заданный аргументом `relative_time`, либо не произойдёт ложное пробуждение. Перед возвратом управления из `wait_for()` объект `lock` снова блокируется.

Возвращаемое значение

`std::cv_status::no_timeout`, если поток был разбужен в результате обращения к `notify_one()` или `notify_all()` либо ложного пробуждения. В противном случае `std::cv_status::timeout`.

Исключения

Исключение типа `std::system_error`, если действие не выполнено. Если объект `lock` был разблокирован при обращении к `wait_for()`, он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

Примечание. Ложное пробуждение означает, что поток, вызвавший `wait_for()`, может быть разбужен, даже если ни один другой поток не обращался к `notify_one()` или `notify_all()`. Поэтому рекомендуется использовать перегруженный вариант `wait_for()`, который принимает предикат. Если это нежелательно, то рекомендуется вызывать `wait_for()` в цикле, где проверяется предикат, ассоциированный с условной переменной. При этом необходимо следить, не истек ли таймаут. Во многих случаях предпочтительнее использовать функцию `wait_until()`. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE_ANY::WAIT_FOR` , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная `std::condition_variable_any` получит сигнал в результате обращения к `notify_one()` или `notify_all()` и при этом предикат равен `true`, либо истечет указанный таймаут.

Объявление

```
template<typename Lockable, typename Rep,
         typename Period, typename Predicate>
bool wait_for(
 Lockable& lock,
 std::chrono::duration<Rep, Period> const& relative_time,
 Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимо и возвращать значение, преобразуемое в тип `bool`. Тип `Lockable` удовлетворяет требованиям концепции `Lockable` и `lock` владеет блокировкой.

Результат

Эквивалентно следующему коду:

```
internal_clock::time_point end
 = internal_clock::now() + relative_time;
while (!pred()) {
 std::chrono::duration<Rep, Period> remaining_time =
  end-internal_clock::now();
 if (wait_for(lock, remaining_time) == std::cv_status::timeout)
  return pred();
}
return true;
```

Возвращаемое значение

`true`, если последнее обращение к `pred()` вернуло `true`; `false`, если истекло время, заданное в аргументе `relative_time` и обращение к `pred()` вернуло `false`.

Примечание. Возможность ложного пробуждения означает, что функция `pred` может вызываться несколько раз (сколько именно, не определено). При любом

вызове pred мьютекс, на который ссылается объект lock, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления (bool)pred() является true или истекло время, заданное в аргументе relative_time. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

Исключения

Исключение, возбужденное в результате обращения к pred, или std::system_error, если действие не выполнено.

Синхронизация

Обращения к функциям notify_one(), notify_all(), wait(), wait_for() и wait_until() одного и того же объекта std::condition_variable_any сериализуются. Обращение к notify_one() или notify_all() будит только потоки, запущенные *до* этого обращения.

STD::CONDITION_VARIABLE_ANY::WAIT_UNTIL , ФУНКЦИЯ-ЧЛЕН

Ожидает, пока условная переменная std::condition_variable_any получит сигнал в результате обращения к notify_one() или notify_all() либо будет достигнут указанный момент времени, либо произойдёт ложное пробуждение.

Объявление

```
template<typename Lockable, typename Clock, typename Duration>
std::cv_status wait_until(
 Lockable& lock,
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

Тип Lockable удовлетворяет требованиям концепции Lockable и lock владеет блокировкой.

Результат

Атомарно разблокирует предоставленный объект lock и блокирует поток, пока он не будет разбужен обращением к notify_one() или notify_all() из другого потока, либо функция Clock::now() не вернет время, большее или равное absolute_time, либо не произойдёт ложное пробуждение. Перед возвратом управления из wait_until() объект lock снова блокируется.

Возвращаемое значение

std::cv_status::no_timeout, если поток был разбужен в результате обращения к notify_one() или notify_all() либо ложного пробуждения. В противном случае std::cv_status::timeout.

Исключения

Исключение типа std::system_error, если действие не выполнено. Если объект lock был разблокирован при обращении к wait_for(), он снова блокируется при выходе из нее, даже если выход произошёл в результате исключения.

Примечание. Ложное пробуждение означает, что поток, вызвавший wait_until(), может быть разбужен, даже если ни один другой поток не обращался к notify_one() или notify_all(). Поэтому рекомендуется использовать перегруженный вариант wait_until(), который принимает предикат. Если это нежелательно, то рекомендуется вызывать wait_until() в цикле, где

проверяется предикат, ассоциированный с условной переменной. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула `false`, то значение, возвращенное `Clock::now()`, больше или равно `absolute_time` в точке, где поток разблокировался.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

`STD::CONDITION_VARIABLE_ANY::WAIT_UNTIL` , ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ-ЧЛЕН, ПРИНИМАЮЩАЯ ПРЕДИКАТ

Ожидает, пока условная переменная `std::condition_variable_any` но лучит сигнал в результате обращения к `notify_one()` или `notify_all()`, и при этом предикат равен `true`, либо будет достигнут указанный момент времени.

Объявление

```
template<typename Lockable, typename Clock,
         typename Duration, typename Predicate>
bool wait_until(
 Lockable& lock,
 std::chrono::time_point<Clock, Duration> const& absolute_time,
 Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимо и возвращать значение, преобразуемое в тип `bool`. Тип `Lockable` удовлетворяет требованиям концепции `Lockable` и `lock` владеет блокировкой.

Результат

Эквивалентно следующему коду:

```
while (!pred()) {
 if (wait_until(lock, absolute_time) == std::cv_status::timeout)
  return pred();
}
return true;
```

Возвращаемое значение

`true`, если последнее обращение к `pred()` вернуло `true`; `false`, если функция `Clock::now()` вернула время, большее или равное `absolute_time`, и обращение к `pred()` вернуло `false`.

Примечание. Возможность ложного пробуждения означает, что функция `pred` может вызываться несколько раз (сколько именно, не определено). При любом вызове `pred` мьютекс, на который ссылается объект `lock`, гарантированно будет захвачен, и функция вернет управление тогда и только тогда, когда результатом вычисления `(bool)pred()` является `true` или функция `Clock::now()` вернула время, большее или равное `absolute_time`. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула `false`, то значение, возвращенное

`Clock::now()`, больше или равно `absolute_time` в точке, где поток разблокировался.

Исключения

Исключение, возбужденное в результате обращения к `pred`, или `std::system_error`, если действие не выполнено.

Синхронизация

Обращения к функциям `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` одного и того же объекта `std::condition_variable_any` сериализуются. Обращение к `notify_one()` или `notify_all()` будит только потоки, запущенные *до* этого обращения.

D.3. Заголовок <atomic>

В заголовке <atomic> объявлены простые атомарные типы и операции над ними, а также шаблон класса для построения атомарной версии определённого пользователем типа, удовлетворяющего некоторым условиям.

Содержимое заголовка

```
#define ATOMIC_BOOL_LOCK_FREE см. описание
#define ATOMIC_CHAR_LOCK_FREE см. описание
#define ATOMIC_SHORT_LOCK_FREE см. описание
#define ATOMIC_INT_LOCK_FREE см. описание
#define ATOMIC_LONG_LOCK_FREE см. описание
#define ATOMIC_LLONG_LOCK_FREE см. описание
#define ATOMIC_CHAR16_T_LOCK_FREE см. описание
#define ATOMIC_CHAR32_T_LOCK_FREE см. описание
#define ATOMIC_WCHAR_T_LOCK_FREE см. описание
#define ATOMIC_POINTER_LOCK_FREE см. описание

#define ATOMIC_VAR_INIT(value) см. описание

namespace std {
enum memory_order;
struct atomic_flag;
typedef см. описание atomic_bool;
typedef см. описание atomic_char;
typedef см. описание atomic_char16_t;
typedef см. описание atomic_char32_t;
typedef см. описание atomic_schar;
typedef см. описание atomic_uchar;
typedef см. описание atomic_short;
typedef см. описание atomic_ushort;
typedef см. описание atomic_int;
typedef см. описание atomic_uint;
typedef см. описание atomic_long;
typedef см. описание atomic_ulong;
typedef см. описание atomic_llong;
typedef см. описание atomic_ullong;
typedef см. описание atomic_wchar_t;

typedef см. описание atomic_int_least8_t;
typedef см. описание atomic_uint_least8_t;
typedef см. описание atomic_int_least16_t;
typedef см. описание atomic_uint_least16_t;
typedef см. описание atomic_int_least32_t;
typedef см. описание atomic_uint_least32_t;
typedef см. описание atomic_int_least64_t;
typedef см. описание atomic_uint_least64_t;
typedef см. описание atomic_int_fast8_t;
typedef см. описание atomic_uint_fast8_t;
typedef см. описание atomic_int_fast16_t;
typedef см. описание atomic_uint_fast16_t;
typedef см. описание atomic_int_fast32_t;
typedef см. описание atomic_uint_fast32_t;
```

```
    typedef см. описание atomic_int_fast64_t;
    typedef см. описание atomic_uint_fast64_t;
    typedef см. описание atomic_int8_t;
    typedef см. описание atomic_uint8_t;
    typedef см. описание atomic_int16_t;
    typedef см. описание atomic_uint16_t;
    typedef см. описание atomic_int32_t;
    typedef см. описание atomic_uint32_t;
    typedef см. описание atomic_int64_t;
    typedef см. описание atomic_uint64_t;
    typedef см. описание atomic_intptr_t;
    typedef см. описание atomic_uintptr_t;
    typedef см. описание atomic_size_t;
    typedef см. описание atomic_ssize_t;
    typedef см. описание atomic_ptrdiff_t;
    typedef см. описание atomic_intmax_t;
    typedef см. описание atomic_uintmax_t;

    template<typename T>
    struct atomic;

    extern "C" void atomic_thread_fence(memory_order order);
    extern "C" void atomic_signal_fence(memory_order order);

    template<typename T>
    T kill_dependency(T);
    }
```

D.3.1. `std::atomic_xxx`, псевдонимы типов

Для совместимости с ожидаемым стандартом C предоставляются псевдонимы `typedef` для атомарных целочисленных типов. Это псевдонимы либо соответствующей специализации `std::atomic<T>`, либо базового класса этой специализации с таким же интерфейсом.

Таблица D.1. Псевдонимы атомарных типов и соответствующие им специализации `std::atomic<>`

`std::atomic_`*`itype`*	**Специализация `std::atomic<>`**
`std::atomic_char`	`std::atomic<char>`
`std::atomic_schar`	`std::atomic<signed char>`
`std::atomic_uchar`	`std::atomic<unsigned char>`
`std::atomic_short`	`std::atomic<short>`
`std::atomic_ushort`	`std::atomic<unsigned short>`
`std::atomic_int`	`std::atomic<int>`
`std::atomic_uint`	`std::atomic<unsigned int>`
`std::atomic_long`	`std::atomic<long>`
`std::atomic_ulong`	`std::atomic<unsigned long>`
`std::atomic_llong`	`std::atomic<long long>`
`std::atomic_ullong`	`std::atomic<unsigned long long>`
`std::atomic_wchar_t`	`std::atomic<wchar_t>`

```
std::atomic_char16_t std::atomic<char16_t>
std::atomic_char32_t std::atomic<char32_t>
```

D.3.2. ATOMIC_*xxx*_LOCK_FREE, макросы

Эти макросы определяют, являются ли атомарные типы, соответствующие различным встроенным типам, свободными от блокировок.

Объявления макросов

```
#define ATOMIC_BOOL_LOCK_FREE см. описание
#define ATOMIC_CHAR_LOCK_FREE см. описание
#define ATOMIC_SHORT_LOCK_FREE см. описание
#define ATOMIC_INT_LOCK_FREE см. описание
#define ATOMIC_LONG_LOCK_FREE см. описание
#define ATOMIC_LLONG_LOCK_FREE см. описание
#define ATOMIC_CHAR16_T_LOCK_FREE см. описание
#define ATOMIC_CHAR32_T_LOCK_FREE см. описание
#define ATOMIC_WCHAR_T_LOCK_FREE см. описание
#define ATOMIC_POINTER_LOCK_FREE см. описание
```

Значением ATOMIC_*xxx*_LOCK_FREE может быть 0, 1 или 2. Значение 0 означает, что операции над знаковыми и беззнаковыми атомарными типами, соответствующими типу *xxx*, никогда не свободны от блокировок; 1 — что операции могут быть свободны от блокировок для одних экземпляров этих типов и не свободны для других; 2 — что операции всегда свободны от блокировок. Например, если ATOMIC_INT_LOCK_FREE равно 2, то операции над любыми экземплярами std::atomic<int> и std::atomic<unsigned> свободны от блокировок.

Макрос ATOMIC_POINTER_LOCK_FREE позволяет узнать, свободны ли от блокировок операции над атомарными специализациями указателя std::atomic<T*>.

D.3.3. ATOMIC_VAR_INIT, макрос

Макрос ATOMIC_VAR_INIT позволяет инициализировать атомарную переменную конкретным значением.

Объявление

```
#define ATOMIC_VAR_INIT(value) см. описание
```

Макрос расширяется в последовательность лексем, которую можно использовать в выражении следующего вида для инициализации одного из стандартных атомарных типов указанным значением:

```
std::atomic<type> x = ATOMIC_VAR_INIT(val);
```

Указанное значение должно быть совместимо с неатомарным типом, соответствующим данной атомарной переменной, например:

```
std::atomic<int> i = ATOMIC_VAR_INIT(42);
std::string s;
std::atomic<std::string*> p = ATOMIC_VAR_INIT(&s);
```

Такая инициализация не атомарна, то есть любой доступ из другого потока к инициализируемой переменной в случае, когда инициализация не происходит-раньше этого доступа, приводит к гонке за данными и, следовательно, к неопределённому поведению.

D.3.4. `std::memory_order`, перечисление

Перечисление `std::memory_order` применяется для задания упорядочения доступа к памяти при выполнении атомарных операций.

Объявление

```
typedef enum memory_order {
 memory_order_relaxed, memory_order_consume,
 memory_order_acquire, memory_order_release,
 memory_order_acq_rel, memory_order_seq_cst
} memory_order;
```

Операции, помеченные элементами этого перечисления, ведут себя, как описано ниже (подробное описание упорядочения доступа к памяти см. в главе 5).

`STD::MEMORY_ORDER_RELAXED`

Операция не обеспечивает никаких дополнительных ограничений на упорядочение.

`STD::MEMORY_ORDER_RELEASE`

Операция освобождения указанной ячейки памяти. Следовательно, она синхронизируется-с операцией захвата той же ячейки памяти, которая читает сохраненное значение.

`STD::MEMORY_ORDER_ACQUIRE`

Операция захвата указанной ячейки памяти. Если сохраненное значение было записано операцией освобождения, то сохранение синхронизируется-с этой операцией.

`STD::MEMORY_ORDER_ACQ_REL`

Операция чтения-модификации-записи. Ведет себя так, как будто одновременно заданы ограничения `std::memory_order_acquire` и `std::memory_order_release` для доступа к указанной ячейке памяти.

`STD::MEMORY_ORDER_SEQ_CST`

Операция является частью цепочки последовательно согласованных операций, на которой определено полное упорядочение. Кроме того, если это сохранение, то оно ведет себя как операция с ограничением `std::memory_order_release`, если загрузка — то как операция с ограничением `std::memory_order_acquire`, а если это операция чтения-модификации-записи, то она ведет себя как операция с обоими ограничениями `std::memory_order_acquire` и `std::memory_order_release`. *Эта семантика по умолчанию подразумевается для всех операций*.

`STD::MEMORY_ORDER_CONSUME`

Операция потребления указанной ячейки памяти.

D.3.5. `std::atomic_thread_fence`, функция

Функция `std::atomic_thread_fence()` вставляет в программу «барьер», чтобы принудительно обеспечить упорядочение доступа к памяти со стороны нескольких операций.

Объявление

```
extern "C" void atomic_thread_fence(std::memory_order order);
```

Результат

Вставляет барьер с требуемыми ограничениями на упорядочение доступа к памяти.

Барьер, для которого параметр `order` равен `std::memory_order_release`, `std::memory_order_acq_rel` или `std::memory_order_seq_cst` синхронизируется-с

операцией захвата некоторой ячейки памяти, если эта операция читает значение, сохраненное атомарной операцией, следующей за барьером в том же потоке, где поставлен барьер.

Операция освобождения синхронизируется-с барьером, для которого параметр `order` равен `std::memory_order_acquire`, `std::memory_order_acq_rel` или `std::memory_order_seq_cst`, если эта операция освобождения сохраняет значение, которое читается атомарной операцией, предшествующей барьеру, в том же потоке, где поставлен барьер.

Исключения

Нет.

D.3.6. `std::atomic_signal_fence`, функция

Функция `std::atomic_signal_fence()` вставляет в программу «барьер», чтобы принудительно обеспечить упорядочение доступа к памяти со стороны операций в некотором потоке и операций в обработчике сигнала, находящемся в том же потоке.

Объявление

```
extern "C" void atomic_signal_fence(std::memory_order order);
```

Результат

Вставляет барьер с требуемыми ограничениями на упорядочение доступа к памяти. Функция эквивалентна `std::atomic_thread_fence(order)` с тем отличием, что ограничения применяются только к потоку и обработчику сигнала в том же потоке.

Исключения

Нет.

D.3.7. `std::atomic_flag`, класс

Класс `std::atomic_flag` предоставляет самый простой атомарный флаг. Это единственный тип данных в стандарте C++, который *гарантированно* свободен от блокировок (хотя в большинстве реализаций этим свойством обладают и многие другие атомарные типы).

Объект типа `std::atomic_flag` может находиться в одном из двух состояний: *установлен* или *сброшен*.

Определение класса

```
struct atomic_flag {
 atomic_flag() noexcept = default;
 atomic_flag(const atomic_flag&) = delete;
 atomic_flag& operator=(const atomic_flag&) = delete;
 atomic_flag& operator=(const atomic_flag&) volatile = delete;
 bool test_and_set(memory_order = memory_order_seq_cst)
  volatile noexcept;
 bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
 void clear(memory_order = memory_order_seq_cst)
  volatile noexcept;
 void clear(memory_order = memory_order_seq_cst) noexcept;
};
bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
```

```
bool atomic_flag_test_and_set_explicit(
 volatile atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_and_set_explicit(
 atomic_flag*, memory_order) noexcept;
void atomic_flag_clear(volatile atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
void atomic_flag_clear_explicit(
 volatile atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_explicit(
 atomic_flag*, memory_order) noexcept;

#define ATOMIC_FLAG_INIT unspecified
```

STD::ATOMIC_FLAG , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Не оговаривается, в каком состоянии находится сконструированный по умолчанию экземпляр `std::atomic_flag`: *установлен* или *сброшен.* Для объектов со статическим временем жизни обеспечивается статическая инициализация.

Объявление

```
std::atomic_flag() noexcept = default;
```

Результат

Конструирует новый объект `std::atomic_flag` в неопределенном состоянии.

Исключения

Нет.

STD::ATOMIC_FLAG , ИНИЦИАЛИЗАЦИЯ МАКРОСОМ ATOMIC_FLAG_INIT

Экземпляр типа `std::atomic_flag` может быть инициализирован макросом `ATOMIC_FLAG_INIT`, и в таком случае его начальное состояние — *сброшен*. Для объектов со статическим временем жизни обеспечивается статическая инициализация.

Объявление

```
#define ATOMIC_FLAG_INIT unspecified
```

Использование

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;
```

Результат

Конструирует новый объект `std::atomic_flag` в состоянии *сброшен*.

Исключения

Нет.

STD::ATOMIC_FLAG::TEST_AND_SET , ФУНКЦИЯ-ЧЛЕН

Атомарно устанавливает флаг и проверяет, был ли он установлен.

Объявление

```
bool test_and_set(memory_order order = memory_order_seq_cst)
 volatile noexcept;
bool test_and_set(memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно устанавливает флаг.

Возвращаемое значение

`true`, если флаг был установлен в точке вызова; `false`, если флаг был сброшен.

Исключения

Нет.

Примечание. Это атомарная операция чтения-модификации-записи для

ячейки памяти, содержащей `*this`.

`STD::ATOMIC_FLAG_TEST_AND_SET` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно устанавливает флаг и проверяет, был ли он установлен.

Объявление

```
bool atomic_flag_test_and_set(
 volatile atomic_flag* flag) noexcept;
bool atomic_flag_test_and_set(atomic_flag* flag) noexcept;
```

Результат

```
return flag->test_and_set();
```

`STD::ATOMIC_FLAG_TEST_AND_SET_EXPLICIT` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно устанавливает флаг и проверяет, был ли он установлен.

Объявление

```
bool atomic_flag_test_and_set_explicit(
 volatile atomic_flag* flag, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(
 atomic_flag* flag, memory_order order) noexcept;
```

Результат

```
return flag->test_and_set(order);
```

`STD::ATOMIC_FLAG::CLEAR` **, ФУНКЦИЯ-ЧЛЕН**

Атомарно сбрасывает флаг.

Объявление

```
void clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void clear(memory_order order = memory_order_seq_cst) noexcept;
```

Предусловия

Параметр order должен принимать одно из значений `std::memory_order_relaxed`, `std::memory_order_release` или `std::memory_order_seq_cst`.

Результат

Атомарно сбрасывает флаг.

Исключения

Нет.

Примечание. Это атомарная операция сохранения для ячейки памяти, содержащей `*this`.

`STD::ATOMIC_FLAG_CLEAR` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сбрасывает флаг.

Объявление

```
void atomic_flag_clear(volatile atomic_flag* flag) noexcept;
void atomic_flag_clear(atomic_flag* flag) noexcept;
```

Результат

```
flag->clear();
```

`STD::ATOMIC_FLAG_CLEAR_EXPLICIT` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сбрасывает флаг.

Объявление

```
void atomic_flag_clear_explicit(
 volatile atomic_flag* flag, memory_order order) noexcept;
void atomic_flag_clear_explicit(
 atomic_flag* flag, memory_order order) noexcept;
```

Результат

```
return flag->clear(order);
```

D.3.8. Шаблон класса `std::atomic`

Шаблон класса `std::atomic` является оберткой, позволяющей строить атомарные операции для любого типа, удовлетворяющего следующим условиям.

Параметр шаблона `BaseType` должен:

• иметь тривиальный конструктор по умолчанию;

• иметь тривиальный копирующий оператор присваивания;

• иметь тривиальный деструктор;

• допускать побитовое сравнение на равенство.

По существу, это означает, что конкретизация `std::atomic<некоторый-встроенный-тип>` допустима, как и конкретизация `std::atomic<некоторая-простая-структура>`, но такие вещи, как `std::atomic<std::string>`, недопустимы.

Помимо основного шаблона, имеются специализации для встроенных целочисленных типов и указателей, которые предоставляют дополнительные операции, например `x++`.

Экземпляры `std::atomic` не удовлетворяют требованиям концепций `CopyConstructible` и `CopyAssignable`, потому что такие операции невозможно выполнить атомарно.

Определение класса

```
template<typename BaseType>
struct atomic {
 atomic() noexcept = default;
 constexpr atomic(BaseType) noexcept;
 BaseType operator=(BaseType) volatile noexcept;
 BaseType operator=(BaseType) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&) = delete;
 atomic& operator=(const atomic&) volatile = delete;

 bool is_lock_free() const volatile noexcept;
 bool is_lock_free() const noexcept;
 void store(BaseType, memory_order = memory_order_seq_cst)
  volatile noexcept;
 void store(BaseType, memory_order = memory_order_seq_cst)
  noexcept;
 BaseType load(memory_order = memory_order_seq_cst)
  const volatile noexcept;
 BaseType load(memory_order = memory_order_seq_cst)
  const noexcept;
 BaseType exchange(BaseType, memory_order = memory_order_seq_cst)
  volatile noexcept;
 BaseType exchange(BaseType, memory_order = memory_order_seq_cst)
  noexcept;
```

```
 bool compare_exchange_strong(
  BaseType & old_value, BaseType new_value,
  memory_order order = memory_order_seq_cst) volatile noexcept;
 bool compare_exchange_strong(
  BaseType & old_value, BaseType new_value,
  memory_order order = memory_order_seq_cst) noexcept;
 bool compare_exchange_strong(
  BaseType & old_value, BaseType new_value,
  memory_order success_order,
  memory_order failure_order) volatile noexcept;
 bool compare_exchange_strong(
  BaseType & old_value, BaseType new_value,
  memory_order success_order,
  memory_order failure_order) noexcept;
 bool compare_exchange_weak(
  BaseType & old_value, BaseType new_value,
  memory_order order = memory_order_seq_cst)
  volatile noexcept;
 bool compare_exchange_weak(
  BaseType & old_value, BaseType new_value,
  memory_order order = memory_order_seq_cst) noexcept;
 bool compare_exchange_weak(
  BaseType & old_value, BaseType new_value,
  memory_order success_order,
  memory_order failure_order) volatile noexcept;
 bool compare_exchange_weak(
  BaseType & old_value, BaseType new_value,
  memory_order success_order,
  memory_order failure_order) noexcept;

 operator BaseType() const volatile noexcept;
 operator BaseType() const noexcept;
};

template<typename BaseType>
bool atomic_is_lock_free(
 volatile const atomic<BaseType>*) noexcept;

template<typename BaseType>
 bool atomic_is_lock_free(const atomic<BaseType>*)
 noexcept;

template<typename BaseType>
void atomic_init(volatile atomic<BaseType>*, void*) noexcept;

template<typename BaseType>
void atomic_init(atomic<BaseType>*, void*) noexcept;

template<typename BaseType>
BaseType atomic_exchange(
 volatile atomic<BaseType>*, memory_order) noexcept;

template<typename BaseType>
BaseType atomic_exchange(
 atomic<BaseType>*, memory_order) noexcept;
```

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
 volatile atomic<BaseType>*, memory_order) noexcept;

template<typename BaseType>
BaseType atomic_exchange_explicit(
 atomic<BaseType>*, memory_order) noexcept;

template<typename BaseType>
void atomic_store(volatile atomic<BaseType>*, BaseType) noexcept;

template<typename BaseType>
void atomic_store(atomic<BaseType>*, BaseType) noexcept;

template<typename BaseType>
void atomic_store_explicit(
 volatile atomic<BaseType>*, BaseType, memory_order) noexcept;

template<typename BaseType>
void atomic_store_explicit(
 atomic<BaseType>*, BaseType, memory_order) noexcept;

template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>*) noexcept;

template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>*) noexcept;

template<typename BaseType>
BaseType atomic_load_explicit(
 volatile const atomic<BaseType>*, memory_order) noexcept;

template<typename BaseType>
BaseType atomic_load_explicit(
 const atomic<BaseType>*, memory_order) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_strong(
 volatile atomic<BaseType>*,
 BaseType * old_value, BaseType new_value) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_strong(
 atomic<BaseType>*,
 BaseType * old_value, BaseType new_value) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
 volatile atomic<BaseType>*, BaseType * old_value,
 BaseType new_value, memory_order success_order,
 memory_order failure_order) noexcept;

template<typename BaseType>
```

```
bool atomic_compare_exchange_strong_explicit(
 atomic<BaseType>*,
 BaseType * old_value, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_weak(
 volatile atomic<BaseType>*,
 BaseType * old_value, BaseType new_value) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_weak(
 atomic<BaseType>*,
 BaseType * old_value, BaseType new_value) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
 volatile atomic<BaseType>*,
 BaseType * old_value, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;

template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
 atomic<BaseType>*,
 BaseType * old_value, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
```

Примечание. Хотя функции, не являющиеся членами класса, определены как шаблоны, они могут быть предоставлены в виде набора перегруженных функций, поэтому задавать явную спецификацию аргументов шаблона не следует.

STD::ATOMIC , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует экземпляр `std::atomic` со значением, инициализированным по умолчанию.

Объявление

```
atomic() noexcept;
```

Результат

Конструирует новый объект `std::atomic` со значением, инициализированным по умолчанию. Для объектов со статическим временем жизни обеспечивается статическая инициализация.

Примечание. Если время жизни объекта `std::atomic` не статическое, то значение, которое будет иметь объект, инициализированный конструктором по умолчанию, непредсказуемо.

Исключения

Нет.

STD::ATOMIC_INIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Неатомарно сохраняет указанное значение в объекте типа `std::atomic<BaseType>`.

Объявление

```
template<typename BaseType>
void atomic_init(
 atomic<BaseType> volatile* p, BaseType v) noexcept;

template<typename BaseType>
void atomic_init(atomic<BaseType>* p, BaseType v) noexcept;
```

Результат

Неатомарно сохраняет значение `v` в `*p`. Вызов `atomic_init()` с передачей в качестве аргумента объекта `atomic<BaseType>`, который не был сконструирован по умолчанию или над которым производились какие-нибудь операции после конструирования, является неопределенным поведением.

> **Примечание**. Поскольку эта операция сохранения неатомарна, то одновременный доступ к объекту, на который указывает `p`, из другого потока (даже с помощью атомарной операции) представляет собой гонку за данными.

Исключения

Нет.

`STD::ATOMIC` , КОНВЕРТИРУЮЩИЙ КОНСТРУКТОР

Конструирует экземпляр `std::atomic` из переданного значения типа `BaseType`.

Объявление

```
constexpr atomic(BaseType b) noexcept;
```

Результат

Конструирует новый объект `std::atomic` из значения `b`. Для объектов со статическим временем жизни обеспечивается статическая инициализация.

Исключения

Нет.

`STD::ATOMIC` , КОНВЕРТИРУЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Сохраняет новое значение в `*this`.

Объявление

```
BaseType operator=(BaseType b) volatile noexcept;
BaseType operator=(BaseType b) noexcept;
```

Результат

```
return this->store(b);
```

`STD::ATOMIC::IS_LOCK_FREE` , ФУНКЦИЯ-ЧЛЕН

Сообщает, являются ли операции над `*this` свободными от блокировок.

Объявление

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

Возвращаемое значение

`true`, если операции над `*this` свободны от блокировок, иначе `false`.

Исключения

Нет.

`STD::ATOMIC_IS_LOCK_FREE` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Сообщает, являются ли операции над `*this` свободными от блокировок.

Объявление

```
template<typename BaseType>
bool atomic_is_lock_free(
 volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>* p) noexcept;
```

Результат

```
return p->is_lock_free();
```

STD::ATOMIC::LOAD , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает текущее значение объекта `std::atomic`.

Объявление

```
BaseType load(memory_order order = memory_order_seq_cst)
 const volatile noexcept;
BaseType load(
 memory_order order = memory_order_seq_cst) const noexcept;
```

Предусловия

Параметр `order` должен принимать одно из значений `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_consume` или `std::memory_order_seq_cst`.

Результат

Атомарно загружает текущее, хранящееся в `*this`.

Возвращаемое значение

Значение, хранящееся в `*this`, в точке вызова.

Исключения

Нет.

Примечание. Это атомарная операция загрузки для ячейки памяти, содержащей `*this`.

STD::ATOMIC_LOAD , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно загружает текущее значение объекта `std::atomic`.

Объявление

```
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>* p) noexcept;
```

Результат

```
return p->load();
```

STD::ATOMIC_LOAD_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно загружает текущее значение объекта `std::atomic`.

Объявление

```
template<typename BaseType>
BaseType atomic_load_explicit(
 volatile const atomic<BaseType>* p,
 memory_order order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
 const atomic<BaseType>* p, memory_order order) noexcept;
```

Результат

```
return p->load(order);
```

STD::ATOMIC::OPERATOR , ОПЕРАТОР ПРЕОБРАЗОВАНИЯ В ТИП BASETYPE

Загружает значение, хранящееся в `*this`.

Объявление

```
operator BaseType() const volatile noexcept;
operator BaseType() const noexcept;
```

Результат

```
return this->load();
```

STD::ATOMIC::STORE , ФУНКЦИЯ-ЧЛЕН

Атомарно сохраняет новое значение в объекте `atomic<BaseType>`.

Объявление

```
void store(
 BaseType new_value, memory_order order = memory_order_seq_cst)
 volatile noexcept;
void store(
 BaseType new_value, memory_order order = memory_order_seq_cst)
 noexcept;
```

Предусловия

Параметр `order` должен принимать одно из значений `std::memory_order_relaxed`, `std::memory_order_release` или `std::memory_order_seq_cst`.

Результат

Атомарно сохраняет значение `new_value` в `*this`.

Исключения

Нет.

Примечание. Это атомарная операция сохранения для ячейки памяти, содержащей `*this`.

STD::ATOMIC_STORE , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сохраняет новое значение в объекте `atomic<BaseType>`.

Объявление

```
template<typename BaseType>
void atomic_store(
 volatile atomic<BaseType>* p, BaseType new_value) noexcept;
template<typename BaseType>
void atomic_store(
 atomic<BaseType>* p, BaseType new_value) noexcept;
```

Результат

```
p->store(new_value);
```

STD::ATOMIC_STORE_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сохраняет новое значение в объекте `atomic<BaseType>`.

Объявление

```
template<typename BaseType>
void atomic_store_explicit(
 volatile atomic<BaseType>* p, BaseType new_value,
 memory_order order) noexcept;
template<typename BaseType>
void atomic_store_explicit(
 atomic<BaseType>* p, BaseType new_value,
 memory_order order) noexcept;
```

Результат

```
p->store(new_value, order);
```

STD::ATOMIC::EXCHANGE , ФУНКЦИЯ-ЧЛЕН

Атомарно сохраняет новое значение и читает старое.

Объявление

```
BaseType exchange(
BaseType new_value,
 memory_order order = memory_order_seq_cst) volatile noexcept;
```

Результат

Атомарно сохраняет значение `new_value` в `*this` и извлекает прежнее значение `*this`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

Примечание. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_EXCHANGE , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сохраняет новое значение в объекте `atomic<BaseType>` и читает предыдущее значение.

Объявление

```
template<typename BaseType>
BaseType atomic_exchange(
 volatile atomic<BaseType>* p, BaseType new_value) noexcept;
template<typename BaseType>
BaseType atomic_exchange(
 atomic<BaseType>* p, BaseType new_value) noexcept;
```

Результат

```
return p->exchange(new_value);
```

STD::ATOMIC_EXCHANGE_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сохраняет новое значение в объекте `atomic<BaseType>` и читает предыдущее значение.

Объявление

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
 volatile atomic<BaseType>* p,
 BaseType new_value, memory_order order)
 noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
 atomic<BaseType>* p,
 BaseType new_value, memory_order order) noexcept;
```

Результат

```
return p->exchange(new_value, order);
```

STD::ATOMIC::COMPARE_EXCHANGE_STRONG , ФУНКЦИЯ-ЧЛЕН

Атомарно сравнивает значение с ожидаемым и, если они равны, сохраняет новое значение. Если значения не равны, то заменяет ожидаемое значение прочитанным.

Объявление

```
bool compare_exchange_strong(
 BaseType& expected, BaseType new_value,
 memory_order order = std::memory_order_seq_cst)
 volatile noexcept;
bool compare_exchange_strong(
 BaseType& expected, BaseType new_value,
 memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_strong(
 BaseType& expected, BaseType new_value,
 memory_order success_order, memory_order failure_order)
 volatile noexcept;
bool compare_exchange_strong(
 BaseType& expected, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
```

Предусловия

Параметр `failure_order` не должен быть равен `std::memory_order_release` или `std::memory_order_acq_rel`.

Результат

Атомарно сравнивает `expected` со значением, хранящимся в `*this`, применяя побитовое сравнение, и сохраняет `new_value` в `*this`, если значения равны. В противном случае записывает в `expected` прочитанное значение.

Возвращаемое значение

`true`, если значение, хранящееся в `*this`, совпало с `expected`. В противном случае `false`.

Исключения

Нет.

> **Примечание**. Этот перегруженный вариант функции с тремя параметрами эквивалентен перегруженному варианту с четырьмя параметрами, где `success_order == order` и `failure_order == order`, с тем отличием, что если `order` равно `std::memory_order_acq_rel`, то `failure_order` равно `std::memory_order_acquire`, а если `order` равно `std::memory_order_release`, то `failure_order` равно `std::memory_order_relaxed`.

> **Примечание**. Если результат равен `true`, то это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`, с упорядочением доступа к памяти `success_order`; в противном случае это атомарная операция загрузки для ячейки памяти, содержащей `*this`, с упорядочением доступа к памяти `failure_order`.

`STD::ATOMIC_COMPARE_EXCHANGE_STRONG` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сравнивает значение с ожидаемым и, если они равны, сохраняет новое значение. Если значения не равны, то заменяет ожидаемое значение прочитанным.

Объявление

```
template<typename BaseType>
bool atomic_compare_exchange_strong(
```

```
 volatile atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
 atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value) noexcept;
```

Результат

```
return p->compare_exchange_strong(*old_value, new_value);
```

STD::ATOMIC_COMPARE_EXCHANGE_STRONG_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сравнивает значение с ожидаемым и, если они равны, сохраняет новое значение. Если значения не равны, то заменяет ожидаемое значение прочитанным.

Объявление

```
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
 volatile atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value,
 memory_order success_order, memory_order failure_order)
 noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
 atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value,
 memory_order success_order, memory_order failure_order)
 noexcept;
```

Результат

```
return p->compare_exchange_strong(
*old_value, new_value, success_order, failure_order) noexcept;
```

STD::ATOMIC::COMPARE_EXCHANGE_WEAK , ФУНКЦИЯ-ЧЛЕН

Атомарно сравнивает значение с ожидаемым и, если они равны и обновление может быть произведено атомарно, то сохраняет новое значение. Если значения не равны или обновление не может быть произведено атомарно, то заменяет ожидаемое значение прочитанным.

Объявление

```
bool compare_exchange_weak(
 BaseType& expected, BaseType new_value,
 memory_order order = std::memory_order_seq_cst)
 volatile noexcept;
bool compare_exchange_weak(
 BaseType& expected, BaseType new_value,
 memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
 BaseType& expected, BaseType new_value,
 memory_order success_order, memory_order failure_order)
 volatile noexcept;
bool compare_exchange_weak(
 BaseType& expected, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
```

Предусловия

Параметр `failure_order` не должен быть равен `std::memory_order_release` или `std::memory_order_acq_rel`.

Результат

Атомарно сравнивает `expected` со значением, хранящимся в `*this`, применяя побитовое сравнение, и сохраняет `new_value` в `*this`, если значения равны. Если значения не равны или обновление не может быть произведено атомарно, записывает в `expected` прочитанное значение.

Возвращаемое значение

`true`, если значение, хранящееся в `*this`, совпало с `expected` и `new_value` успешно сохранено в `*this`. В противном случае `false`.

Исключения

Нет.

> **Примечание**. Этот перегруженный вариант функции с тремя параметрами эквивалентен перегруженному варианту с четырьмя параметрами, где `success_order == order` и `failure_order == order`, с тем отличием, что если `order` равно `std::memory_order_acq_rel`, то `failure_order` равно `std::memory_order_acquire`, а если order равно `std::memory_order_release`, то `failure_order` равно `std::memory_order_relaxed`.

> **Примечание**. Если результат равен `true`, то это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`, с упорядочением доступа к памяти `success_order`; в противном случае это атомарная операция загрузки для ячейки памяти, содержащей `*this`, с упорядочением доступа к памяти `failure_order`.

STD::ATOMIC_COMPARE_EXCHANGE_WEAK , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сравнивает значение с ожидаемым и, если они равны и обновление может быть произведено атомарно, то сохраняет новое значение. Если значения не равны или обновление не может быть произведено атомарно, то заменяет ожидаемое значение прочитанным.

Объявление

```
template<typename BaseType>
bool atomic_compare_exchange_weak(
 volatile atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
 atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value) noexcept;
```

Результат

```
return p->compare_exchange_weak(*old_value, new_value);
```

STD::ATOMIC_COMPARE_EXCHANGE_WEAK_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно сравнивает значение с ожидаемым и, если они равны и обновление может быть произведено атомарно, то сохраняет новое значение. Если значения не равны или обновление не может быть произведено атомарно, то заменяет ожидаемое значение прочитанным.

```
template<typename BaseType>
```

```
bool atomic_compare_exchange_weak_explicit(
 volatile atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
 atomic<BaseType>* p,
 BaseType * old_value, BaseType new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
```

Результат

```
return p->compare_exchange_weak(
 *old_value, new_value, success_order, failure_order);
```

D.3.9. Специализации шаблона `std::atomic`

Предоставляются специализации шаблона `std::atomic` для целочисленных и указательных типов. Для целочисленных типов специализации обеспечивают атомарные операции сложения, вычитания и поразрядные в дополнение к имеющимся в основном шаблоне. Для указательных типов в дополнение к основному шаблону предоставляются арифметические операции над указателями.

Имеются специализации для следующих целочисленных типов:

```
std::atomic<bool>
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

а также для типа `std::atomic<T*>` при любом типе `T`.

D.3.10. Специализации `std::atomic<`*`integral-type`*`>`

Специализации `std::atomic<`*`integral-type`*`>` шаблона класса `std::atomic` дают атомарный целочисленный тип для каждого фундаментального целочисленного типа, с полным набором операций.

Ниже перечислены все такие специализации шаблона `std::atomic<>`:

```
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
```

```
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

Экземпляры этих специализаций не удовлетворяют требованиям концепций `CopyConstructible` и `CopyAssignable`, поскольку такие операции невозможно выполнить атомарно.

Определение класса

```
template<>
struct atomic<integral-type> {
 atomic() noexcept = default;
 constexpr atomic(integral-type) noexcept;
 bool operator=(integral-type) volatile noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&) = delete;
 atomic& operator=(const atomic&) volatile = delete;

 bool is_lock_free() const volatile noexcept;
 bool is_lock_free() const noexcept;

 void store(
  integral-type, memory_order = memory_order_seq_cst)
  volatile noexcept;
 void store(
  integral-type, memory_order = memory_order_seq_cst) noexcept;
 integral-type load(memory_order = memory_order_seq_cst)
  const volatile noexcept;
 integral-type load(
  memory_order = memory_order_seq_cst) const noexcept;
 integral-type exchange(
  integral-type,
  memory_order = memory_order_seq_cst) volatile noexcept;
 integral-type exchange(
  integral-type, memory_order = memory_order_seq_cst) noexcept;

 bool compare_exchange_strong(
  integral-type & old_value, integral-type new_value,
  memory_order order = memory_order_seq_cst)
  volatile noexcept;
 bool compare_exchange_strong(
  integral-type & old_value, integral-type new_value,
  memory_order order = memory_order_seq_cst) noexcept;
 bool compare_exchange_strong(
  integral-type & old_value, integral-type new_value,
  memory_order success_order, memory_order failure_order)
  volatile noexcept;
 bool compare_exchange_strong(
  integral-type & old_value, integral-type new_value,
  memory_order success_order,
  memory_order failure_order) noexcept;
```

```
bool compare_exchange_weak(
 integral-type & old_value, integral-type new_value,
 memory_order order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(
 integral-type & old_value, integral-type new_value,
 memory_order order = memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
 integral-type & old_value, integral-type new_value,
 memory_order success_order, memory_order failure_order)
 volatile noexcept;
bool compare_exchange_weak(
 integral-type & old_value, integral-type new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;

operator integral-type() const volatile noexcept;
operator integral-type() const noexcept;
integral-type fetch_add(
 integral-type, memory_order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_add(
 integral-type, memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_sub(
 integral-type, memory_order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_sub(
 integral-type, memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_and(
 integral-type, memory_order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_and(
 integral-type, memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_or(
 integral-type, memory_order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_or(
 integral-type, memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_xor(
 integral-type, memory_order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_xor(
 integral-type, memory_order = memory_order_seq_cst) noexcept;
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;

integral-type operator+=(integral-type) volatile noexcept;
integral-type operator+=(integral-type) noexcept;
integral-type operator-=(integral-type) volatile noexcept;
integral-type operator-=(integral-type) noexcept;
```

```
 integral-type operator&=(integral-type) volatile noexcept;
 integral-type operator&=(integral-type) noexcept;
 integral-type operator|=(integral-type) volatile noexcept;
 integral-type operator|=(integral-type) noexcept;
 integral-type operator^=(integral-type) volatile noexcept;
 integral-type operator^=(integral-type) noexcept;
};

bool atomic_is_lock_free(
 volatile const atomic<integral-type>*) noexcept;
bool atomic_is_lock_free(const atomic<integral-type>*) noexcept;
void atomic_init(
 volatile atomic<integral-type>*, integral-type) noexcept;
void atomic_init(atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_exchange(
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_exchange(
 atomic<integral-type>*, integral-type) noexcept;

integral-type atomic_exchange_explicit(
 volatile atomic<integral-type>*, integral-type, memory_order)
 noexcept;
integral-type atomic_exchange_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
void atomic_store(
 volatile atomic<integral-type>*, integral-type) noexcept;
void atomic_store(
 atomic<integral-type>*, integral-type) noexcept;
void atomic_store_explicit(
 volatile atomic<integral-type>*,
 integral-type, memory_order) noexcept;
void atomic_store_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
integral-type atomic_load(
 volatile const atomic<integral-type>*) noexcept;
integral-type atomic_load(
 const atomic<integral-type>*) noexcept;
integral-type atomic_load_explicit(
 volatile const atomic<integral-type>*, memory_order) noexcept;
integral-type atomic_load_explicit(
 const atomic<integral-type>*, memory_order) noexcept;
bool atomic_compare_exchange_strong(
 volatile atomic<integral-type>*, integral-type * old_value,
 integral-type new_value) noexcept;
bool atomic_compare_exchange_strong(
 atomic<integral-type>*,
 integral-type * old_value, integral-type new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
 volatile atomic<integral-type>*,
 integral-type * old_value, integral-type new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
 atomic<integral-type>*,
 integral-type * old_value, integral-type new_value,
```

```
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
 volatile atomic<integral-type>*,
 integral-type * old_value, integral-type new_value) noexcept;
bool atomic_compare_exchange_weak(
 atomic<integral-type>*,
 integral-type * old_value, integral-type new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
 volatile atomic<integral-type>*,
 integral-type * old_value, integral-type new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
 atomic<integral-type>*,
 integral-type * old_value, integral-type new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
 integral-type atomic_fetch_add(
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_add(
 atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_add_explicit(
 volatile atomic<integral-type>*, integral-type,
 memory_order) noexcept;
integral-type atomic_fetch_add_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub(
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_sub(
 atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_sub_explicit(
 volatile atomic<integral-type>*,
 integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
integral-type atomic_fetch_and(
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_and(
 atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_and_explicit(
 volatile atomic<integral-type>*,
 integral-type, memory_order) noexcept;
integral-type atomic_fetch_and_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
integral-type atomic_fetch_or(
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_or(
 atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_or_explicit(
 volatile atomic<integral-type>*,
 integral-type, memory_order) noexcept;
integral-type atomic_fetch_or_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor(
```

```
 volatile atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_xor(
 atomic<integral-type>*, integral-type) noexcept;
integral-type atomic_fetch_xor_explicit(
 volatile atomic<integral-type>*,
 integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor_explicit(
 atomic<integral-type>*, integral-type, memory_order) noexcept;
```

Те операции, которые предоставляются также основным шаблоном (см. D.3.8), имеют точно такую же семантику.

STD::ATOMIC<INTEGRAL-TYPE>::FETCH_ADD , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение и заменяет его суммой его самого и аргумента `i`.

Объявление

```
fetch_add(
 integral-type i, memory_order order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_add(
 integral-type i, memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно возвращает прежнее значение `*this` и сохраняет в `*this` значение `old-value + i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

> **Примечание**. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_ADD , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его суммой этого значения и аргумента `i`.

Объявление

```
integral-type atomic_fetch_add(
 volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_add(
 atomic<integral-type>* p, integral-type i) noexcept;
```

Результат

```
return p->fetch_add(i);
```

STD::ATOMIC_FETCH_ADD_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его суммой этого значения и аргумента `i`.

Объявление

```
integral-type atomic_fetch_add_explicit(
 volatile atomic<integral-type>* p, integral-type i,
 memory_order order) noexcept;
integral-type atomic_fetch_add_explicit(
```

```
 atomic<integral-type>* p, integral-type i,
 memory_order order) noexcept;
```

Результат

```
return p->fetch_add(i,order);
```

STD::ATOMIC<INTEGRAL-TYPE>::FETCH_SUB , ФУНКЦИЯ-ЧЛЕН

Атомарно читает значение и заменяет его разностью этого значения и аргумента `i`.

Объявление

```
integral-type fetch_sub(
 integral-type i,
 memory_order order = memory_order_seq_cst) volatile noexcept;
integral-type fetch_sub(
 integral-type i,
 memory_order order = memory_order_seq_cst) noexcept;
```

Результат

Атомарно возвращает прежнее значение `*this` и сохраняет в `*this` значение `old-value - i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

Примечание. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_SUB , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его разностью этого значения и аргумента `i`.

Объявление

```
integral-type atomic_fetch_sub(
 volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_sub(
 atomic<integral-type>* p, integral-type i) noexcept;
```

Результат

```
return p->fetch_sub(i);
```

STD::ATOMIC_FETCH_SUB_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его разностью этого значения и аргумента `i`.

Объявление

```
integral-type atomic_fetch_sub_explicit(
 volatile atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
integral-type atomic_fetch_sub_explicit(
 atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
```

Результат

```
return p->fetch_sub(i, order);
```

STD::ATOMIC<INTEGRAL-TYPE>::FETCH_AND , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение и заменяет его результатом операции поразрядное-и

между этим значением и аргументом `i`.

Объявление

```
integral-type fetch_and(
 integral-type i, memory_order order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_and(
 integral-type i, memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно возвращает прежнее значение `*this` и сохраняет в `*this` значение `old-value & i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

Примечание. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_AND , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное-и между этим значением и аргументом `i`. *Объявление*

```
integral-type atomic_fetch_and(
 volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_and(
 atomic<integral-type>* p, integral-type i) noexcept;
```

Результат

```
return p->fetch_and(i);
```

STD::ATOMIC_FETCH_AND_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное-и между этим значением и аргументом `i`.

Объявление

```
integral-type atomic_fetch_and_explicit(
 volatile atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
integral-type atomic_fetch_and_explicit(
 atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
```

Результат

```
return p->fetch_and(i,order);
```

STD::ATOMIC<INTEGRAL-TYPE>::FETCH_OR , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение и заменяет его результатом операции поразрядное-или между этим значением и аргументом `i`.

Объявление

```
integral-type fetch_or(
 integral-type i, memory_order order = memory_order_seq_cst)
volatile noexcept;
integral-type fetch_or(
```

```
 integral-type i, memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно возвращает прежнее значение `*this` и сохраняет в `*this` значение `old-value | i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

> **Примечание**. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_OR , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное-или между этим значением и аргументом `i`.

Объявление

```
integral-type atomic_fetch_or(
 volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_or(
 atomic<integral-type>* p, integral-type i) noexcept;
```

Результат

```
return p->fetch_or(i);
```

STD::ATOMIC_FETCH_OR_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное-или между этим значением и аргументом *i*.

Объявление

```
integral-type atomic_fetch_or_explicit(
 volatile atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
integral-type atomic_fetch_or_explicit(
 atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
```

Результат

```
return p->fetch_or(i, order);
```

STD::ATOMIC<INTEGRAL-TYPE>::FETCH_XOR , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение и заменяет его результатом операции поразрядное исключающее-или между этим значением и аргументом i.

Объявление

```
integral-type fetch_xor(
 integral-type i, memory_order order = memory_order_seq_cst)
 volatile noexcept;
integral-type fetch_xor(
 integral-type i, memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно возвращает прежнее значение `*this` и сохраняет в `*this` значение `old-value ^ i`.

Возвращаемое значение
Значение `*this` непосредственно перед сохранением.
Исключения
Нет.

Примечание. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_XOR , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное исключающее-или между этим значением и аргументом `i`.

Объявление

```
integral-type atomic_fetch_xor(
 volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_xor(
 atomic<integral-type>* p, integral-type i) noexcept;
```

Результат

```
return p->fetch_xor(i);
```

STD::ATOMIC_FETCH_XOR_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<integral-type>` и заменяет его результатом операции поразрядное исключающее-или между этим значением и аргументом `i`.

Объявление

```
integral-type atomic_fetch_xor_explicit(
 volatile atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
integral-type atomic_fetch_xor_explicit(
 atomic<integral-type>* p,
 integral-type i, memory_order order) noexcept;
```

Результат

```
return p->fetch_xor(i,order);
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR++ , ОПЕРАТОР ПРЕДИНКРЕМЕНТА

Атомарно инкрементирует значение, хранящееся в `*this`, и возвращает новое значение.

Объявление

```
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
```

Результат

```
return this->fetch_add(1) + 1;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR++ , ОПЕРАТОР ПОСТИНКРЕМЕНТА

Атомарно инкрементирует значение, хранящееся в `*this`, и возвращает старое значение.

Объявление

```
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
```

Результат

```
return this->fetch_add(1);
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR-- , ОПЕРАТОР ПРЕДЕКРЕМЕНТА

Атомарно декрементирует значение, хранящееся в `*this`, и возвращает новое значение.

Объявление

```
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
```

Результат

```
return this->fetch_sub(1) - 1;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR-- , ОПЕРАТОР ПОСТДЕКРЕМЕНТА

Атомарно декрементирует значение, хранящееся в `*this`, и возвращает старое значение.

Объявление

```
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;
```

Результат

```
return this->fetch_sub(1);
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR+= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно складывает значение аргумента со значением, хранящимся в `*this`, и возвращает новое значение.

Объявление

```
integral-type operator+=(integral-type i) volatile noexcept;
integral-type operator+=(integral-type i) noexcept;
```

Результат

```
return this->fetch_add(i) + i;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR-= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно вычитает значение аргумента из значения, хранящегося в `*this`, и возвращает новое значение.

Объявление

```
integral-type operator-=(integral-type i) volatile noexcept;
integral-type operator-=(integral-type i) noexcept;
```

Результат

```
return this->fetch_sub(i, std::memory_order_seq_cst) - i;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR&= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно заменяет значение, хранящееся в `*this`, результатом операции поразрядное-и между этим значением и значением аргумента и возвращает новое значение.

Объявление

```
integral-type operator&=(integral-type i) volatile noexcept;
integral-type operator&=(integral-type i) noexcept;
```

Результат

```
return this->fetch_and(i) & i;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR|= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно заменяет значение, хранящееся в `*this`, результатом операции поразрядное-или между этим значением и значением аргумента и возвращает новое значение.

Объявление

```
integral-type operator|=(integral-type i) volatile noexcept;
integral-type operator|=(integral-type i) noexcept;
```

Результат

```
return this->fetch_or(i, std::memory_order_seq_cst) | i;
```

STD::ATOMIC<INTEGRAL-TYPE>::OPERATOR^= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно заменяет значение, хранящееся в `*this`, результатом операции поразрядное исключающее-или между этим значением и значением аргумента и возвращает новое значение.

Объявление

```
integral-type operator^=(integral-type i) volatile noexcept;
integral-type operator^=(integral-type i) noexcept;
```

Результат

```
return this->fetch_xor(i, std::memory_order_seq_cst) ^ i;
```

STD::ATOMIC<T*> , ЧАСТИЧНАЯ СПЕЦИАЛИЗАЦИЯ

Частичная специализация `std::atomic<T*>` шаблона `std::atomic` предоставляет атомарный тип для любого указательного типа, с полным набором операций.

Экземпляры `std::atomic<T*>` не удовлетворяют требованиям концепций `CopyConstructible` и `CopyAssignable`, поскольку такие операции невозможно выполнить атомарно.

Определение класса

```
template<typename T>
struct atomic<T*> {
 atomic() noexcept = default;
 constexpr atomic(T*) noexcept;
 bool operator=(T*) volatile;
 bool operator=(T*);

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&) = delete;
 atomic& operator=(const atomic&) volatile = delete;

 bool is_lock_free() const volatile noexcept;
 bool is_lock_free() const noexcept;
 void store(T*, memory_order = memory_order_seq_cst)
  volatile noexcept;
 void store(T*, memory_order = memory_order_seq_cst) noexcept;
 T* load(memory_order = memory_order_seq_cst)
  const volatile noexcept;
 T* load(memory_order = memory_order_seq_cst) const noexcept;
 T* exchange(T*, memory_order = memory_order_seq_cst)
  volatile noexcept;
 T* exchange(T*, memory_order = memory_order_seq_cst) noexcept;

 bool compare_exchange_strong(
  T* & old_value, T* new_value,
  memory_order order = memory_order_seq_cst)
  volatile noexcept;
 bool compare_exchange_strong(
  T* & old_value, T* new_value,
  memory_order order = memory_order_seq_cst) noexcept;
 bool compare_exchange_strong(
  T* & old_value, T* new_value,
  memory_order success_order, memory_order failure_order)
  volatile noexcept;
 bool compare_exchange_strong(
```

```
  T* & old_value, T* new_value,
  memory_order success_order,
  memory_order failure_order) noexcept;
 bool compare_exchange_weak(
  T* & old_value, T* new_value,
  memory_order order = memory_order_seq_cst) volatile noexcept;
 bool compare_exchange_weak(
  T* & old_value, T* new_value,
  memory_order order = memory_order_seq_cst) noexcept;
 bool compare_exchange_weak(
  T* & old_value, T* new_value,
  memory_order success_order, memory_order failure_order)
  volatile noexcept;
 bool compare_exchange_weak(
  T* & old_value, T* new_value,
  memory_order success_order,
  memory_order failure_order) noexcept;

 operator T*() const volatile noexcept;
 operator T*() const noexcept;

 T* fetch_add(
  ptrdiff_t, memory_order = memory_order_seq_cst)
  volatile noexcept;
 T* fetch_add(
  ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
 T* fetch_sub(
  ptrdiff_t, memory_order = memory_order_seq_cst)
  volatile noexcept;
 T* fetch_sub(
  ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
 T* operator++() volatile noexcept;
 T* operator++() noexcept;
 T* operator++(int) volatile noexcept;
 T* operator++(int) noexcept;
 T* operator--() volatile noexcept;
 T* operator--() noexcept;
 T* operator--(int) volatile noexcept;
 T* operator--(int) noexcept;
 T* operator+=(ptrdiff_t) volatile noexcept;
 T* operator+=(ptrdiff_t) noexcept;
 T* operator-=(ptrdiff_t) volatile noexcept;
 T* operator-=(ptrdiff_t) noexcept;
};

bool atomic_is_lock_free(volatile const atomic<T*>*) noexcept;
bool atomic_is_lock_free(const atomic<T*>*) noexcept;
void atomic_init(volatile atomic<T*>*, T*) noexcept;
void atomic_init(atomic<T*>*, T*) noexcept;
T* atomic_exchange(volatile atomic<T*>*, T*) noexcept;
T* atomic_exchange(atomic<T*>*, T*) noexcept;
T* atomic_exchange_explicit(
 volatile atomic<T*>*, T*, memory_order) noexcept;
T* atomic_exchange_explicit(
 atomic<T*>*, T*, memory_order) noexcept;
void atomic_store(volatile atomic<T*>*, T*) noexcept;
void atomic_store(atomic<T*>*, T*) noexcept;
```

```
void atomic_store_explicit(
 volatile atomic<T*>*, T*, memory_order) noexcept;
void atomic_store_explicit(
 atomic<T*>*, T*, memory_order) noexcept;
T* atomic_load(volatile const atomic<T*>*) noexcept;
T* atomic_load(const atomic<T*>*) noexcept;
T* atomic_load_explicit(
 volatile const atomic<T*>*, memory_order) noexcept;
T* atomic_load_explicit(
 const atomic<T*>*, memory_order) noexcept;
bool atomic_compare_exchange_strong(
 volatile atomic<T*>*, T** old_value, T* new_value) noexcept;
bool atomic_compare_exchange_strong(
 volatile atomic<T*>*, T** old_value, T* new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
 atomic<T*>*, T** old_value, T* new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
 atomic<T*>*, T** old_value, T* new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
 volatile atomic<T*>*, T** old_value, T* new_value) noexcept;
bool atomic_compare_exchange_weak(
 atomic<T*>*, T** old_value, T* new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
 volatile atomic<T*>*,
 T** old_value, T* new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
 atomic<T*>*, T** old_value, T* new_value,
 memory_order success_order,
 memory_order failure_order) noexcept;
T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add_explicit(
 volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_add_explicit(
 atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub_explicit(
 volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub_explicit(
 atomic<T*>*, ptrdiff_t, memory_order) noexcept;
```

Те операции, которые предоставляются также основным шаблоном (см. приложение D.3.8), имеют точно такую же семантику.

`STD::ATOMIC<T*>::FETCH_ADD` , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение, заменяет его суммой этого значения и аргумента `i`, применяя стандартные правила арифметики указателей, и возвращает старое значение.

Объявление

```
T* fetch_add(
 ptrdiff_t i, memory_order order = memory_order_seq_cst)
 volatile noexcept;
```

```
T* fetch_add(
 ptrdiff_t i, memory_order order = memory_order_seq_cst) noexcept;
```

Результат

Атомарно возвращает текущее значение `*this` и сохраняет в `*this` значение `old-value + i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

> **Примечание**. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_ADD_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<T*>` и заменяет его суммой этого значения и аргумента `i`, применяя стандартные правила арифметики указателей.

Объявление

```
T* atomic_fetch_add_explicit(
 volatile atomic<T*>* p, ptrdiff_t i, memory_order order)
 noexcept;
T* atomic_fetch_add_explicit(
 atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

Результат

```
return p->fetch_add(i, order);
```

STD::ATOMIC<T*>::FETCH_SUB , ФУНКЦИЯ-ЧЛЕН

Атомарно загружает значение, заменяет его разностью этого значения и аргумента `i`, применяя стандартные правила арифметики указателей, и возвращает старое значение.

Объявление

```
T* fetch_sub(
 ptrdiff_t i, memory_order order = memory_order_seq_cst)
 volatile noexcept;
T* fetch_sub(
 ptrdiff_t i, memory_order order = memory_order_seq_cst)
 noexcept;
```

Результат

Атомарно возвращает текущее значение `*this` и сохраняет в `*this` значение `old-value - i`.

Возвращаемое значение

Значение `*this` непосредственно перед сохранением.

Исключения

Нет.

> **Примечание**. Это атомарная операция чтения-модификации-записи для ячейки памяти, содержащей `*this`.

STD::ATOMIC_FETCH_SUB , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<T*>` и заменяет его разностью этого

значения и аргумента `i`, применяя стандартные правила арифметики указателей.

Объявление

```
T* atomic_fetch_sub(
 volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_sub(atomic<T*>* p, ptrdiff_t i) noexcept;
```

Результат

```
return p->fetch_sub(i);
```

STD::ATOMIC_FETCH_SUB_EXPLICIT , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Атомарно читает значение из экземпляра `atomic<T*>` и заменяет его разностью этого значения и аргумента `i`, применяя стандартные правила арифметики указателей.

Объявление

```
T* atomic_fetch_sub_explicit(
 volatile atomic<T*>* p, ptrdiff_t i, memory_order order)
 noexcept;
T* atomic_fetch_sub_explicit(
 atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

Результат

```
return p->fetch_sub(i, order);
```

STD::ATOMIC<T*>::OPERATOR++ , ОПЕРАТОР ПРЕДИНКРЕМЕНТА

Атомарно инкрементирует значение, хранящееся в `*this`, применяя стандартные правила арифметики указателей, и возвращает новое значение.

Объявление

```
T* operator++() volatile noexcept;
T* operator++() noexcept;
```

Результат

```
return this->fetch_add(1) + 1;
```

STD::ATOMIC<T*>::OPERATOR++ , ОПЕРАТОР ПОСТИНКРЕМЕНТА

Атомарно инкрементирует значение, хранящееся в `*this`, и возвращает старое значение.

Объявление

```
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
```

Результат

```
return this->fetch_add(1);
```

STD::ATOMIC<T*>::OPERATOR-- , ОПЕРАТОР ПРЕДЕКРЕМЕНТА

Атомарно декрементирует значение, хранящееся в `*this`, применяя стандартные правила арифметики указателей, и возвращает новое значение.

Объявление

```
T* operator--() volatile noexcept;
T* operator--() noexcept;
```

Результат

```
return this->fetch_sub(1) - 1;
```

STD::ATOMIC<T*>::OPERATOR-- , ОПЕРАТОР ПОСТДЕКРЕМЕНТА

Атомарно декрементирует значение, хранящееся в `*this`, применяя стандартные правила арифметики указателей, и возвращает старое значение.

Объявление

```
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
```

Результат

```
return this->fetch_sub(1);
```

STD::ATOMIC<T*>::OPERATOR+= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно складывает значение аргумента со значением, хранящимся в `*this`, применяя стандартные правила арифметики указателей, и возвращает новое значение.

Объявление

```
T* operator+=(ptrdiff_t i) volatile noexcept;
T* operator+=(ptrdiff_t i) noexcept;
```

Результат

```
return this->fetch_add(i) + i;
```

STD::ATOMIC<T*>::OPERATOR-= , СОСТАВНОЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Атомарно вычитает значение аргумента из значения, хранящегося в `*this`, применяя стандартные правила арифметики указателей, и возвращает новое значение.

Объявление

```
T* operator-=(ptrdiff_t i) volatile noexcept;
T* operator-=(ptrdiff_t i) noexcept;
```

Результат

```
return this->fetch_sub(i) - i;
```

D.4. Заголовок <future>

В заголовке <future> объявлены средства для обработки результатов асинхронных операций, которые могли быть выполнены в другом потоке.

Содержимое заголовка

```
namespace std {
enum class future_status {
 ready, timeout, deferred
};

enum class future_errc {
 broken_promise,
 future_already_retrieved,
 promise_already_satisfied,
 no_state
};

class future_error;

const error_category& future_category();
error_code make_error_code(future_errc e);
error_condition make_error_condition(future_errc e);

template<typename ResultType>
class future;

template<typename ResultType>
class shared_future;

template<typename ResultType>
class promise;

template<typename FunctionSignature>
class packaged_task; // определение не предоставляется

template<typename ResultType, typename ... Args>
class packaged_task<ResultType (Args...)>;

enum class launch {
 async, deferred
};

template<typename FunctionType, typename ... Args>
future<result_of<FunctionType(Args...)>::type>
async(FunctionType&& func, Args&& ... args);

template<typename FunctionType, typename ... Args>
future<result_of<FunctionType(Args...)>::type>
async(std::launch policy, FunctionType&& func, Args&& ... args);
}
```

D.4.1. Шаблон класса std::future

Шаблон класса `std::future` предоставляет средства для ожидания результата асинхронной операции, начатой в другом потоке, и используется в сочетании с шаблонами классов `std::promise` и `std::packaged_task` и шаблоном функции `std::async`, которая применяется для возврата асинхронного результата. В каждый момент времени только один экземпляр `std::future` может ссылаться на данный асинхронный результат.

Экземпляры `std::future` удовлетворяют требованиям концепций `MoveConstructible` и `MoveAssignable`, но не концепций `CopyConstructible` и `CopyAssignable`.

Определение класса

```
template<typename ResultType>
class future {
public:
 future() noexcept;
 future(future&&) noexcept;
 future& operator=(future&&) noexcept;
 ~future();

 future(future const&) = delete;
 future& operator=(future const&) = delete;

 shared_future<ResultType> share();

 bool valid() const noexcept;

 см. описание get();

 void wait();

 template<typename Rep, typename Period>
 future_status wait_for(
  std::chrono::duration<Rep, Period> const& relative_time);

 template<typename Clock, typename Duration>
 future_status wait_until(
  std::chrono::time_point<Clock, Duration> const& absolute_time);
};
```

`STD::FUTURE` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::future`, с которым не связан асинхронный результат.

Объявление

```
future() noexcept;
```

Результат

Конструирует новый экземпляр `std::future`.

Постусловия

`valid()` возвращает `false`.

Исключения

Нет.

`STD::FUTURE` , ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Конструирует объект `std::future`, передавая владение асинхронным результатом от другого объекта `std::future` вновь сконструированному.

Объявление

```
future(future&& other) noexcept;
```

Результат

Конструирует новый экземпляр `std::future` путем перемещения содержимого объекта `other`.

Постусловия

Асинхронный результат, ассоциированный с объектом `other` перед вызовом конструктора, ассоциируется с вновь сконструированным объектом `std::future`. С объектом `other` больше не ассоциирован никакой асинхронный результат. Функция `this->valid()` возвращает то же значение, которое возвращала функция `other.valid()` перед вызовом конструктора. Функция `other.valid()` возвращает `false`.

Исключения

Нет.

`STD::FUTURE` , ПЕРЕМЕЩАЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Передает владение асинхронным результатом, ассоциированным с объектом `std::future`, другому объекту.

Объявление

```
future(future&& other) noexcept;
```

Результат

Передает владение асинхронным состоянием между экземплярами `std::future`.

Постусловия

Асинхронный результат, ассоциированный с объектом other перед вызовом оператора, ассоциируется с `*this`. Объект `*this` перестаёт быть владельцем своего прежнего асинхронного состояния (если оно было с ним ассоциировано), и если эта ссылка на асинхронное состояние была последней, то оно уничтожается. Функция `this->valid()` возвращает то же значение, которое возвращала функция `other`, `valid()` перед вызовом оператора. Функция `other.valid()` возвращает `false`.

Исключения

Нет.

`STD::FUTURE` , ДЕСТРУКТОР

Уничтожает объект `std::future`.

Объявление

```
~future();
```

Результат

Уничтожает `*this`. Если с `*this` была ассоциирована последняя ссылка на асинхронный результат (при условии, что с `*this` вообще что-то ассоциировано), то этот асинхронный результат уничтожается.

Исключения

Нет.

`STD::FUTURE::SHARE` , ФУНКЦИЯ-ЧЛЕН

Конструирует новый экземпляр `std::shared_future` и передаёт ему владение асинхронным результатом, ассоциированным с `*this`.

Объявление

```
shared_future<ResultType> share();
```

Результат

Эквивалентно `shared_future<ResultType>(std::move(*this))`.

Постусловия

Асинхронный результат, ассоциированный с объектом `*this` перед вызовом `share()` (если с ним что-то было ассоциировано), ассоциируется с вновь сконструированным экземпляром `std::shared_future`. Функция `this->valid()` возвращает `false`.

Исключения

Нет.

`STD::FUTURE::VALID` , ФУНКЦИЯ-ЧЛЕН

Проверяет, ассоциирован ли с экземпляром `std::future` асинхронный результат.

Объявление

```
bool valid() const noexcept;
```

Возвращаемое значение

`true`, если с `*this` ассоциирован асинхронный результат, иначе `false`.

Исключения

Нет.

`STD::FUTURE::WAIT` , ФУНКЦИЯ-ЧЛЕН

Если состояние, ассоциированное с `*this`, содержит отложенную функцию, то эта функция вызывается. В противном случае ждет, пока будет готов асинхронный результат, ассоциированный с данным экземпляром `std::future`.

Объявление

```
void wait();
```

Предусловия

`this->valid()` должно возвращать `true`.

Результат

Если ассоциированное состояние содержит отложенную функцию, то вызывает эту функцию и сохраняет возвращенное ей значение или объект-исключение в виде асинхронного результата. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с `*this`.

Исключения

Нет.

`STD::FUTURE::WAIT_FOR` , ФУНКЦИЯ-ЧЛЕН

Ждет, когда будет готов асинхронный результат, ассоциированный с данным экземпляром `std::future`, или истечет заданное время.

Объявление

```
template<typename Rep, typename Period>
future_status wait_for(
 std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

`this->valid()` должно возвращать `true`.

Результат

Если асинхронный результат, ассоциированный с `*this`, содержит отложенную функцию, полученную обращением к `std::async`, и эта функция, еще не начала исполняться, то возвращает управление немедленно без блокирования потока. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с `*this`, или до истечения времени, заданного в аргументе `relative_time`.

Возвращаемое значение

`std::future_status::deferred`, если асинхронный результат, ассоциированный с

*this, содержит отложенную функцию, полученную обращением к std::async, и эта функция, еще не начала исполняться. std::future_status::ready, если асинхронный результат, ассоциированный с *this, готов, std::future_status::timeout, если истекло время, заданное в аргументе relative_time.

Примечание. Поток может быть заблокирован на время, превышающее указанное. Если возможно, время измеряется по стабильным часам.

Исключения

Нет.

STD::FUTURE::WAIT_UNTIL , ФУНКЦИЯ-ЧЛЕН

Ждет, когда будет готов асинхронный результат, ассоциированный с данным экземпляром std::future, или наступит заданный момент времени.

Объявление

```
template<typename Clock, typename Duration>
future_status wait_until(
std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

this->valid() должно возвращать true.

Результат

Если асинхронный результат, ассоциированный с *this, содержит отложенную функцию, полученную обращением к std::async, и эта функция, еще не начала исполняться, то возвращает управление немедленно без блокирования потока. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с *this, или до момента, когда функция Clock::now() вернет время, большее или равное absolute_time.

Возвращаемое значение

std::future_status::deferred, если асинхронный результат, ассоциированный с *this, содержит отложенную функцию, полученную обращением к std::async, и эта функция, еще не начала исполняться. std::future_status::ready, если асинхронный результат, ассоциированный с *this, готов, std::future_status::timeout, если Clock::now() вернула время, большее или равное absolute_time.

Примечание. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула std::future_status::timeout, то значение, возвращенное Clock::now(), больше или равно absolute_time в точке, где поток разблокировался.

Исключения

Нет.

STD::FUTURE::GET , ФУНКЦИЯ-ЧЛЕН

Если ассоциированное состояние содержит отложенную функцию, полученную в результате обращения к std::async, то вызывает эту функцию и возвращает результат. В противном случае ждет готовности асинхронного результата, ассоциированного с экземпляром std::future, а затем либо возвращает сохраненное в нем значение, либо

возбуждает сохраненное в нем исключение.

Объявление

```
void future<void>::get();
R& future<R&>::get();
R future<R>::get();
```

Предусловия

`this->valid()` должно возвращать `true`.

Результат

Если состояние, ассоциированное с `*this`, содержит отложенную функцию, то вызывает эту функцию и возвращает результат или возбуждает хранящееся исключение. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с `*this`. Если в результате хранится исключение, возбуждает его, иначе возвращает хранящееся значение.

Возвращаемое значение

Если ассоциированное состояние содержит отложенную функцию, то возвращает результат вызова этой функции. Иначе, если `ResultType` — `void`, то функция просто возвращает управление. Если `ResultType` — `R&` для некоторого типа `R`, то возвращает хранящуюся ссылку. Иначе возвращает хранящееся значение.

Исключения

Исключение, возбужденное отложенной функцией или сохраненное в асинхронном результате (если таковое имеется).

Постусловие

```
this->valid() == false
```

D.4.2. Шаблон класса `std::shared_future`

Шаблон класса `std::shared_future` предоставляет средства для ожидания результата асинхронной операции, начатой в другом потоке, и используется в сочетании с шаблонами классов `std::promise` и `std::packaged_task` и шаблоном функции `std::async`, которая применяется для возврата асинхронного результата. В каждый момент времени ссылаться на один и тот же асинхронный результат могут несколько объектов `std::shared_future`. Экземпляры `std::shared_future` удовлетворяют требованиям концепций `CopyConstructible` и `CopyAssignable`. Разрешается также конструировать объект `std::shared_future` перемещением из объекта `std::future` с тем же самым параметром `ResultType`.

Обращения к данному экземпляру `std::shared_future` не синхронизированы. Поэтому доступ к одному экземпляру `std::shared_future` из разных потоков без внешней синхронизации *не безопасен*. Однако обращения к ассоциированному состоянию синхронизированы, поэтому несколько потоков могут *безопасно* обращаться к разным экземплярам `std::shared_future`, которые разделяют одно и то же ассоциированное состояние, без внешней синхронизации.

Определение класса

```
template<typename ResultType>
class shared_future {
public:
shared_future() noexcept;
shared_future(future<ResultType>&&) noexcept;
```

```
shared_future(shared_future&&) noexcept;
shared_future(shared_future const&);
shared_future& operator=(shared_future const&);
shared_future& operator=(shared_future&&) noexcept;
~shared_future();

bool valid() const noexcept;

см. описание get() const;

void wait() const;

template<typename Rep, typename Period>
future_status wait_for(
std::chrono::duration<Rep, Period> const& relative_time) const;

template<typename Clock, typename Duration>
future_status wait_until(
std::chrono::time_point<Clock, Duration> const& absolute_time)
const;
};
```

STD::SHARED_FUTURE , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::shared_future`, с которым не ассоциирован асинхронный результат.

Объявление

```
shared_future() noexcept;
```

Результат

Конструирует новый экземпляр `std::shared_future`.

Постусловия

Функция `valid()` вновь сконструированного экземпляра возвращает `false`.

Исключения

Нет.

STD::SHARED_FUTURE , ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Конструирует один объект `std::shared_future` из другого, передавая владение асинхронным результатом, ассоциированным со старым объектом `std::shared_future`, вновь сконструированному.

Объявление

```
shared_future(shared_future&& other) noexcept;
```

Результат

Конструирует новый экземпляр `std::shared_future`.

Постусловия

Асинхронный результат, ассоциированный с объектом `other` перед вызовом конструктора, ассоциируется с вновь сконструированным объектом `std::shared_future`. С объектом `other` больше не ассоциирован никакой асинхронный результат.

Исключения

Нет.

STD::SHARED_FUTURE , КОНСТРУКТОР MOVE-FROM- STD::FUTURE

Конструирует объект `std::shared_future` из объекта `std::future`, передавая владение

асинхронным результатом, ассоциированным с объектом `std::future`, вновь сконструированному объекту `std::shared_future`.

Объявление

```
shared_future(std::future<ResultType>&& other) noexcept;
```

Результат

Конструирует новый экземпляр `std::shared_future`.

Постусловия

Асинхронный результат, ассоциированный с объектом other перед вызовом конструктора, ассоциируется с вновь сконструированным объектом `std::shared_future`. С объектом `other` больше не ассоциирован никакой асинхронный результат.

Исключения

Нет.

STD::SHARED_FUTURE , КОПИРУЮЩИЙ КОНСТРУКТОР

Конструирует один объект `std::shared_future` из другого, так что исходный объект и копия ссылаются на асинхронный результат, ассоциированный с исходным объектом `std::shared_future`, если таковой был.

Объявление

```
shared_future(shared_future const& other);
```

Результат

Конструирует новый экземпляр `std::shared_future`.

Постусловия

Асинхронный результат, ранее ассоциированный с объектом `other` перед вызовом конструктора, теперь ассоциирован как с вновь сконструированным объектом `std::shared_future`, так и с объектом `other`.

Исключения

Нет.

STD::SHARED_FUTURE , ДЕСТРУКТОР

Уничтожает объект `std::shared_future`.

Объявление

```
~shared_future();
```

Результат

Уничтожает `*this`. Если больше не существует объекта `std::promise` или `std::packaged_task`, ассоциированного с асинхронным результатом, который ассоциирован с `*this`, и это последний экземпляр `std::shared_future`, ассоциированный с этим асинхронным результатом, то асинхронный результат уничтожается.

Исключения

Нет.

STD::SHARED_FUTURE::VALID , ФУНКЦИЯ-ЧЛЕН

Проверяет, ассоциирован ли асинхронный результат с данным экземпляром `std::shared_future`.

Объявление

```
bool valid() const noexcept;
```

Возвращаемое значение

`true`, если с `*this` ассоциирован асинхронный результат, иначе `false`.

Исключения

Нет.

STD::SHARED_FUTURE::WAIT , ФУНКЦИЯ-ЧЛЕН

Если состояние, ассоциированное с `*this`, содержит отложенную функцию, то эта функция вызывается. В противном случае ждет, когда будет готов асинхронный результат, ассоциированный с данным экземпляром `std::shared_future`.

Объявление

```
void wait() const;
```

Предусловия

`this->valid()` должна возвращать `true`.

Результат

Обращения из нескольких потоков к функциям `get()` и `wait()` экземпляров `std::shared_future`, разделяющих одно и то же ассоциированное состояние, сериализуются. Если ассоциированное состояние содержит отложенную функцию, то первое обращение к `get()` или `wait()` приводит к вызову этой функции и сохранению возвращенного ей значения или возбужденного ей исключения в асинхронном результате. Блокирует поток, пока не будет готов асинхронный результат, ассоциированный с `*this`.

Исключения

Нет.

STD::SHARED_FUTURE::WAIT_FOR , ФУНКЦИЯ-ЧЛЕН

Ждет, когда будет готов асинхронный результат, ассоциированный с данным экземпляром `std::shared_future`, или истечет заданное время.

Объявление

```
template<typename Rep, typename Period>
future_status wait_for(
std::chrono::duration<Rep, Period> const& relative_time) const;
```

Предусловия

`this->valid()` должно возвращать `true`.

Результат

Если асинхронный результат, ассоциированный с `*this`, содержит отложенную функцию, полученную обращением к `std::async`, и эта функция, еще не начала исполняться, то возвращает управление немедленно без блокирования потока. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с `*this`, или до истечения времени, заданного в аргументе `relative_time`.

Возвращаемое значение

`std::future_status::deferred`, если асинхронный результат, ассоциированный с `*this`, содержит отложенную функцию, полученную обращением к `std::async`, и эта функция, еще не начала исполняться. `std::future_status::ready`, если асинхронный результат, ассоциированный с `*this`, готов, `std::future_status::timeout`, если истекло время, заданное в аргументе `relative_time`.

Примечание. Поток может быть заблокирован на время, превышающее указанное. Если возможно, время измеряется по стабильным часам.

Исключения

Нет.

STD::SHARED_FUTURE::WAIT_UNTIL , ФУНКЦИЯ-ЧЛЕН

Ждет, когда будет готов асинхронный результат, ассоциированный с данным

экземпляром std::shared_future, или наступит заданный момент времени.

Объявление

```
template<typename Clock, typename Duration>
bool wait_until(
std::chrono::time_point<Clock, Duration> const& absolute_time) const;
```

Предусловия

this->valid() должно возвращать true.

Результат

Если асинхронный результат, ассоциированный с *this, содержит отложенную функцию, полученную обращением к std::async, и эта функция, еще не начала исполняться, то возвращает управление немедленно без блокирования потока. В противном случае блокирует поток до момента готовности асинхронного результата, ассоциированного с *this, или до момента, когда функция Clock::now() вернет время, большее или равное absolute_time.

Возвращаемое значение

std::future_status::deferred, если асинхронный результат, ассоциированный с *this, содержит отложенную функцию, полученную обращением к std::async, и эта функция, еще не начала исполняться. std::future_status::ready, если асинхронный результат, ассоциированный с *this, готов, std::future_status::timeout, если Clock::now() вернула время, большее или равное absolute_time.

> **Примечание**. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула std::future_status::timeout, то значение, возвращенное Clock::now(), больше или равно absolute_time в точке, где поток разблокировался.

Исключения

Нет.

STD::SHARED_FUTURE::GET , ФУНКЦИЯ-ЧЛЕН

Если ассоциированное состояние содержит отложенную функцию, полученную в результате обращения к std::async, то вызывает эту функцию и возвращает результат. В противном случае ждет готовности асинхронного результата, ассоциированного с экземпляром std::shared_future, а затем либо возвращает сохраненное в нем значение, либо возбуждает сохраненное в нем исключение.

Объявление

```
void shared_future<void>::get() const;
R& shared_future<R&>::get() const;
R const& shared_future<R>::get() const;
```

Предусловия

this->valid() должно возвращать true.

Результат

Обращения из нескольких потоков к функциям get() и wait() экземпляров std::shared_future, разделяющих одно и то же ассоциированное состояние, сериализуются. Если ассоциированное состояние содержит отложенную функцию, то первое обращение к get() или wait() приводит к вызову этой функции и сохранению

возвращенного ей значения или возбужденного ей исключения в асинхронном результате.

Блокирует поток, пока не будет готов асинхронный результат, ассоциированный с `*this`. Если в результате хранится исключение, возбуждает его, иначе возвращает хранящееся значение.

Возвращаемое значение

Если `ResultType` — `void`, то функция просто возвращает управление. Если `ResultType` — `R&` для некоторого типа `R`, то возвращает хранящуюся ссылку. Иначе возвращает константную ссылку на хранящееся значение.

Исключения

Хранящееся исключение, если таковое имеется.

D.4.3. Шаблон класса `std::packaged_task`

Шаблон класса `std::packaged_task` упаковывает функцию или другой допускающий вызов объект, так что при вызове функции через экземпляр `std::packaged_task` результат сохраняется в виде асинхронного результата, который может быть получен с помощью объекта `std::future`.

Экземпляры `std::packaged_task` удовлетворяют требованиям концепций `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` и `CopyAssignable`.

Определение класса

```
template<typename FunctionType>
class packaged_task; // не определен

template<typename ResultType, typename... ArgTypes>
class packaged_task<ResultType(ArgTypes...)> {
public:
packaged_task() noexcept;
packaged_task(packaged_task&&) noexcept;
~packaged_task();

packaged_task& operator=(packaged_task&&) noexcept;

packaged_task(packaged_task const&) = delete;
packaged_task& operator=(packaged_task const&) = delete;

void swap(packaged_task&) noexcept;

template<typename Callable>
explicit packaged_task(Callable&& func);

template<typename Callable, typename Allocator>
packaged_task(
std::allocator_arg_t, const Allocator&, Callable&&);

bool valid() const noexcept;
std::future<ResultType> get_future();
void operator()(ArgTypes...);
void make_ready_at_thread_exit(ArgTypes...); void reset();
};
```

STD::PACKAGED_TASK , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::packaged_task`.

Объявление

```
packaged_task() noexcept;
```

Результат

Конструирует экземпляр `std::packaged_task`, с которым не ассоциировала ни задача, ни асинхронный результат.

Исключения

Нет.

STD::PACKAGED_TASK , КОНСТРУИРОВАНИЕ ИЗ ДОПУСКАЮЩЕГО ВЫЗОВ ОБЪЕКТА

Конструирует экземпляр `std::packaged_task`, с которым ассоциированы задача и асинхронный результат.

Объявление

```
template<typename Callable>
packaged_task(Callable&& func);
```

Предусловия

Должно быть допустимо выражение `func(args...)`, где каждый элемент `args-`*`i`* в списке `args...` должен быть значением соответственного типа `ArgTypes-`*`i`* в списке `ArgTypes...`. Возвращаемое значение должно допускать преобразование в тип `ResultType`.

Результат

Конструирует экземпляр `std::packaged_task`, с которым ассоциированы еще *не готовый* асинхронный результат типа `ResultType` и задача типа `Callable`, полученная копированием `func`.

Исключения

Исключение типа `std::bad_alloc`, если конструктор не смог выделить память для асинхронного результата. Любое исключение, возбуждаемое копирующим или перемещающим конструктором `Callable`.

STD::PACKAGED_TASK , КОНСТРУИРОВАНИЕ ИЗ ДОПУСКАЮЩЕГО ВЫЗОВ ОБЪЕКТА С РАСПРЕДЕЛИТЕЛЕМ

Конструирует экземпляр `std::packaged_task`, с которым ассоциированы задача и асинхронный результат, применяя предоставленный распределитель для выделения памяти под асинхронный результат и задачу

Объявление

```
template<typename Allocator, typename Callable>
packaged_task(
std::allocator_arg_t, Allocator const& alloc, Callable&& func);
```

Предусловия

Должно быть допустимо выражение `func(args...)`, где каждый элемент `args-`*`i`* в списке `args...` должен быть значением соответственного типа `ArgTypes-`*`i`* в списке `ArgTypes...`. Возвращаемое значение должно допускать преобразование в тип `ResultType`.

Результат

Конструирует экземпляр `std::packaged_task`, с которым ассоциированы еще *не готовый* асинхронный результат типа `ResultType` и задача типа `Callable`, полученная копированием `func`. Память под асинхронный результат и задачу выделяется с помощью распределителя `alloc` или его копии.

Исключения

Любое исключение, возбуждаемое распределителем в случае неудачной попытки выделить память под асинхронный результат или задачу. Любое исключение, возбуждаемое копирующим или перемещающим конструктором `Callable`.

`std::packaged_task`, ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Конструирует один объект `std::packaged_task` из другого, передавая владение асинхронным результатом и задачей, ассоциированными с объектом `other`, вновь сконструированному.

Объявление

```
packaged_task(packaged_task&& other) noexcept;
```

Результат

Конструирует новый экземпляр `std::packaged_task`.

Постусловия

Асинхронный результат и задача, которые были ассоциированы с объектом `other` до вызова конструктора, ассоциируются со вновь сконструированным объектом `std::packaged_task`. С объектом `other` больше не связан никакой асинхронный результат.

Исключения

Нет.

`std::packaged_task`, ПЕРЕМЕЩАЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Передает владение ассоциированным асинхронным результатом от одного объекта `std::packaged_task` другому.

Объявление

```
packaged_task& operator=(packaged_task&& other) noexcept;
```

Результат

Передает владение асинхронным результатом и задачей, ассоциированными с объектом `other`, объекту `*this` и отбрасывает ранее ассоциированный асинхронный результат, как если бы было выполнено предложение `std::packaged_task(other).swap(*this)`.

Постусловия

Асинхронный результат и задача, которые были ассоциированы с объектом `other` до вызова перемещающего оператора присваивания, ассоциируются с `*this`. С объектом `other` больше не связан никакой асинхронный результат.

Возвращаемое значение

`*this`

Исключения

Нет.

`std::packaged_task::swap`, ФУНКЦИЯ-ЧЛЕН

Обменивает владение асинхронными результатами, ассоциированными с двумя объектами `std::packaged_task`.

Объявление

```
void swap(packaged_task& other) noexcept;
```

Результат

Обменивает владение асинхронными результатами и задачами, ассоциированными с объектами `other` и `*this`.

Постусловия

Асинхронный результат и задача, которые были ассоциированы с объектом `other` до вызова `swap` (если таковые действительно были), ассоциируются с `*this`. Асинхронный

результат и задача, которые были ассоциировать с объектом `*this` до вызова `swap` (если таковые действительно были), ассоциируются с `other`.

Исключения

Нет.

`STD::PACKAGED_TASK` , ДЕСТРУКТОР

Уничтожает объект `std::packaged_task`.

Объявление

```
~packaged_task();
```

Результат

Уничтожает `*this`. Если с `*this` ассоциирован асинхронный результат и в этом результате не хранится задача или исключение, то результат становится *готов*, причем в него помещается исключение `std::future_error` с кодом ошибки `std::future_errc::broken_promise`.

Исключения

Нет.

`STD::PACKAGED_TASK::GET_FUTURE` , ФУНКЦИЯ-ЧЛЕН

Извлекает экземпляр `std::future` для асинхронного результата, ассоциированного с `*this`.

Объявление

```
std::future<ResultType> get_future();
```

Предусловия

С `*this` ассоциирован асинхронный результат.

Возвращаемое значение

Экземпляр `std::future` для асинхронного результата, ассоциированного с `*this`.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`, если объект `std::future` уже был получен для этого асинхронного результата с помощью предшествующего обращения к `get_future()`.

`STD::PACKAGED_TASK::RESET` , ФУНКЦИЯ-ЧЛЕН

Ассоциирует экземпляр `std::packaged_task` с новым асинхронным результатом для той же задачи.

Объявление

```
void reset();
```

Предусловия

С `*this` ассоциирована асинхронная задача.

Результат

Эквивалентно `*this = packaged_task(std::move(f))`, где `f` — хранимая задача, ассоциированная с `*this`.

Исключения

Исключение типа `std::bad_alloc`, если не удалось выделить память для нового асинхронного результата.

`STD::PACKAGED_TASK::VALID` , ФУНКЦИЯ-ЧЛЕН

Проверяет, ассоциированы ли с `*this` задача и асинхронный результат.

Объявление

```
bool valid() const noexcept;
```

Возвращаемое значение

`true`, если с `*this` ассоциированы задача и асинхронный результат, иначе `false`.

Исключения

Нет.

`std::packaged_task::operator()` , ОПЕРАТОР ВЫЗОВА

Вызывает задачу, ассоциированную с экземпляром `std::packaged_task`, и сохраняет возвращенное ей значение или исключение в ассоциированном асинхронном результате.

Объявление

```
void operator()(ArgTypes... args);
```

Предусловия

С `*this` ассоциирована задача.

Результат

Вызывает ассоциированную задачу, как если бы было выполнено предложение `INVOKE(func, args...)`. Если вызов завершается нормально, то сохраняет возвращенное значение в асинхронном результате, ассоциированном с `*this`. Если задача возбуждает исключение, то сохраняет это исключение в асинхронном результате, ассоциированном с `*this`.

Постусловия

Асинхронный результат, ассоциированный с `*this`, готов и содержит значение или исключение. Все потоки, ожидающие асинхронного результата, разблокируются.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение.

Синхронизация

Успешное обращение к оператору вызова синхронизируется-с обращением к `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которое извлекает хранимое значение или исключение.

`std::packaged_task::make_ready_at_thread_exit` , ФУНКЦИЯ-ЧЛЕН

Вызывает задачу, ассоциированную с экземпляром `std::packaged_task`, и сохраняет возвращенное ей значение или исключение в ассоциированном асинхронном результате, но не делает этот результат готовым раньше момента завершения потока.

Объявление

```
void make_ready_at_thread_exit(ArgTypes... args);
```

Предусловия

С `*this` ассоциирована задача.

Результат

Вызывает ассоциированную задачу, как если бы было выполнено предложение `INVOKE(func, args...)`. Если вызов завершается нормально, то сохраняет возвращенное значение в асинхронном результате, ассоциированном с `*this`. Если задача возбуждает исключение, то сохраняет это исключение в асинхронном результате, ассоциированном с `*this`. Планирует перевод ассоциированного асинхронного результата в состояние готовности в момент завершения потока.

Постусловия

Асинхронный результат, ассоциированный с `*this`, содержит значение или исключение, но не является готовым до завершения текущего потока. Все потоки, ожидающие

асинхронного результата, будут разблокированы, когда текущий поток завершится.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение. Исключение типа `std::future_error` с кодом ошибки `std::future_errc::no_state`, если с `*this` не ассоциировано асинхронное состояние.

Синхронизация

Завершение потока, в котором была успешно вызвала функция `make_ready_at_thread_exit()`, синхронизируется-с обращением к `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которое извлекает хранимое значение или исключение.

D.4.4. Шаблон класса `std::promise`

Шаблон класса `std::promise` предоставляет средства для установки асинхронного результата, который может быть получен в другом потоке с помощью экземпляра `std::future`.

Параметр `ResultType` — это тип значения, сохраняемого в асинхронном результате.

Объект `std::future`, ассоциированный с асинхронным результатом конкретного экземпляра `std::promise`, можно получить путем обращения к функции-члену `get_future()`. В асинхронный результат записывается либо значение типа `ResultType` функцией-членом `set_value()`, либо исключение функцией-членом `set_exception()`.

Экземпляры `std::promise` удовлетворяют требованиям концепций `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` **или** `CopyAssignable`.

Определение класса

```
template<typename ResultType>
class promise {
public:
promise();
promise(promise&&) noexcept;
~promise();
promise& operator=(promise&&) noexcept;

template<typename Allocator>
promise(std::allocator_arg_t, Allocator const&);

promise(promise const&) = delete;
promise& operator=(promise const&) = delete;

void swap(promise&) noexcept;
std::future<ResultType> get_future();

void set_value(see description);
void set_exception(std::exception_ptr p);
};
```

`STD::PROMISE` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::promise`.

Объявление

```
promise();
```

Результат

Конструирует экземпляр `std::promise`, с которым ассоциировал неготовый асинхронный результат типа `ResultType`.

Исключения

Исключение типа `std::bad_alloc`, если конструктор не смог выделить память для асинхронного результата.

STD::PROMISE , КОНСТРУКТОР С РАСПРЕДЕЛИТЕЛЕМ

Конструирует экземпляр std::promise, применяя предоставленный распределитель для выделения памяти под ассоциированный асинхронный результат.

Объявление

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

Результат

Конструирует экземпляр `std::promise`, с которым ассоциировал неготовый асинхронный результат типа `ResultType`. Память под асинхронный результат выделяется с помощью распределителя `alloc`.

Исключения

Любое исключение, возбуждаемое распределителем в случае неудачной попытки выделить память под асинхронный результат.

STD::PROMISE , ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Конструирует один объект `std::promise` из другого, передавая владение асинхронным результатом от объекта `other` вновь сконструированному.

Объявление

```
promise(promise&& other) noexcept;
```

Результат

Конструирует новый экземпляр `std::promise`.

Постусловия

Асинхронный результат, который был ассоциирован с объектом `other` до вызова конструктор, ассоциируется с вновь сконструированным объектом `std::promise`. С объектом `other` больше не ассоциирован никакой асинхронный результат.

Исключения

Нет.

STD::PROMISE , ПЕРЕМЕЩАЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Передает владение асинхронным результатом, ассоциированным с объектом `std::promise`, другому объекту.

Объявление

```
promise& operator=(promise&& other) noexcept;
```

Результат

Передает владение асинхронным результатом, ассоциированным с `*this`. Если с `*this` уже был ассоциирован асинхронный результат, то результат становится *готов*, причем в него помещается исключение `std::future_error` с кодом ошибки `std::future_errc::broken_promise`.

Постусловия

Асинхронный результат, который был ассоциирован с объектом `other` до вызова перемещающего оператора присваивания, ассоциируется с `*this`. С объектом `other` больше

не ассоциирован никакой асинхронный результат.

Возвращаемое значение

`*this`

Исключения

Нет.

`STD::PROMISE::SWAP` , ФУНКЦИЯ-ЧЛЕН

Обменивает владение асинхронными результатами, ассоциированными с двумя объектами `std::promise`.

Объявление

```
void swap(promise& other);
```

Результат

Обменивает владение асинхронными результатами, ассоциированными с объектами `other` и `*this`.

Постусловия

Асинхронный результат, который был ассоциирован с объектом `other` до вызова `swap` (если таковой действительно был), ассоциируется с `*this`. Асинхронный результат, который был ассоциирован с объектом `*this` до вызова `swap` (если таковой действительно был), ассоциируется с `other`.

Исключения

Нет.

`STD::PROMISE` , ДЕСТРУКТОР

Уничтожает объект `std::promise`.

Объявление

```
~promise();
```

Результат

Уничтожает `*this`. Если с `*this` ассоциирован асинхронный результат и в этом результате не хранится задача или исключение, то результат становится *готов*, причем в него помещается исключение `std::future_error` с кодом ошибки `std::future_errc::broken_promise`.

Исключения

Нет.

`STD::PROMISE::GET_FUTURE` , ФУНКЦИЯ-ЧЛЕН

Извлекает экземпляр `std::future` для асинхронного результата, ассоциированного с `*this`.

Объявление

```
std::future<ResultType> get_future();
```

Предусловия

С `*this` ассоциировал асинхронный результат.

Возвращаемое значение

Экземпляр `std::future` для асинхронного результата, ассоциированного с `*this`.

`Исключения`

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`, если объект `std::future` уже был получен для этого асинхронного результата с помощью предшествующего обращения к `get_future()`.

`STD::PROMISE::SET_VALUE` , ФУНКЦИЯ-ЧЛЕН

Сохраняет значение в асинхронном результате, ассоциированном с `*this`.

Объявление

```
void promise<void>::set_value();
void promise<R&>::set_value(R& r);
void promise<R>::set_value(R const& r);
void promise<R>::set_value(R&& r);
```

Предусловия

С `*this` ассоциирован асинхронный результат.

Результат

Сохраняет `r` в асинхронном результате, ассоциированном с `*this`, если `ResultType` — не `void`.

Постусловия

Асинхронный результат, ассоциированный с `*this`, готов и содержит значение. Все потоки, ожидающие асинхронного результата, разблокируются.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение. Любое исключение, возбужденное копирующим или перемещающим конструктором `r`.

Синхронизация

Обращения к `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` сериализуются. Успешное обращение к `set_value()` происходит-раньше обращения к функции `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которая извлекает сохраненное значение.

`STD::PROMISE::SET_VALUE_AT_THREAD_EXIT` , ФУНКЦИЯ-ЧЛЕН

Сохраняет значение в асинхронном результате, ассоциированном с `*this`, но не делает этот результат готовым раньше момента завершения потока.

Объявление

```
void promise<void>::set_value_at_thread_exit();
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<R>::set_value_at_thread_exit(R const& r);
void promise<R>::set_value_at_thread_exit(R&& r);
```

Предусловия

С `*this` ассоциирован асинхронный результат.

Результат

Сохраняет `r` в асинхронном результате, ассоциированном с `*this`, если `ResultType` — не `void`. Помечает, что в асинхронном результате хранится значение. Планирует перевод ассоциированного асинхронного результата в состояние готовности в момент завершения потока.

Постусловия

Асинхронный результат, ассоциированный с `*this`, содержит значение, но не является готовым до завершения текущего потока. Все потоки, ожидающие асинхронного результата, будут разблокированы, когда текущий поток завершится.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение. Любое исключение, возбужденное копирующим или перемещающим конструктором `r`.

Синхронизация

Обращения к `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` сериализуются. Успешное обращение к `set_value()` происходит-раньше обращения к функции `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которая извлекает сохраненное значение.

`STD::PROMISE::SET_EXCEPTION` , ФУНКЦИЯ-ЧЛЕН КЛАССА

Сохраняет исключение в асинхронном результате, ассоциированном с `*this`.

Объявление

```
void set_exception(std::exception_ptr e);
```

Предусловия

С `*this` ассоциирован асинхронный результат. `(bool)e` равно `true`.

Результат

Сохраняет `e` в асинхронном результате, ассоциированном с `*this`.

Постусловия

Асинхронный результат, ассоциированный с `*this`, готов и содержит исключение. Все потоки, ожидающие асинхронного результата, разблокируются.

Исключения

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение.

Синхронизация

Обращения к `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` сериализуются. Успешное обращение к `set_value()` происходит-раньше обращения к функции `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которая извлекает сохраненное исключение.

`STD::PROMISE::SET_EXCEPTION_AT_THREAD_EXIT`, ФУНКЦИЯ-ЧЛЕН

Сохраняет исключение в асинхронном результате, ассоциированном с `*this`, но не делает этот результат готовым раньше момента завершения потока.

Объявление

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

Предусловия

С `*this` ассоциирован асинхронный результат, `(bool)e` равно `true`.

Результат

Сохраняет `e` в асинхронном результате, ассоциированном с `*this`. Планирует перевод ассоциированного асинхронного результата в состояние готовности в момент завершения потока.

Постусловия

Асинхронный результат, ассоциированный с `*this`, содержит исключение, но не является готовым до завершения текущего потока. Все потоки, ожидающие асинхронного результата, будут разблокированы, когда текущий поток завершится. *Исключения*

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в асинхронном результате уже находится значение или исключение.

Синхронизация

Обращения к `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и

`set_exception_at_thread_exit()` сериализуются. Успешное обращение к `set_value()` происходит-раньше обращения к функции `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, которая извлекает сохраненное исключение.

D.4.5. Шаблон функции `std::async`

Шаблон функции `std::async` дает простой способ выполнить автономную асинхронную задачу с использованием доступного аппаратного параллелизма. Обращение к `std::async` возвращает объект `std::future`, который содержит результат задачи. В зависимости от политики запуска задача выполняется либо асинхронно в отдельном потоке, либо синхронно в том потоке, который вызвал функции-члены `wait()` или `get()` объекта `std::future`.

Объявление

```
enum class launch {
async, deferred
};

template<typename Callable, typename... Args>
future<result_of<Callable(Args...)>::type>
async(Callable&& func, Args&& ... args);

template<typename Callable, typename ... Args>
future<result_of<Callable(Args...)>::type>
async(launch policy, Callable&& func, Args&& ... args);
```

Предусловия

Выражение `INVOKE(func, args)` допустимо для переданных значений `func` и `args`. Тип `Callable` и все члены `Args` удовлетворяют требованиям концепции `MoveConstructible`.

Результат

Конструирует копии `func` и `args...` во внутренней памяти (далее обозначаются `fff` и `xyz...` соответственно).

Если `policy` равно `std::launch::async`, то вызывает функцию `INVOKE(fff, xyz...)` в отдельном потоке. Возвращенный объект `std::future` становится *готов*, когда этот поток завершится, и будет содержать либо возвращенное функцией значение, либо возбужденное ей исключение. Деструктор последнего будущего объекта, ассоциированного с асинхронным состоянием возвращенного объекта `std::future`, блокирует поток, пока будущий результат не будет *готов*.

Если `policy` равно `std::launch::deferred`, то `fff` и `xyz...` сохраняются в возвращенном объекте `std::future` как отложенный вызов функции. При первом обращении к функции-члену `wait()` или `get()` будущего результата, который разделяет то же самое ассоциированное состояние, функция `INVOKE(fff, xyz...)` синхронно вызывается в потоке, который обратился к `wait()` или `get()`.

В ответ на вызов функции `get()` этого объекта `std::future` либо возвращается значение, полученное от `INVOKE(fff, xyz...)`, либо возбуждается исключение, которое имело место в этой функции.

Если `policy` равно `std::launch::async | std::launch::deferred` или аргумент `policy` опущен, то поведение такое же, как если бы была задана политика `std::launch::async` или `std::launch::deferred`. Реализация сама выбирает нужное

поведение при каждом вызове, чтобы в максимальной степени задействовать доступный аппаратный параллелизм, не вызывая при этом превышения лимита.

В любом случае функция `std::async` возвращает управление немедленно.

Синхронизация

Завершение вызова функции происходит-раньше успешного возврата из функций `wait()`, `get()`, `wait_for()` и `wait_until()` любого экземпляра `std::future` или `std::shared_future`, который ссылается на то же ассоциированное состояние, что и объект `std::future`, возвращенный функцией `std::async`. Если `policy` равно `std::launch::async`, то завершение потока, в котором имел место вызов `std::async`, также происходит-раньше успешного возврата из этих функций.

Исключения

`std::bad_alloc`, если не удалось выделить внутреннюю память или `std::future_error`, если не удалось добиться желаемого эффекта, или исключение, возбужденное в ходе конструирования `fff` или `xyz...`

D.5. Заголовок <mutex>

В заголовке <mutex> объявлены средства, обеспечивающие взаимное исключение: типы мьютексов и блокировок, различные функции и механизм, гарантирующий, что некая операция выполнена ровно один раз.

Содержимое заголовка

```
namespace std {
class mutex;
class recursive_mutex;
class timed_mutex;
class recursive_timed_mutex;

struct adopt_lock_t;
struct defer_lock_t;
struct try_to_lock_t;

constexpr adopt_lock_t adopt_lock{};
constexpr defer_lock_t defer_lock{};
constexpr try_to_lock_t try_to_lock{};

template<typename LockableType>
class lock_guard;

template<typename LockableType>
class unique_lock;

template<typename LockableType1, typename... LockableType2>
void lock(LockableType1& m1, LockableType2& m2...);

template<typename LockableType1, typename... LockableType2>
int try_lock(LockableType1& m1, LockableType2& m2...);

struct once_flag;

template<typename Callable, typename... Args>
void call_once(once_flag& flag, Callable func, Args args...);
}
```

D.5.1. Класс std::mutex

Класс std::mutex предоставляет базовые средства взаимного исключения и синхронизации потоков, применяемые для защиты разделяемых данных. Перед тем как обращаться к данным, защищаемым мьютексом, этот мьютекс необходимо *захватить* (или *заблокировать*), вызвав функцию lock() или try_lock(). В любой момент времени удерживать мьютекс может только один поток; если другой поток попытается захватить тот же мьютекс, то функция try_lock() вернет ошибку, а функция lock() приостановит выполнение потока. Закончив операции над разделяемыми данными, поток должен вызвать функцию unlock(), чтобы освободить мьютекс и дать другим потокам возможность захватить его.

Экземпляр `std::mutex` удовлетворяет требованиям концепции `Lockable`.

Определение класса

```
class mutex {
public:
 mutex(mutex const&)=delete;
 mutex& operator=(mutex const&)=delete;

 constexpr mutex() noexcept;
 ~mutex();

 void lock();
 void unlock();
 bool try_lock();
};
```

`STD::MUTEX` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::mutex`.

Объявление

```
constexpr mutex() noexcept;
```

Результат

Конструирует экземпляр `std::mutex`.

Постусловия

Вновь сконструированный объект `std::mutex` первоначально не захвачен.

Исключения

Нет.

`STD::MUTEX` , ДЕСТРУКТОР

Уничтожает объект `std::mutex`.

Объявление

```
~mutex();
```

Предусловия

Объект `*this` не должен быть захвачен.

Результат

Уничтожает `*this`.

Исключения

Нет.

`STD::MUTEX::LOCK` , ФУНКЦИЯ-ЧЛЕН

Захватывает объект `std::mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`.

Результат

Блокирует текущий поток, пока мьютекс `*this` не будет захвачен.

Постусловия

`*this` захвачен текущим потоком.

Исключения

Исключение типа `std::system_error` в случае ошибки.

`STD::MUTEX::TRY_LOCK` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`.

Результат

Пытается захватить объект `std::mutex` для текущего потока без блокирования.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

> **Примечание**. Функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`.

`STD::MUTEX::UNLOCK` , ФУНКЦИЯ-ЧЛЕН

Освобождает объект `std::mutex`, удерживаемый текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать мьютекс `*this`.

Результат

Освобождает мьютекс `std::mutex`, удерживаемый текущим потоком.

Если другие потоки были блокированы в ожидании `*this`, то один из них разблокируется.

Постусловия

`*this` не захвачен вызывающим потоком.

Исключения

Нет.

D.5.2. Класс `std::recursive_mutex`

Класс `std::recursive_mutex` предоставляет базовые средства взаимного исключения и синхронизации потоков, применяемые для защиты разделяемых данных. Перед тем как обращаться к данным, защищаемым мьютексом, этот мьютекс необходимо *захватить* (или *заблокировать*), вызвав функцию `lock()` или `try_lock()`. В любой момент времени удерживать мьютекс может только один поток; если другой поток попытается захватить тот же мьютекс, то функция `try_lock()` вернет ошибку, а функция `lock()` приостановит выполнение потока. Закончив операции над разделяемыми данными, поток должен вызвать функцию `unlock()`, чтобы освободить мьютекс и дать другим потокам возможность захватить его.

Этот мьютекс называется *рекурсивным*, потому что поток, удерживающий мьютекс типа `std::recursive_mutex`, может снова обратиться к функции `lock()` или `try_lock()`, что приведёт к увеличению счетчика захватов. Никакой другой поток не сможет захватить этот

мьютекс, пока владеющий им поток не вызовет функцию `unlock` столько раз, сколько было успешных вызовов `lock()` или `try_lock()`.

Экземпляр `std::recursive_mutex` удовлетворяет требованиям концепции `Lockable`.

Определение класса

```
class recursive_mutex {
public:
 recursive_mutex(recursive_mutex const&) = delete;
 recursive_mutex& operator=(recursive_mutex const&) = delete;
 recursive_mutex() noexcept;
 ~recursive_mutex();

 void lock();
 void unlock();
 bool try_lock() noexcept;
};
```

STD::RECURSIVE_MUTEX , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::recursive_mutex`.

Объявление

```
recursive_mutex() noexcept;
```

Результат

Конструирует экземпляр `std::recursive_mutex`.

Постусловия

Вновь сконструированный объект `std::recursive_mutex` первоначально не захвачен.

Исключения

Исключение типа `std::system_error`, если не удалось создать экземпляр `std::recursive_mutex`.

STD::RECURSIVE_MUTEX , ДЕСТРУКТОР

Уничтожает объект `std::recursive_mutex`.

Объявление

```
~recursive_mutex();
```

Предусловия

Объект `*this` не должен быть захвачен.

Результат

Уничтожает `*this`.

Исключения

Нет.

STD::RECURSIVE_MUTEX::LOCK , ФУНКЦИЯ-ЧЛЕН

Захватывает объект `std::recursive_mutex` для текущего потока.

Объявление

```
void lock();
```

Результат

Блокирует текущий поток, пока мьютекс `*this` не будет захвачен.

Постусловия

`*this` захвачен текущим потоком. Если вызывающий поток уже удерживал `*this`, то счетчик захватов увеличивается на единицу.

Исключения

Исключение типа `std::system_error` в случае ошибки.

STD::RECURSIVE_MUTEX::TRY_LOCK , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::recursive_mutex` для текущего потока.

Объявление

```
bool try_lock() noexcept;
```

Результат

Пытается захватить объект `std::recursive_mutex` для текущего потока без блокирования.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

> **Примечание**. Если вызывающий поток уже удерживал `*this`, то функция возвращает `true`, и счетчик захватов `*this` текущим потоком увеличивается на единицу. Если текущий поток не удерживал `*this`, то функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`.

`STD::RECURSIVE_MUTEX::UNLOCK` , ФУНКЦИЯ-ЧЛЕН

Освобождает объект `std::recursive_mutex`, удерживаемый текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать мьютекс `*this`.

Результат

Освобождает мьютекс `std::recursive_mutex`, удерживаемый текущим потоком. Если это последний захват `*this` данным потоком, и другие потоки были блокированы в ожидании `*this`, то один из них разблокируется.

Постусловия

Количество захватов `*this` вызывающим потоком, уменьшается на единицу.

Исключения

Нет.

D.5.3. Класс `std::timed_mutex`

Класс `std::timed_mutex` предоставляет поддержку блокировок с таймаутами сверх базовых средств взаимного исключения и синхронизации, предоставляемых классом `std::mutex`. Перед тем как обращаться к данным, защищаемым мьютексом, этот мьютекс необходимо *захватить* (или *заблокировать*), вызвав функцию `lock()`, `try_lock()`, `try_lock_for()` или `try_lock_until()`. Если мьютекс уже захвачен другим потоком, то функция `try_lock()` вернет ошибку, функция `lock()` приостановит выполнение потока до получения мьютекса, а функции `try_lock_for()` и `try_lock_until()` приостановят выполнение потока до получения мьютекса или истечения таймаута. Закончив операции над разделяемыми данными, поток должен вызвать функцию `unlock()` (вне зависимости от того, какой функцией мьютекс был захвачен), чтобы освободить мьютекс и дать другим потокам

возможность захватить его.

Экземпляр `std::timed_mutex` удовлетворяет требованиям концепции `TimedLockable`.

Определение класса

```
class timed_mutex {
public:
 timed_mutex(timed_mutex const&)=delete;
 timed_mutex& operator=(timed_mutex const&)=delete;
 timed_mutex();
 ~timed_mutex();

 void lock();
 void unlock();
 bool try_lock();

 template<typename Rep, typename Period>
 bool try_lock_for(
  std::chrono::duration<Rep, Period> const& relative_time);

 template<typename Clock, typename Duration>
 bool try_lock_until(
  std::chrono::time_point<Clock, Duration> const& absolute_time);
};
```

`STD::TIMED_MUTEX` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::timed_mutex`.

Объявление

```
timed_mutex();
```

Результат

Конструирует экземпляр `std::timed_mutex`.

Постусловия

Вновь сконструированный объект `std::timed_mutex` первоначально не захвачен.

Исключения

Исключение типа `std::system_error`, если не удалось создать экземпляр `std::timed_mutex`.

`STD::TIMED_MUTEX` , ДЕСТРУКТОР

Уничтожает объект `std::timed_mutex`.

Объявление

```
~timed_mutex();
```

Предусловия

Объект `*this` не должен быть захвачен.

Результат

Уничтожает `*this`.

Исключения

Нет.

`STD::TIMED_MUTEX::LOCK` , ФУНКЦИЯ-ЧЛЕН

Захватывает объект `std::timed_mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`

Результат

Блокирует текущий поток, пока мьютекс `*this` не будет захвачен.

Постусловия

`*this` захвачен текущим потоком.

Исключения

Исключение типа `std::system_error` в случае ошибки.

`STD::TIMED_MUTEX::TRY_LOCK` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::timed_mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`.

Результат

Пытается захватить объект `std::timed_mutex` для текущего потока без блокирования.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

Примечание. Функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`.

`STD::TIMED_MUTEX::TRY_LOCK_FOR` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::timed_mutex` для текущего потока.

Объявление

```
template<typename Rep, typename Period>
bool try_lock_for(
 std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`.

Результат

Пытается захватить объект `std::timed_mutex` для текущего потока в течение времени, заданного аргументом `relative_time`. Если `relative_time.count()` равно нулю или отрицательно, то функция возвращается немедленно, как если бы это был вызов `try_lock()`. В противном случае вызывающий поток приостанавливается до получения мьютекса или до истечения времени, заданного аргументом `relative_time`.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

Примечание. Функция может не захватить мьютекс (и вернуть `false`), даже

если никакой другой поток не удерживает `*this`. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

STD::TIMED_MUTEX::TRY_LOCK_UNTIL , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::timed_mutex` для текущего потока.

Объявление

```
template<typename Clock, typename Duration>
bool try_lock_until(
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

Вызывающий поток не должен удерживать мьютекс `*this`.

Результат

Пытается захватить объект `std::timed_mutex` для текущего потока, пока не наступит момент времени, заданный аргументом `absolute_time`. Если в момент вызова `absolute_time <= Clock::now()`, то функция возвращается немедленно, как если бы это был вызов `try_lock()`. В противном случае вызывающий поток приостанавливается до получения мьютекса или до наступления момента времени, большего или равного `absolute_time`.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

Примечание. Функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула `false`, то значение, возвращенное `Clock::now()`, больше или равно `absolute_time` в точке, где поток разблокировался.

STD::TIMED_MUTEX::UNLOCK , ФУНКЦИЯ-ЧЛЕН

Освобождает объект `std::timed_mutex`, удерживаемый текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать мьютекс `*this`.

Результат

Освобождает мьютекс `*this`, удерживаемый текущим потоком. Если другие потоки были блокированы в ожидании `*this`, то один из них разблокируется.

Постусловия

`*this` не захвачен вызывающим потоком.

Исключения

Нет.

D.5.4. Класс `std::recursive_timed_mutex`

Класс `std::recursive_timed_mutex` предоставляет поддержку блокировок с таймаутами сверх базовых средств взаимного исключения и синхронизации, предоставляемых классом `std::recursive_mutex`. Перед тем как обращаться к данным, защищаемым мьютексом, этот мьютекс необходимо *захватить* (или *заблокировать*), вызвав функцию `lock()`, `try_lock()`, `try_lock_for()` или `try_lock_until()`. Если мьютекс уже захвачен другим потоком, то функция `try_lock()` вернет ошибку, функция `lock()` приостановит выполнение потока до получения мьютекса, а функции `try_lock_for()` и `try_lock_until()` приостановят выполнение потока до получения мьютекса или истечения таймаута. Закончив операции над разделяемыми данными, поток должен вызвать функцию `unlock()` (вне зависимости от того, какой функцией мьютекс был захвачен), чтобы освободить мьютекс и дать другим потокам возможность захватить его.

Этот мьютекс называется *рекурсивным*, потому что поток, удерживающий мьютекс типа `std::recursive_timed_mutex`, может снова захватить его любой функцией захвата. Никакой другой поток не сможет захватить этот мьютекс, пока владеющий им поток не вызовет функцию `unlock` столько раз, сколько было успешных вызовов функций захвата.

Экземпляр `std::recursive_timed_mutex` удовлетворяет требованиям концепции `TimedLockable`.

Определение класса

```
class recursive_timed_mutex {
public:
 recursive_timed_mutex(recursive_timed_mutex const&)=delete;
 recursive_timed_mutex& operator=(
  recursive_timed_mutex const&)=delete;

 recursive_timed_mutex();
 ~recursive_timed_mutex();

 void lock();
 void unlock();
 bool try_lock() noexcept;

 template<typename Rep, typename Period>
 bool try_lock_for(
  std::chrono::duration<Rep, Period> const& relative_time);

 template<typename Clock, typename Duration>
 bool try_lock_until(
  std::chrono::time_point<Clock, Duration> const& absolute_time);
};
```

`STD::RECURSIVE_TIMED_MUTEX` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::recursive_timed_mutex`.

Объявление

```
recursive_timed_mutex();
```

Результат

Конструирует экземпляр `std::recursive_timed_mutex`.

Постусловия

Вновь сконструированный объект `std::recursive_timed_mutex` первоначально не

захвачен.

Исключения

Исключение типа `std::system_error`, если не удалось создать экземпляр `std::recursive_timed_mutex`.

`STD::RECURSIVE_TIMED_MUTEX` , ДЕСТРУКТОР

Уничтожает объект `std::recursive_timed_mutex`.

Объявление

```
~recursive_timed_mutex();
```

Предусловия

Объект `*this` не должен быть захвачен.

Результат

Уничтожает `*this`.

Исключения

Нет.

`STD::RECURSIVE_TIMED_MUTEX::LOCK` , ФУНКЦИЯ-ЧЛЕН

Захватывает объект `std::recursive_timed_mutex` для текущего потока.

Объявление

```
void lock();
```

Результат

Блокирует текущий поток, пока мьютекс `*this` не будет захвачен.

Постусловия

`*this` захвачен текущим потоком. Если вызывающий поток уже удерживал `*this`, то счетчик захватов увеличивается на единицу.

Исключения

Исключение типа `std::system_error` в случае ошибки.

`STD::RECURSIVE_TIMED_MUTEX::TRY_LOCK` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока.

Объявление

```
bool try_lock() noexcept;
```

Результат

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока без блокирования.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

> **Примечание**. Если вызывающий поток уже удерживал `*this`, то функция возвращает `true`, и счетчик захватов `*this` текущим потоком увеличивается на единицу. Если текущий поток не удерживал `*this`, то функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`.

`STD::RECURSIVE_TIMED_MUTEX::TRY_LOCK_FOR` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока.

Объявление

```
template<typename Rep, typename Period>
bool try_lock_for(
 std::chrono::duration<Rep, Period> const& relative_time);
```

Результат

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока в течение времени, заданного аргументом `relative_time`. Если `relative_time.count()` равно нулю или отрицательно, то функция возвращается немедленно, как если бы это был вызов `try_lock()`. В противном случае вызывающий поток приостанавливается до получения мьютекса или до истечения времени, заданного аргументом `relative_time`.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

> **Примечание**. Если вызывающий поток уже удерживал `*this`, то функция возвращает `true`, и счетчик захватов `*this` текущим потоком увеличивается на единицу. Если текущий поток не удерживал `*this`, то функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

`STD::RECURSIVE_TIMED_MUTEX::TRY_LOCK_UNTIL` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока.

Объявление

```
template<typename Clock, typename Duration>
bool try_lock_until(
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Результат

Пытается захватить объект `std::recursive_timed_mutex` для текущего потока, пока не наступит момент времени, заданный аргументом `absolute_time`. Если в момент вызова `absolute_time <= Clock::now()`, то функция возвращается немедленно, как если бы это был вызов `try_lock()`. В противном случае вызывающий поток приостанавливается до получения мьютекса или до наступления момента времени, большего или равного `absolute_time`.

Возвращаемое значение

`true`, если вызывающий поток захватил мьютекс, иначе `false`.

Постусловия

`*this` захвачен вызывающим потоком, если функция вернула `true`.

Исключения

Нет.

> **Примечание**. Если вызывающий поток уже удерживал `*this`, то функция возвращает `true`, и счетчик захватов `*this` текущим потоком увеличивается на

единицу. Если текущий поток не удерживал `*this`, то функция может не захватить мьютекс (и вернуть `false`), даже если никакой другой поток не удерживает `*this`. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что если функция вернула `false`, то значение, возвращенное `Clock::now()`, больше или равно `absolute_time` в точке, где поток разблокировался.

`STD::RECURSIVE_TIMED_MUTEX::UNLOCK` , ФУНКЦИЯ-ЧЛЕН

Освобождает объект `std::recursive_timed_mutex`, удерживаемый текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать мьютекс `*this`.

Результат

Освобождает мьютекс `*this`, удерживаемый текущим потоком. Если это последний захват `*this` данным потоком, и другие потоки были блокированы в ожидании `*this`, то один из них разблокируется.

Постусловия

Количество захватов `*this` вызывающим потоком, уменьшается на единицу.

Исключения

Нет.

D.5.5. Шаблон класса `std::lock_guard`

Шаблон класса `std::lock_guard` предоставляет простую обертку владения блокировкой. Тип блокируемого мьютекса задается параметром шаблона `Mutex` и должен удовлетворять требованиям концепции `Lockable`. Заданный мьютекс захватывается в конструкторе и освобождается в деструкторе. Тем самым мы получаем простое средство захвата мьютекса в некотором блоке кода, которое гарантирует освобождение мьютекса при выходе из блока вне зависимости от того, как этот выход произведен: по достижении закрывающей скобки, в результате предложения, меняющего поток управления, например `break` или `return`, или вследствие исключения.

Экземпляры `std::lock_guard` не удовлетворяют требованиям концепций `MoveConstructible`, `CopyConstructible` и `CopyAssignable`.

Определение класса

```
template <class Mutex>
class lock_guard {
public:
 typedef Mutex mutex_type;

 explicit lock_guard(mutex_type& m);
 lock_guard(mutex_type& m, adopt_lock_t);
 ~lock_guard();

 lock_guard(lock_guard const&) = delete;
 lock_guard& operator=(lock_guard const&) = delete;
};
```

`std::lock_guard`, захватывающий конструктор

Конструирует экземпляр `std::lock_guard`, который захватывает указанный мьютекс.

Объявление

```
explicit lock_guard(mutex_type& m);
```

Результат

Конструирует экземпляр `std::lock_guard`, который ссылается на указанный мьютекс. Вызывает `m.lock()`.

Исключения

Любое исключение, возбуждаемое `m.lock()`.

Постусловия

`*this` владеет блокировкой `m`.

`std::lock_guard`, конструктор, перенимающий блокировку

Конструирует экземпляр `std::lock_guard`, который владеет блокировкой указанного мьютекса.

Объявление

```
lock_guard(mutex_type& m, std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть блокировкой `m`.

Результат

Конструирует экземпляр `std::lock_guard`, который ссылается на указанный мьютекс и принимает владение блокировкой `m`, удерживаемой вызывающим потоком.

Исключения

Нет.

Постусловия

`*this` владеет блокировкой `m`, удерживаемой вызывающим потоком.

`std::lock_guard`, деструктор

Уничтожает экземпляр `std::lock_guard` и освобождает соответствующий мьютекс.

Объявление

```
~lock_guard();
```

Результат

Вызывает `m.unlock()` для мьютекса `m`, заданного при конструировании `*this`.

Исключения

Нет.

D.5.6. Шаблон класса `std::unique_lock`

Шаблон класса `std::unique_lock` предоставляет более общую обертку владения блокировкой, чем `std::lock_guard`. Тип блокируемого мьютекса задается параметром шаблона `Mutex` и должен удовлетворять требованиям концепции `BasicLockable`. Вообще говоря, заданный мьютекс захватывается в конструкторе и освобождается в деструкторе, хотя имеются также дополнительные конструкторы и функции-члены, предлагающие другие возможности. Тем самым мы получаем средство захвата мьютекса в некотором блоке кода, которое гарантирует освобождение мьютекса при выходе из блока вне зависимости от того, как этот выход произведен: по достижении закрывающей скобки, в результате предложения, меняющего поток управления, например `break` или `return`, или вследствие исключения.

Функции ожидания в классе `std::condition_variable` требуют объекта `std::unique_lock<std::mutex>`, и любая конкретизация шаблона `std::unique_lock` может быть использована в качестве параметра типа `Lockable` в любом варианте функции `wait` из класса `std::condition_variable_any`.

Если тип `Mutex` удовлетворяет требованиям концепции `Lockable`, то им удовлетворяет и тип `std::unique_lock<Mutex>`. Если, кроме того, тип `Mutex` удовлетворяет требованиям концепции `TimedLockable`, то им удовлетворяет и тип `std::unique_lock<Mutex>`.

Экземпляры `std::unique_lock` удовлетворяют требованиям концепций `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` и `CopyAssignable`.

Определение класса

```
template <class Mutex>
class unique_lock {
public:
 typedef Mutex mutex_type;

 unique_lock() noexcept;
 explicit unique_lock(mutex_type& m);
 unique_lock(mutex_type& m, adopt_lock_t);
 unique_lock(mutex_type& m, defer_lock_t) noexcept;
 unique_lock(mutex_type& m, try_to_lock_t);

 template<typename Clock, typename Duration>
 unique_lock(
  mutex_type& m,
  std::chrono::time_point<Clock, Duration> const& absolute_time);

 template<typename Rep, typename Period>
 unique_lock(
  mutex_type& m,
  std::chrono::duration<Rep, Period> const& relative_time);

 ~unique_lock();
 unique_lock(unique_lock const&) = delete;
 unique_lock& operator=(unique_lock const&) = delete;

 unique_lock(unique_lock&&);
 unique_lock& operator=(unique_lock&&);

 void swap(unique_lock& other) noexcept;

 void lock();
 bool try_lock();
 template<typename Rep, typename Period>
 bool try_lock_for(
  std::chrono::duration<Rep, Period> const& relative_time);
 template<typename Clock, typename Duration>
 bool try_lock_until(
  std::chrono::time_point<Clock, Duration> const& absolute_time);
 void unlock();

 explicit operator bool() const noexcept;
 bool owns_lock() const noexcept;
 Mutex* mutex() const noexcept;
```

```
 Mutex* release() noexcept;
};
```

STD::UNIQUE_LOCK , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует экземпляр `std::unique_lock`, с которым не ассоциирован мьютекс.

Объявление

```
unique_lock() noexcept;
```

Результат

Конструирует экземпляр `std::unique_lock`, с которым не ассоциирован мьютекс.

Постусловия

`this->mutex() == NULL`, `this->owns_lock() == false`.

STD::UNIQUE_LOCK , ЗАХВАТЫВАЮЩИЙ КОНСТРУКТОР

Конструирует экземпляр `std::unique_lock`, который захватывает указанный мьютекс.

Объявление

```
explicit unique_lock(mutex_type& m);
```

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс. Вызывает `m.lock()`.

Исключения

Любое исключение, возбуждаемое `m.lock()`.

Постусловия

`this->owns_lock() == true`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , КОНСТРУКТОР, ПЕРЕНИМАЮЩИЙ БЛОКИРОВКУ

Конструирует экземпляр `std::unique_lock`, который владеет блокировкой указанного мьютекса.

Объявление

```
unique_lock(mutex_type& m, std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть блокировкой `m`.

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс и принимает владение блокировкой `m`, удерживаемой вызывающим потоком.

Исключения

Нет.

Постусловия

`this->owns_lock() == true`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , КОНСТРУКТОР ОТЛОЖЕННОЙ БЛОКИРОВКИ

Конструирует экземпляр `std::unique_lock`, который не владеет блокировкой указанного мьютекса.

Объявление

```
unique_lock(mutex_type& m, std::defer_lock_t) noexcept;
```

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс.

Исключения

Нет.

Постусловия

`this->owns_lock() == false`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , КОНСТРУКТОР ПРОБНОЙ БЛОКИРОВКИ

Конструирует экземпляр `std::unique_lock`, ассоциированный с указанным мьютексом, и пытается захватить этот мьютекс.

Объявление

```
unique_lock(mutex_type& m, std::try_to_lock_t);
```

Предусловия

Тип `Mutex`, которым конкретизирован шаблон `std::unique_lock`, должен удовлетворять требованиям концепции `Lockable`.

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс. Вызывает `m.try_lock()`.

Исключения

Нет.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock()`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , КОНСТРУКТОР ПРОБНОЙ БЛОКИРОВКИ С ОТНОСИТЕЛЬНЫМ ТАЙМАУТОМ

Конструирует экземпляр `std::unique_lock`, ассоциированный с указанным мьютексом, и пытается захватить этот мьютекс.

Объявление

```
template<typename Rep, typename Period>
unique_lock(
 mutex_type& m,
 std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

Тип `Mutex`, которым конкретизирован шаблон `std::unique_lock`, должен удовлетворять требованиям концепции `TimedLockable`.

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс. Вызывает `m.try_lock_for(relative_time)`.

Исключения

Нет.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock_for()`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , КОНСТРУКТОР ПРОБНОЙ БЛОКИРОВКИ С АБСОЛЮТНЫМ ТАЙМАУТОМ

Конструирует экземпляр `std::unique_lock`, ассоциированный с указанным мьютексом, и пытается захватить этот мьютекс.

Объявление

```
template<typename Clock, typename Duration>
unique_lock(
 mutex_type& m,
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, которым конкретизирован шаблон `std::unique_lock`, должен удовлетворять требованиям концепции `TimedLockable`.

Результат

Конструирует экземпляр `std::unique_lock`, который ссылается на указанный мьютекс. Вызывает `m.try_lock_until(relative_time)`.

Исключения

Нет.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock_until()`, `this->mutex() == &m`.

STD::UNIQUE_LOCK , ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Передает владение блокировкой от существующего объекта `std::unique_lock` вновь сконструированному.

Объявление

```
unique_lock(unique_lock&& other) noexcept;
```

Результат

Конструирует экземпляр `std::unique_lock`. Если объект `other` владел блокировкой мьютекса до вызова конструктора, то теперь этой блокировкой владеет вновь сконструированный объект `std::unique_lock`.

Постусловия

Для вновь сконструированного объекта `std::unique_lock x`, `x.mutex()` равно значению `other.mutex()` до вызова конструктора, а `x.owns_lock()` равно значению `other.owns_lock()` до вызова конструктора. `other.mutex() == NULL`, `other.owns_lock() == false`.

Исключения

Нет.

> **Примечание**. Объекты `std::unique_lock` не удовлетворяют требованиям концепции `CopyConstructible`, поэтому копирующего конструктора не существует, существует только этот перемещающий конструктор.

STD::UNIQUE_LOCK , ПЕРЕМЕЩАЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Передает владение блокировкой от одного объекта `std: :unique_ lock` другому.

Объявление

```
unique_lock& operator=(unique_lock&& other) noexcept;
```

Результат

Если `this->owns_lock()` возвращала `true` до вызова этого оператора, то вызывает `this->unlock()`. Если объект `other` владел блокировкой мьютекса до присваивания, то теперь этой блокировкой владеет `*this`.

Постусловия

`this.mutex()` равно значению `other.mutex()` до присваивания, а `this.owns_lock()` равно значению `other.owns_lock()` до присваивания. `other.mutex() == NULL`, `other.owns_lock() == false`.

Исключения

Нет.

> **Примечание**. Объекты `std::unique_lock` не удовлетворяют требованиям концепции `CopyAssignable`, поэтому копирующего оператора присваивания не

существует, существует только этот перемещающий оператор присваивания.

STD::UNIQUE_LOCK , ДЕСТРУКТОР

Уничтожает экземпляр `std::unique_lock` и освобождает соответствующий мьютекс, если им владел уничтоженный экземпляр.

Объявление

```
~unique_lock();
```

Результат

Если `this->owns_lock()` возвращает `true`, то вызывает `this->mutex()->unlock()`.

Исключения

Нет.

STD::UNIQUE_LOCK::SWAP , ФУНКЦИЯ-ЧЛЕН

Обменивает владение ассоциированными блокировками мьютекса между двумя объектами `std::unique_lock`.

Объявление

```
void swap(unique_lock& other) noexcept;
```

Результат

Если `other` владел блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `*this`. Если `*this` владел блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `other`.

Постусловия

`this.mutex()` равно значению `other.mutex()` до вызова, `other.mutex()` равно значению `this.mutex()` до вызова, `this.owns_lock()` равно значению `other.owns_lock()` до вызова, `other.owns_lock()` равно значению `this.owns_lock()` до вызова.

Исключения

Нет.

STD::SWAP , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Обменивает владение ассоциированными блокировками мьютекса между двумя объектами `std::unique_lock`.

Объявление

```
void swap(unique_lock& lhs, unique_lock& rhs) noexcept;
```

Результат

```
lhs.swap(rhs)
```

Исключения

Нет.

STD::UNIQUE_LOCK::LOCK , ФУНКЦИЯ-ЧЛЕН

Захватывает мьютекс, ассоциированный с `*this`.

Объявление

```
void lock();
```

Предусловия

`this->mutex() != NULL`, `this->owns_lock() == false`.

Результат

Вызывает `this->mutex()->lock()`.

Исключения

Любое исключение, возбужденное `this->mutex()->lock()`. Исключение типа `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this-`

>mutex() == NULL. Исключение типа std::system_error с кодом ошибки std::errc::resource_deadlock_would_occur, если this->owns_lock() == true в момент вызова.

Постусловия

this->owns_lock() == true.

STD::UNIQUE_LOCK::TRY_LOCK , ФУНКЦИЯ-ЧЛЕН

Пытается захватить мьютекс, ассоциированный с *this.

Объявление

```
bool try_lock();
```

Предусловия

Тип Mutex, которым конкретизируется шаблон std::unique_lock, должен удовлетворять требованиям концепции Lockable. this->mutex() != NULL, this->owns_lock() == false.

Результат

Вызывает this->mutex ()->try_lock().

Возвращаемое значение

true, если вызов this->mutex()->try_lock() вернул true, иначе false.

Исключения

Любое исключение, возбужденное this->mutex()->try_lock(). Исключение типа std::system_error с кодом ошибки std::errc::operation_not_permitted, если this->mutex() == NULL. Исключение типа std::system_error с кодом ошибки std::errc::resource_deadlock_would_occur, если this->owns_lock() == true в момент вызова.

Постусловия

Если функция возвращает true, то this->owns_lock() == true, иначе this->owns_lock() == false.

STD::UNIQUE_LOCK::UNLOCK , ФУНКЦИЯ-ЧЛЕН

Освобождает мьютекс, ассоциированный с *this.

Объявление

```
void unlock();
```

Предусловия

this->mutex() != NULL, this->owns_lock() == true.

Результат

Вызывает this->mutex()->unlock().

Исключения

Любое исключение, возбужденное this->mutex()->unlock(). Исключение типа std::system_error с кодом ошибки std::errc::operation_not_permitted, если this->owns_lock() == false в момент вызова.

Постусловия

this->owns_lock() == false.

STD::UNIQUE_LOCK::TRY_LOCK_FOR , ФУНКЦИЯ-ЧЛЕН

Пытается захватить мьютекс, ассоциированный с *this, в течение указанного времени.

Объявление

```
template<typename Rep, typename Period>
bool try_lock_for(
std::chrono::duration<Rep, Period> const& relative_time);
```

Предусловия

Тип `Mutex`, которым конкретизируется шаблон `std::unique_lock`, должен удовлетворять требованиям концепции `TimedLockable`. `this->mutex() != NULL`, `this->owns_lock() == false`.

Результат

Вызывает `this->mutex ()->try_lock_for(relative_time)`.

Возвращаемое значение

`true`, если вызов `this->mutex()->try_lock_for()` вернул `true`, иначе `false`.

Исключения

Любое исключение, возбужденное `this->mutex()->try_lock_for()`. Исключение типа `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex() == NULL`. Исключение типа `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если `this->owns_lock() == true` в момент вызова.

Постусловия

Если функция вернула `true`, то `this->owns_lock() == true`, иначе `this->owns_lock() == false`.

`STD::UNIQUE_LOCK::TRY_LOCK_UNTIL` , ФУНКЦИЯ-ЧЛЕН

Пытается захватить мьютекс, ассоциированный с `*this`, в течение указанного времени.

Объявление

```
template<typename Clock, typename Duration>
bool try_lock_until(
std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, которым конкретизируется шаблон `std::unique_lock`, должен удовлетворять требованиям концепции `TimedLockable`. `this->mutex() != NULL`, `this->owns_lock() == false`.

Результат

Вызывает `this->mutex()->try_lock_until(absolute_time)`.

Возвращаемое значение

`true`, если вызов `this->mutex()->try_lock_until()` вернул `true`, иначе `false`.

Исключения

Любое исключение, возбужденное `this->mutex()->try_lock_until()`. Исключение типа `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex() == NULL`. Исключение типа `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если `this->owns_lock() == true` в момент вызова.

Постусловия

Если функция вернула `true`, то `this->owns_lock() == true`, иначе `this->owns_lock() == false`.

`STD::UNIQUE_LOCK::OPERATOR BOOL` , ФУНКЦИЯ-ЧЛЕН

Проверяет, владеет ли `*this` блокировкой мьютекса.

Объявление

```
explicit operator bool() const noexcept;
```

Возвращаемое значение `this->owns_lock()`. *Исключения*

Нет.

Примечание. Это оператор явного преобразования, поэтому он вызывается неявно только в контекстах, где результат используется как булевское значение, а не тогда, когда результат трактуется как целое, равное 0 или 1.

`STD::UNIQUE_LOCK::OWNS_LOCK` , ФУНКЦИЯ-ЧЛЕН

Проверяет, владеет ли `*this` блокировкой мьютекса.

Объявление

```
bool owns_lock() const noexcept;
```

Возвращаемое значение

`true`, если `*this` владеет блокировкой мьютекса, иначе `false`.

Исключения

Нет.

`STD::UNIQUE_LOCK::MUTEX` , ФУНКЦИЯ-ЧЛЕН

Возвращает мьютекс, ассоциированный с `*this`, если таковой имеется.

Объявление

```
mutex_type* mutex() const noexcept;
```

Возвращаемое значение

Указатель на мьютекс, ассоциированный с `*this`, если таковой имеется, иначе `NULL`.

Исключения

Нет.

`STD::UNIQUE_LOCK::RELEASE` , ФУНКЦИЯ-ЧЛЕН

Возвращает мьютекс, ассоциированный с `*this`, если таковой имеется, и разрывает эту ассоциацию.

Объявление

```
mutex_type* release() noexcept;
```

Результат

Разрывает ассоциацию мьютекса с `*this`, не освобождая блокировку.

Возвращаемое значение

Указатель на мьютекс, ассоциированный с `*this`, если таковой имеется, иначе `NULL`.

Постусловия

`this->mutex() == NULL`, `this->owns_lock() == false`.

Исключения

Нет.

Примечание. Если `this->owns_lock()` вернула бы до этого обращения `true`, то с этого момента за освобождение мьютекса отвечает вызывающая программа.

D.5.7. Шаблон функции `std::lock`

Шаблон функции `std::lock` предоставляет возможность захватить сразу несколько мьютексов, не опасаясь возникновения взаимоблокировки из-за несогласованного порядка захвата.

Объявление

```
template<typename LockableType1, typename... LockableType2>
void lock(LockableType1& m1, LockableType2& m2...);
```

Предусловия

Типы параметров `LockableType1`, `LockableType2`, ... должны удовлетворять требованиям концепции `Lockable`.

Результат

Захватывает все объекты `m1`, `m2`, ... допускающих блокировку типов, обращаясь к функциям-членам `lock()`, `try_lock()` и `unlock()` этих типов в порядке, который гарантированно не приводит к взаимоблокировкам, но в остальном не специфицирован.

Постусловия

Текущий поток захватывает все переданные в аргументах объекты.

Исключения

Любое исключение, возбуждаемое обращениями к функциям `lock()`, `try_lock()` и `unlock()`.

> **Примечание**. Если исключение распространяется за пределы `std::lock`, то для любого объекта `m1`, `m2`, ..., для которого в результате обращения к `lock()` или `try_lock()` была успешно получена блокировка, гарантированно будет вызвана функция `unlock()`.

D.5.8. Шаблон функции `std::try_lock`

Шаблон функции `std::try_lock` предоставляет возможность захватить сразу несколько допускающих блокировку объектов, так что либо захвачены все, либо ни один.

Объявление

```
template<typename LockableType1, typename... LockableType2>
int try_lock(LockableType1& m1, LockableType2& m2...);
```

Предусловия

Типы параметров `LockableType1`, `LockableType2`, ... должны удовлетворять требованиям концепции `Lockable`.

Результат

Пытается захватить все объекты `m1`, `m2`, ... допускающих блокировку типов, обращаясь по очереди к функции `try_lock()` каждого из них. Если `try_lock()` вернёт `false` или возбудит исключение, то уже захваченные блокировки освобождаются путем вызова функции `unlock()` соответствующего объекта.

Возвращаемое значение

-1, если были захвачены все блокировки (то есть все вызовы `try_lock()` вернули `true`), в противном случае начинающийся с нуля индекс объекта, для которого вызов `try_lock()` вернул `false`.

Постусловия

Если функция вернула -1, то текущий поток захватил все переданные в аргументах объекты. В противном случае все объекты, которая функция успела захватить, освобождены.

Исключения

Любое исключение, возбуждаемое обращениями к функции `try_lock`.

> **Примечание**. Если исключение распространяется за пределы `std::try_lock`, то для любого объекта `m1`, `m2`, ..., для которого в результате обращения к `try_lock()` была успешно получена блокировка, гарантированно будет вызвана

функция `unlock()`.

D.5.9. Класс `std::once_flag`

Экземпляры класса `std::once_flag` используются совместно с шаблоном функции `std::call_once` для гарантии того, что некая функция будет вызвала ровно один раз, даже если ее могут вызывать одновременно несколько потоков.

Экземпляры `std::once_flag` не удовлетворяют требованиям концепций `CopyConstructible`, `CopyAssignable`, `MoveConstructible` и `MoveAssignable`.

Определение класса

```
struct once_flag {
 constexpr once_flag() noexcept;

 once_flag(once_flag const&) = delete;
 once_flag& operator=(once_flag const&) = delete;
};
```

`STD::ONCE_FLAG` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Создает объект `std::once_flag` в состоянии, обозначающем, что ассоциированная функция еще не вызывалась.

Объявление

```
constexpr once_flag() noexcept;
```

Результат

Конструирует новый экземпляр `std::once_flag`, оставляя его в состоянии, означающем, что ассоциированная функция еще не вызывалась. Поскольку в конструкторе присутствует квалификатор `constexpr`, то экземпляр со статическим временем жизни конструируется на этапе статической инициализации, что предотвращает состояние гонки и зависимость от порядка инициализации.

D.5.10. Шаблон функции `std::call_once`

Шаблон функции `std::call_once` используется совместно с объектом `std::once_flag` для гарантии того, что некая функция будет вызвала ровно один раз, даже если ее могут вызывать одновременно несколько потоков.

Объявление

```
template<typename Callable, typename... Args>
void call_once(
 std::once_flag& flag, Callable func, Args args...);
```

Предусловия

Выражение `INVOKE(func, args)` допустимо для переданных значений `func` и `args`. Тип `Callable` и все члены `Args` удовлетворяют требованиям концепции `MoveConstructible`.

Результат

Обращения к `std::call_once` с одним и тем же объектом `std::once_flag` сериализуются. Если раньше не было результативного обращения к `std::call_once` с данным объектом `std::once_flag`, то аргумент `func` (или его копия) вызывается так, будто имело место обращение к `INVOKE(func, args)`, причем вызов `std::call_once` считается результативным тогда и только тогда, когда вызов `func` завершился без возбуждения

исключения. Если имело место исключение, то оно передается вызывающей программе. Если ранее уже было результативное обращение к `std::call_once` с данным объектом `std::once_flag`, то новый вызов `std::call_once` возвращает управление, не вызывая `func`.

Синхронизация

Возврат из результативного вызова `std::call_once` с объектом `std::once_flag` происходит-раньше всех последующих вызовов `std::call_once` с тем же объектом `std::once_flag`.

Исключения

Исключение типа `std::system_error`, если желаемого эффекта добиться не удалось, или любое исключение, возбужденное при обращении к `func`.

D.6. Заголовок <ratio>

В заголовке <ratio> объявлены средства для поддержки арифметических операций с рациональными числами на этапе компиляции.

Содержимое заголовка

```
namespace std {
template<intmax_t N, intmax_t D=1>
class ratio;

// арифметические операции с рациональными числами
template <class R1, class R2>
using ratio_add = см. описание;

template <class R1, class R2>
using ratio_subtract = см. описание;

template <class R1, class R2>
using ratio_multiply = см. описание;

template <class Rl, class R2>
using ratio_divide = см. описание;

// сравнение рациональных чисел
template <class R1, class R2>
struct ratio_equal;

template <class R1, class R2>
struct ratio_not_equal;

template <class R1, class R2>
struct ratio_less;

template <class R1, class R2>
struct ratio_less_equal;

template <class R1, class R2>
struct ratio_greater;

template <class R1, class R2>
struct ratio_greater_equal;

typedef ratio<1, 1000000000000000000> atto;
typedef ratio<1, 1000000000000000> femto;
typedef ratio<1, 1000000000000> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
```

```
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000, 1> tera;
typedef ratio<1000000000000000, 1> peta;
typedef ratio<1000000000000000000, 1> exa;
```

D.6.1. Шаблон класса `std::ratio`

Шаблон класса `<std::ratio>` предоставляет механизм для выполнения на этапе компиляции арифметических операций с рациональными числами, например: деления пополам (`std::ratio<1, 2>`), нахождения двух третей (`std::ratio<2, 3>`) пятнадцати сорок третьих (`std::ratio<15, 43>`). В стандартной библиотеке C++ этот шаблон используется для задания периода при конкретизации шаблона класса `std::chrono::duration`.

Определение класса

```
template <intmax_t N, intmax_t D = 1>
class ratio {
public:
 typedef ratio<num, den> type;

 static constexpr intmax_t num = см. ниже;
 static constexpr intmax_t den = см. ниже;
};
```

Требования

`D` не может быть равно нулю.

Описание

`num` и `den` — соответственно числитель и знаменатель дроби `N/D` после сокращения без общих множителей. Значение `den` всегда положительно. Если `N` и `D` одного знака, то `num` положительно, иначе `num` отрицательно.

Примеры

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

D.6.2. Псевдоним шаблона `std::ratio_add`

Псевдоним шаблона `std::ratio_add` предоставляет механизм сложения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_add = std::ratio<см. ниже>;
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Результат

`ratio_add<R1,R2>` определяется как псевдоним конкретизации `std::ratio`, представляющий сумму дробей, представленных параметрами `R1` и `R2`, если эту сумму можно вычислить без переполнения. Если при вычислении возникает переполнение, то

программа считается некорректной. В отсутствии переполнения `std::ratio_add<R1, R2>` будет иметь такие же значения `num` и `den`, как в конкретизации `std::ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>`.

Примеры

```
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::num == 11
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::den == 15
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::num == 3
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::den == 2
```

D.6.3. Псевдоним шаблона `std::ratio_subtract`

Псевдоним шаблона `std::ratio_subtract` предоставляет механизм вычитания двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_subtract = std::ratio<см. ниже>;
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Результат

`ratio_subtract<R1, R2>` определяется как псевдоним конкретизации `std::ratio`, представляющий разность дробей, представленных параметрами `R1` и `R2`, если эту разность можно вычислить без переполнения. Если при вычислении возникает переполнение, то программа считается некорректной. В отсутствии переполнения `std::ratio_subtract<R1, R2>` будет иметь такие же значения `num` и `den`, как в конкретизации `std::ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>`.

Примеры

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::num == 2
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::den == 15
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::num == -5
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::den == 6
```

D.6.4. Псевдоним шаблона `std::ratio_multiply`

Псевдоним шаблона `std::ratio_multiply` предоставляет механизм умножения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_multiply = std::ratio<см. ниже>;
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Результат

`ratio_multiply<R1, R2>` определяется как псевдоним конкретизации `std::ratio`, представляющий произведение дробей, представленных параметрами `R1` и `R2`, если это произведение можно вычислить без переполнения. Если при вычислении возникает переполнение, то программа считается некорректной. В отсутствии переполнения

`std::ratio_multiply<R1, R2>` будет иметь такие же значения `num` и `den`, как в конкретизации `std::ratio<R1::num * R2::num, R1::den * R2::den>`.

Примеры

```
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::num == 2
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::den == 15
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::num == 5
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::den == 7
```

D.6.5. Псевдоним шаблона `std::ratio_divide`

Псевдоним шаблона `std::ratio_divide` предоставляет механизм деления двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_divide = std::ratio<см. ниже>;
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Результат

`ratio_divide<R1, R2>` определяется как псевдоним конкретизации `std::ratio`, представляющий частное дробей, представленных параметрами `R1` и `R2`, если это частное можно вычислить без переполнения. Если при вычислении возникает переполнение, то программа считается некорректной. В отсутствии переполнения `std::ratio_divide<R1, R2>` будет иметь такие же значения `num` и `den`, как в конкретизации `std::ratio<R1::num * R2::den, R1::den * R2::num>`.

Примеры

```
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::num == 5
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::den == 6
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::num == 7
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::den == 45
```

D.6.6. Шаблон класса `std::ratio_equal`

Шаблон класса `std::ratio_equal` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_equal:
 public std::integral_constant<
  bool, (R1::num == R2::num) && (R1::den == R2::den)> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Примеры

```
std::ratio_equal<std::ratio<1,3>, std::ratio<2,6> >::value
 == true
std::ratio_equal<std::ratio<1,3>, std::ratio<1,6> >::value
 == false
std::ratio_equal<std::ratio<1,3>, std::ratio<2,3> >::value
```

```
 == false
std::ratio_equal<std::ratio<1,3>, std::ratio<1,3> >::value
 == true
```

D.6.7. Шаблон класса `std::ratio_not_equal`

Шаблон класса `std::ratio_not_equal` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_not_equal:
 public std::integral_constant<
  bool, !ratio_equal<R1,R2>::value> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Примеры

```
std::ratio_not_equal<
 std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_not_equal<
 std::ratio<1,3>, std::ratio<1,6> >::value == true
std::ratio_not_equal<
 std::ratio<1,3>, std::ratio<2,3> >::value == true
std::ratio_not_equal<
 std::ratio<1,3>, std::ratio<1,3> >::value == false
```

D.6.8. Шаблон класса `std::ratio_less`

Шаблон класса `std::ratio_less` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_less:
 public std::integral_constant<bool, см. ниже> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

Результат

`std::ratio_less<R1,R2>` наследует шаблону `std::integral_constant<bool, value>`, где `value` — это `(R1::num*R2::den) < (R2::num*R1::den)`. Если возможно, реализация должна использовать такой метод вычисления результата, при котором не возникает переполнения. Если при вычислении возникает переполнение, то программа считается некорректной.

Примеры

```
std::ratio_less<std::ratio<1,3>, std::ratio<2,6> >::value
 == false
std::ratio_less<std::ratio<1,6>, std::ratio<1,3> >::value
 == true
std::ratio_less<
 std::ratio<999999999,1000000000>,
```

```
 std::ratio<1000000001,1000000000> >::value == true
std::ratio_less<
 std::ratio<1000000001,1000000000>,
 std::ratio<999999999,1000000000> >::value == false
```

D.6.9. Шаблон класса `std::ratio_greater`

Шаблон класса `std::ratio_greater` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_greater:
 public std::integral_constant<
  bool, ratio_less<R2, R1>::value> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

D.6.10. Шаблон класса `std::ratio_less_equal`

Шаблон класса `std::ratio_less_equal` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_less_equal:
 public std::integral_constant<
  bool, !ratio_less<R2, R1>::value> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

D.6.11. Шаблон класса `std::ratio_greater_equal`

Шаблон класса `std::ratio_greater_equal` предоставляет механизм сравнения двух значений `std::ratio` на этапе компиляции с применением правил арифметических операций с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_greater_equal:
 public std::integral_constant<
  bool, !ratio_less<R1,R2>::value> {};
```

Предусловия

`R1` и `R2` должны быть конкретизациями шаблона `std::ratio`.

D.7. Заголовок <thread>

В заголовке <thread> объявлены средства для идентификации и управления потоками, а также функции для приостановки потоков.

Содержимое заголовка

```
namespace std {

class thread;

namespace this_thread {
thread::id get_id() noexcept;

void yield() noexcept;

template<typename Rep, typename Period>
void sleep_for(
 std::chrono::duration<Rep, Period> sleep_duration);

template<typename Clock, typename Duration>
void sleep_until(
 std::chrono::time_point<Clock, Duration> wake_time);
}
}
```

D.7.1. Класс std::thread

Класс Std::thread применяется для управления потоком выполнения. В нем имеются средства для запуска нового потока и ожидания завершения потока, а также для идентификации потоков. Также в класс включены другие функции для управления потоком выполнения.

Определение класса

```
class thread {
public:
 // Типы
 class id;
 typedef implementation-defined
 native_handle_type; // необязательно

 // Конструкторы и деструкторы
 thread() noexcept;
 ~thread();

 template<typename Callable, typename Args...>
 explicit thread(Callable&& func, Args&&... args);

 // Копирование и перемещение
 thread(thread const& other) = delete;
 thread(thread&& other) noexcept;

 thread& operator=(thread const& other) = delete;
```

```
 thread& operator=(thread&& other) noexcept;

 void swap(thread& other) noexcept;

 void join();
 void detach();
 bool joinable() const noexcept;

 id get_id() const noexcept;

 native_handle_type native_handle();

 static unsigned hardware_concurrency() noexcept;
};

void swap(thread& lhs, thread& rhs);
```

STD::THREAD::ID , КЛАСС

Экземпляр класса `std::thread::id` идентифицирует конкретный поток выполнения.

Определение класса

```
class thread::id {
public:
 id() noexcept;
};

bool operator==(thread::id x, thread::id y) noexcept;
bool operator!=(thread::id x, thread::id y) noexcept;
bool operator<(thread::id x, thread::id y) noexcept;
bool operator<=(thread::id x, thread::id y) noexcept;
bool operator>(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

template<typename charT, typename traits>
basic_ostream<charT, traits>&
 operator<<(basic_ostream<charT, traits>&& out, thread::id id);
```

> **Примечание**. Значение `std::thread::id`, идентифицирующее конкретный поток выполнения, должно отличаться от значения экземпляра `std::thread::id`, сконструированного по умолчанию, и от значения, представляющего любой другой поток.

> **Примечание**. Значения `std::thread::id` для конкретных потоков непредсказуемы и могут различаться при разных прогонах одной и той же программы.

Экземпляры `std::thread::id` удовлетворяют требованиям концепций `CopyConstructible` и `CopyAssignable`, поэтому их можно копировать и присваивать друг другу

STD::THREAD::ID , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::thread::id`, который не представляет никакой поток выполнения.

Объявление

```
id() noexcept;
```

Результат

Конструирует экземпляр `std::thread::id`, с которым связано особое значение, интерпретируемое как *не поток*.

Исключения

Нет.

Примечания. Во всех сконструированных по умолчанию экземпляров `std::thread::id` хранится одно и то же значение.

STD::THREAD::ID , ОПЕРАТОР СРАВНЕНИЯ НА РАВЕНСТВО

Сравнивает два экземпляра `std::thread::id`, проверяя, представляют ли они один и тот же поток.

Объявление

```
bool operator==(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

`true`, если `lhs` и `rhs` представляют один и тот же поток выполнения или оба имеют значение *не поток*, `false`, если `lhs` и `rhs` представляют разные потоки или один представляет поток, а другой имеет значение *не поток*.

Исключения

Нет.

STD::THREAD::ID , ОПЕРАТОР СРАВНЕНИЯ НА НЕРАВЕНСТВО

Сравнивает два экземпляра `std::thread::id`, проверяя, представляют ли они разные потоки.

Объявление

```
bool operator!=(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

```
!(lhs==rhs)
```

Исключения

Нет.

STD::THREAD::ID , ОПЕРАТОР СРАВНЕНИЯ МЕНЬШЕ

Сравнивает два экземпляра `std::thread::id`, проверяя, предшествует ли один из них другому в смысле отношения полного порядка, существующего на множестве значений идентификаторов потоков.

Объявление

```
bool operator<(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

`true`, если значение `lhs` предшествует значению `rhs` в смысле отношения полного порядка, существующего на множестве значений идентификаторов потоков. Если `lhs != rhs`, то истинно ровно одно из утверждений `lhs < rhs` и `rhs < lhs`, тогда как второе ложно. Если `lhs == rhs`, то оба утверждения `lhs < rhs` и `rhs < lhs` ложны.

Исключения

Нет.

Примечание. Особое значение *не поток*, которое хранится в сконструированном по умолчанию экземпляре `std::thread::id`, меньше любого другого экземпляра `std::thread::id`, представляющего поток выполнения. Если два экземпляра `std::thread::id` равны, то ни один из них не меньше другого. Любое множество различных значений `std::thread::id` полностью упорядочено, и этот порядок остается непротиворечивым на всем протяжении работы программы. Порядок может изменяться при разных прогонах одной и той же программы.

`STD::THREAD::ID` , ОПЕРАТОР СРАВНЕНИЯ МЕНЬШЕ ИЛИ РАВНО

Сравнивает два экземпляра `std::thread::id`, проверяя, предшествует ли один из них другому в смысле отношения полного порядка, существующего на множестве значений идентификаторов потоков, или оба экземпляра совпадают.

Объявление

```
bool operator<=(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

```
!(rhs < lhs)
```

Исключения

Нет.

`STD::THREAD::ID` , ОПЕРАТОР СРАВНЕНИЯ БОЛЬШЕ

Сравнивает два экземпляра `std::thread::id`, проверяя, следует ли один из них за другим в смысле отношения полного порядка, существующего на множестве значений идентификаторов потоков.

Объявление

```
bool operator>(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

```
rhs < lhs
```

Исключения

Нет.

`STD::THREAD::ID` , ОПЕРАТОР СРАВНЕНИЯ БОЛЬШЕ ИЛИ РАВНО

Сравнивает два экземпляра `std::thread::id`, проверяя, следует ли один из них за другим в смысле отношения полного порядка, существующего на множестве значений идентификаторов потоков, или оба экземпляра совпадают.

Объявление

```
bool operator>=(
 std::thread::id lhs, std::thread::id rhs) noexcept;
```

Возвращаемое значение

```
!(lhs < rhs)
```

Исключения

Нет.

`STD::THREAD::ID` , ОПЕРАТОР ВСТАВКИ В ПОТОК

Выводит строковое представление значения `std::thread::id` в указанный поток.

Объявление

```
template<typename charT, typename traits>
basic_ostream<charT, traits>&
```

```
operator<<(basic_ostream<charT, traits>&& out, thread::id id);
```

Результат

Выводит строковое представление значения `std::thread::id` в указанный поток.

Возвращаемое значение

`out`

Исключения

Нет.

> **Примечание**. Формат строкового представления не специфицирован. Равные экземпляры имеют одинаковое представление, неравные — различное.

`STD::THREAD::NATIVE_HANDLE_TYPE` , ПСЕВДОНИМ ТИПА

`native_handle_type` — это псевдоним типа, который можно использовать в сочетании с платформенно-зависимыми API.

Объявление

```
typedef implementation-defined native_handle_type;
```

> **Примечание**. Этот псевдоним типа необязателен. Если он определен, то реализация должна предоставить тип, пригодный для использования в сочетании с платформенно-зависимыми API.

`STD::THREAD::NATIVE_HANDLE` , ФУНКЦИЯ-ЧЛЕН

Возвращает значение типа `native_handle_type`, представляющее поток выполнения, ассоциированный с `*this`.

Объявление

```
native_handle_type native_handle();
```

> **Примечание**. Эта функция необязательна. Если она имеется, то возвращаемое значение должно быть пригодно для использования в сочетании с платформенно-зависимыми API.

`STD::THREAD` , КОНСТРУКТОР ПО УМОЛЧАНИЮ

Конструирует объект `std::thread`, с которым не ассоциирован никакой поток выполнения.

Объявление

```
thread() noexcept;
```

Результат

Конструирует экземпляр `std::thread`, с которым не ассоциирован никакой поток выполнения.

Постусловия

Для вновь сконструированного объекта x типа `std::thread x.get_id()==id()`.

Исключения

Нет.

`STD::THREAD` , КОНСТРУКТОР

Конструирует экземпляр `std::thread`, ассоциированный с новым потоком выполнения.

Объявление

```
template<typename Callable, typename Args...>
```

```
explicit thread(Callable&& func, Args&&... args);
```

Предусловия

`func` и каждый элемент списка `args` должен удовлетворять требованиям концепции `MoveConstructible`.

Результат

Конструирует экземпляр `std::thread` и ассоциирует с ним вновь созданный потоком выполнения. Копирует или перемещает аргумент `func` и все элементы списка `args` во внутреннюю память, где они хранятся на протяжении всего времени жизни потока выполнения. Вызывает `INVOKE(copy-of-func, copy-of-args)` в новом потоке выполнения.

Постусловия

Для вновь сконструированного объекта `x` типа `std::thread x.get_id() != id()`.

Исключения

Исключение типа `std::system_error`, если не удалось запустить новый поток. Любое исключение, возбужденное при копировании `func` или `args` во внутреннюю память.

Синхронизация

Вызов этого конструктора происходит-раньше выполнения переданной функции во вновь созданном потоке выполнения.

`STD::THREAD` , ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР

Передает владение потоком выполнения от существующего объекта `std::thread` вновь созданному.

Объявление

```
thread(thread&& other) noexcept;
```

Результат

Конструирует экземпляр `std::thread`. Если с объектом `other` перед вызовом конструктора был ассоциирован поток выполнения, то теперь этот поток оказывается ассоциирован с вновь созданным объектом `std::thread`. В противном случае с вновь созданным объектом std::thread не ассоциирован никакой поток.

Постусловия

Для вновь сконструированного объекта `x` типа `std::thread x.get_id()` равно значению `other.get_id()` до вызова конструктора, `other.get_id() == id()`.

Исключения

Нет.

> **Примечание**. Объекты `std::thread` не удовлетворяют требованиям концепции `CopyConstructible`, поэтому копирующего конструктора не существует, существует только этот перемещающий конструктор.

`STD::THREAD` , ДЕСТРУКТОР

Уничтожает объект `std::thread`.

Объявление

```
~thread();
```

Результат

Уничтожает `*this`. Если с `*this` ассоциирован поток выполнения (`this->joinable()` возвращает `true`), то вызывает `std::terminate()`, то есть аварийно завершает программу.

Исключения

Нет.

STD::THREAD , ПЕРЕМЕЩАЮЩИЙ ОПЕРАТОР ПРИСВАИВАНИЯ

Передает владение потоком выполнения от одного объекта `std::thread` другому.

Объявление

```
thread& operator=(thread&& other) noexcept;
```

Результат

Если до вызова этого оператора `this->joinable()` возвращала `true`, то вызывает `std::terminate()` для аварийного завершения программы. Если с `other` до вызова оператора был ассоциирован поток выполнения, то после вызова он оказывается ассоциирован с `*this`. В противном случае с `*this` не ассоциирован никакой поток выполнения.

Постусловия

`this->get_id()` равно значению `other.get_id()` до вызова конструктора. `other.get_id() == id()`.

Исключения

Нет.

> **Примечание**. Объекты `std::thread` не удовлетворяют требованиям концепции `CopyAssignable`, поэтому копирующего оператора присваивания не существует, существует только этот перемещающий оператор присваивания.

STD::THREAD::SWAP , ФУНКЦИЯ-ЧЛЕН

Обменивает владение ассоциированными потоками выполнения между двумя объектами `std::thread`.

Объявление

```
void swap(thread& other) noexcept;
```

Результат

Если с `other` до вызова функции был ассоциирован поток выполнения, то после вызова он оказывается ассоциирован с `*this`. В противном случае с `*this` не ассоциирован никакой поток выполнения. Если с `*this` до вызова функции был ассоциирован поток выполнения, то после вызова он оказывается ассоциирован с `other`. В противном случае с `other` не ассоциирован никакой поток выполнения.

Постусловия

`this->get_id()` равно значению `other.get_id()` до вызова функции. `other.get_id()` равно значению `this->get_id()` до вызова функции.

Исключения

Нет.

STD::THREAD::SWAP , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Обменивает владение ассоциированными потоками выполнения между двумя объектами `std::thread`.

Объявление

```
void swap(thread& lhs, thread& rhs) noexcept;
```

Результат

```
lhs.swap(rhs)
```

Исключения

Нет.

STD::THREAD::JOINABLE , ФУНКЦИЯ-ЧЛЕН

Опрашивает, ассоциирован ли с `*this` поток выполнения.

Объявление

```
bool joinable() const noexcept;
```

Возвращаемое значение

`true`, если с `*this` ассоциирован поток выполнения, иначе `false`.

Исключения

Нет.

STD::THREAD::JOIN , ФУНКЦИЯ-ЧЛЕН

Ожидает завершения потока выполнения, ассоциированного с `*this`.

Объявление

```
void join();
```

Предусловия

`this->joinable()` должна возвращать `true`.

Результат

Блокирует текущий поток, пока не завершится поток, ассоциированный с `*this`.

Постусловия

`this->get_id() == id()`. Поток выполнения, который был ассоциирован с `*this` до вызова этой функции, завершился.

Синхронизация

Завершение потока выполнения, который был ассоциирован с `*this` до вызова этой функции, происходит-раньше возврата из `join()`.

Исключения

`std::system_error`, если требуемого эффекта добиться не удалось или если `this->joinable()` возвращает `false`.

STD::THREAD::DETACH , ФУНКЦИЯ-ЧЛЕН

Отсоединяет поток выполнения, ассоциированный с `*this`.

Объявление

```
void detach();
```

Предусловия

`this->joinable()` возвращает `true`.

Результат

Отсоединяет поток выполнения, ассоциированный с `*this`.

Постусловия

`this->get_id() == id()`, `this->joinable() == false`. Поток выполнения, который был ассоциирован с `*this` до вызова этой функции, отсоединен и более не ассоциирован ни с каким объектом `std::thread`.

Исключения

`std::system_error`, если требуемого эффекта добиться не удалось или если `this->joinable()` возвращает `false` в момент вызова.

STD::THREAD::GET_ID , ФУНКЦИЯ-ЧЛЕН

Возвращает значение типа `Std::thread::id`, идентифицирующее поток выполнения, ассоциированный с `*this`.

Объявление

```
thread::id get_id() const noexcept;
```

Возвращаемое значение

Если с `*this` ассоциирован поток выполнения, то возвращает экземпляр `std::thread::id`, который идентифицирует этот поток. В противном случае возвращает

сконструированный по умолчанию экземпляр `std::thread::id`.

Исключения

Нет.

`STD::THREAD::HARDWARE_CONCURRENCY` , СТАТИЧЕСКАЯ ФУНКЦИЯ-ЧЛЕН

Возвращает информацию о том, сколько потоков могут одновременно работать на имеющемся оборудовании.

Объявление

```
unsigned hardware_concurrency() noexcept;
```

Возвращаемое значение

Количество потоков, которые могут одновременно исполняться на имеющемся оборудовании. Например, это может быть число процессоров. Если информация недоступна или определена неточно, возвращает 0.

Исключения

Нет.

D.7.2. Пространство имен `this_thread`

Функции из пространства имен `std::this_thread` применяются к вызывающему потоку.

`STD::THIS_THREAD::GET_ID` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Возвращает значение типа `std::thread::id`, идентифицирующее текущий поток выполнения.

Объявление

```
thread::id get_id() noexcept;
```

Возвращаемое значение

Экземпляр `std::thread::id`, идентифицирующий текущий поток выполнения.

Исключения

Нет.

`STD::THIS_THREAD::YIELD` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Информирует библиотеку о том, что поток, вызвавший эту функцию, в данный момент не хочет выполняться. Обычно используется в коротких циклах, чтобы не потреблять излишне много процессорного времени.

Объявление

```
void yield() noexcept;
```

Результат

Предоставляет библиотеке возможность запланировать другой поток вместо текущего.

Исключения

Нет.

`STD::THIS_THREAD::SLEEP_FOR` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Приостанавливает выполнение текущего потока на указанное время.

Объявление

```
template<typename Rep, typename Period>
void sleep_for(
 std::chrono::duration<Rep, Period> const& relative_time);
```

Результат

Приостанавливает выполнение текущего потока на указанное время `relative_time`.

Примечание. Поток может быть блокирован дольше, чем указано. Если возможно, истекшее время измеряется по стабильным часам.

Исключения

Нет.

`STD::THIS_THREAD::SLEEP_UNTIL` , ФУНКЦИЯ, НЕ ЯВЛЯЮЩАЯСЯ ЧЛЕНОМ КЛАССА

Приостанавливает выполнение текущего потока до указанного момента времени.

Объявление

```
template<typename Clock, typename Duration>
void sleep_until(
 std::chrono::time_point<Clock, Duration> const& absolute_time);
```

Результат

Приостанавливает выполнение текущего потока до наступления момента `absolute_time` по указанным часам `Clock`.

Примечание. Не дается никаких гарантий относительно того, сколько времени будет блокирован вызывающий поток. Гарантируется лишь, что значение, возвращенное `Clock::now()`, больше или равно `absolute_time` в точке, где поток разблокировался.

Исключения

Нет.

Ресурсы

Печатные ресурсы

Cargill, Tom, "Exception Handling: A False Sense of Security," in C++ Report 6, no. 9, (November-December 1994). Доступно также по адресу http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

Hoare, C.A.R., Communicating Sequential Processes (Prentice Hall International, 1985), ISBN 0131532898. Доступно также по адресу http://www.usingcsp.com/cspbook.pdf.[22]

Michael, Maged M., "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes" in PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing (2002), ISBN 1-58113-485-1.

——. U.S. Patent and Trademark Office application 20040107227, "Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation."

Sutter, Herb, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (Addison Wesley Professional, 1999), ISBN 0-201-61562-2.[23]

——.The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, в Dr. Dobb's Journal 30, no. 3 (March 2005). Доступно также по адресу http://www.gotw.ca/publications/concurrency-ddj.htm.

Сетевые ресурсы

Atomic Ptr Plus Project Home, http://atomic-ptr-plus.sourceforge.net/.
Boost C++ library collection, http://www.boost.org.
C++0x/C++11 Support in GCC, http://gcc.gnu.org/projects/cxx0x.html.
C++11 — The Recently Approved New ISO C++ Standard, http://www.research.att.com/~bs/C++0xFAQ.html.
Erlang Programming Language, http://www.erlang.org/.
GNU General Public License, http://www.gnu.org/licenses/gpl.html.
Haskell Programming Language, http://www.haskell.org/.

---

notes

1

Страница состояния компилятора GNU C++0x/C++11 http://gcc.gnu.org/projects/cxx0x.html.

2

Реализация `just::thread` библиотеки C++ Standard Thread Library, http://www.stdthread.co.uk.

Библиотеки Boost для C++, http://www.boost.org.

4

Tom Cargill «Exception Handling: A False Sense of Security» в журнале *C++ Report* 6, № 9 (ноябрь-декабрь 1994). Доступна также по адресу http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

Herb Sutter, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (Addison Wesley Professional, 1999).

Howard E. Hinnant, “Multithreading API for C++0X-A Layered Approach,” C++ Standards Committee Paper N2094, http://www.open-std.org/jtcl/sc22/wg21/docs/papers/2006/n2094.html.

7

В книге «Путеводитель для путешествующих автостопом по галактике» был построен компьютер Deep Thought, который должен был найти «ответ на главный вопрос жизни, Вселенной и всего на свете». Оказалось, что ответ на вопрос — 42.

promise — обещание. Прим. перев.

http://www.haskell.org/.

Communicating Sequential Processes, C.A.R. Hoare, Prentice Hall, 1985. Бесплатная онлайновая версия доступна по адресу http://www.usingcsp.com/cspbook.pdf.

11

О том, что такое спекулятивное исполнение, см. http://en.wikipedia.org/wiki/Speculative_execution. Прим. перев.

«Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes», Maged M. Michael, в сборнике *PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, «Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation».

GNU General Public License http://www.gnu.org/licenses/gpl.html.

IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/

Atomic Ptr Plus Project, http://atomic-ptr-plus.sourceforge.net/.

http://www.mpi-forum.org/

http://www.openmp.org/

http://setiathome.ssl.berkeley.edu/

http://threadingbuildingblocks.org/

http://www.research.att.com/~bs/C++0xFAQ.html

Имеется русский перевод. Ч. Хоар «Взаимодействующие последовательные процессы», Мир, 1989. *Прим. перев*.

Имеется русский перевод. Герб Саттер «Решение сложных задач на C++», Вильямс, 2008. *Прим. перев*.