Энтони Уильямс



C++

Практика многопоточного программирования

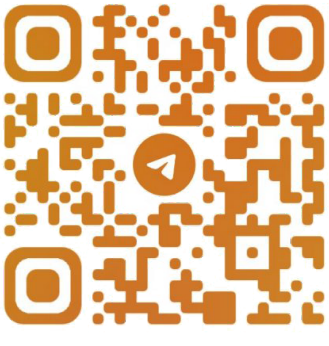
ВТОРОЕ ИЗДАНИЕ

MANNING

C++ Concurrency in Action

Second Edition





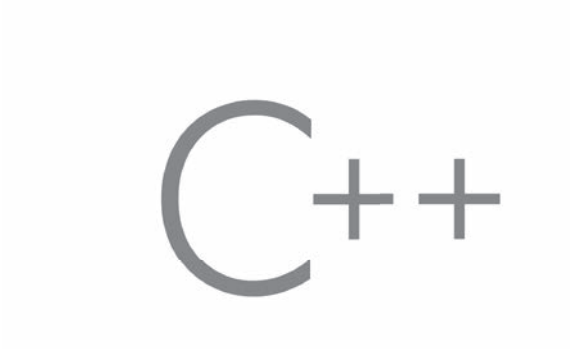
@CODELIBRARY_IT

ANTHONY WILLIAMS



MANNING
SHELTER ISLAND

Энтони Уильямс



Практика многопоточного программирования

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2020

Энтони Уильямс

C++. Практика многопоточного программирования

Серия «Для профессионалов»

Перевел с английского *Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*М. Колесников*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*А. Дубейко, Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

ББК 32.973.2-018.1
УДК 004.43

Уильямс Энтони

У36 C++. Практика многопоточного программирования. — СПб.: Питер, 2020. — 640 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0831-2

Язык C++ выбирают, когда надо создать по-настоящему молниеносные приложения. А качественная конкурентная обработка сделает их еще быстрее. Новые возможности C++17 позволяют использовать всю мощь многопоточного программирования, чтобы с легкостью решать задачи графической обработки, машинного обучения и др.

Энтони Уильямс, эксперт конкурентной обработки, рассматривает примеры и описывает практические задачи, а также делится секретами, которые пригодятся всем, в том числе и самым опытным разработчикам. Теперь вам доступны все аспекты конкурентной обработки на C++17 — от создания новых потоков до проектирования полнофункциональных многопоточных алгоритмов и структур данных.

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

ISBN 978-1617294693 англ.
ISBN 978-5-4461-0831-2

© 2019 by Manning Publications Co. All rights reserved.
© Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Для профессионалов», 2020

Права на издание получены по соглашению с Apress. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 05.12.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 51,600. Тираж 1000. Заказ 0000.

Краткое содержание

Предисловие ... 16
Благодарности ... 18
О книге ... 20
Об авторе ... 24
Об иллюстрации на обложке ... 25
От издательства ... 26
Глава 1. Здравствуй, мир конкурентности в C++! ... 27
Глава 2. Управление потоками ... 44
Глава 3. Совместное использование данных несколькими потоками ... 65
Глава 4. Синхронизация конкурентных операций ... 105
Глава 5. Модель памяти C++ и операции над атомарными типами ... 162
Глава 6. Разработка конкурентных структур данных с блокировками ... 217
Глава 7. Разработка конкурентных структур данных без блокировок ... 251
Глава 8. Разработка конкурентного кода ... 302
Глава 9. Усовершенствованное управление потоками ... 355
Глава 10. Алгоритмы параллельных вычислений ... 383
Глава 11. Тестирование и отладка многопоточных приложений ... 396

Приложения

Приложение А. Краткий справочник по некоторым функциям языка C++11 ... 414
Приложение Б. Краткое сравнение библиотек для написания конкурентных программ ... 444
Приложение В. Среда передачи сообщений и полный пример программы управления банкоматом ... 446
Приложение Г. Справочник по C++ Thread Library ... 463

Оглавление

Предисловие ... 16

Благодарности ... 18

О книге ... 20
- Структура издания ... 20
- Для кого предназначена эта книга ... 21
- Порядок чтения ... 21
- Условные обозначения и загрузка кода ... 22
- Требования к программным средствам ... 22
- Форум, посвященный книге ... 22

Об авторе ... 24

Об иллюстрации на обложке ... 25

От издательства ... 26

Глава 1. Здравствуй, мир конкурентности в C++! ... 27

1.1. Что такое конкурентность ... 28
- 1.1.1. Конкурентность в компьютерных системах ... 28
- 1.1.2. Подходы к конкурентности ... 31
- 1.1.3. Сравнение конкурентности и параллелизма ... 33

1.2. Зачем используется конкурентность ... 34
- 1.2.1. Конкурентность для разделения неотложных задач ... 34
- 1.2.2. Конкурентность для повышения производительности: параллелизм задач и данных ... 35
- 1.2.3. Когда не нужна конкурентность ... 36

1.3. Конкурентность и многопоточность в C++ ... 37
- 1.3.1. История поддержки многопоточности в C++ ... 38
- 1.3.2. Поддержка конкурентности в стандарте C++11 ... 39

1.3.3. Расширение поддержки конкурентности и параллелизма в C++14 и C++17 ... 39
1.3.4. Эффективность, обеспеченная библиотекой потоков C++ ... 40
1.3.5. Средства, ориентированные на использование конкретной платформы ... 41
1.4. Приступаем к практической работе ... 41
1.4.1. Здравствуй, мир конкурентности ... 42
Резюме ... 43

Глава 2. Управление потоками ... 44
2.1. Основы управления потоками ... 45
2.1.1. Запуск потока ... 45
2.1.2. Ожидание завершения потока ... 48
2.1.3. Ожидание в исключительных обстоятельствах ... 49
2.1.4. Запуск потоков в фоновом режиме ... 51
2.2. Передача аргументов функции потока ... 53
2.3. Передача права владения потоком ... 55
2.4. Выбор количества потоков в ходе выполнения программы ... 60
2.5. Идентификация потоков ... 63
Резюме ... 64

Глава 3. Совместное использование данных несколькими потоками ... 65
3.1. Проблемы совместного использования данных несколькими потоками ... 66
3.1.1. Состояние гонки ... 68
3.1.2. Пути обхода проблемных состояний гонок ... 69
3.2. Защита совместно используемых данных с применением мьютексов ... 70
3.2.1. Использование мьютексов в C++ ... 70
3.2.2. Структуризация кода для защиты совместно используемых данных ... 72
3.2.3. Обнаружение состояния гонки, присущего интерфейсам ... 74
3.2.4. Взаимная блокировка: проблема и решение ... 81
3.2.5. Дополнительные рекомендации по обходу взаимных блокировок ... 84
3.2.6. Гибкая блокировка с применением std::unique_lock ... 91
3.2.7. Передача владения мьютексом между областями видимости ... 92
3.2.8. Блокировка с соответствующей степенью детализации ... 94
3.3. Альтернативные средства защиты совместно используемых данных ... 97
3.3.1. Защита совместно используемых данных во время инициализации ... 97
3.3.2. Защита редко обновляемых структур данных ... 101
3.3.3. Рекурсивная блокировка ... 103
Резюме ... 104

Глава 4. Синхронизация конкурентных операций ... 105
4.1. Ожидание наступления события или создания другого условия ... 106
4.1.1. Ожидание выполнения условий с применением условных переменных ... 107
4.1.2. Создание потокобезопасной очереди с условными переменными ... 110
4.2. Ожидание единичных событий с помощью фьючерсов ... 115
4.2.1. Возвращение значений из фоновых задач ... 116
4.2.2. Связывание задачи с фьючерсом ... 119
4.2.3. Создание промисов std::promise ... 121
4.2.4. Сохранение исключения на будущее ... 124
4.2.5. Ожидание сразу в нескольких потоках ... 125
4.3. Ожидание с ограничением по времени ... 128
4.3.1. Часы ... 128
4.3.2. Продолжительность ... 130
4.3.3. Моменты времени ... 131
4.3.4. Функции, принимающие сроки ожидания ... 133
4.4. Применение синхронизации операций для упрощения кода ... 135
4.4.1. Функциональное программирование с применением фьючерсов ... 135
4.4.2. Синхронизация операций путем передачи сообщений ... 141
4.4.3. Конкурентность, организованная в стиле продолжений с применением Concurrency TS ... 146
4.4.4. Выстраиваем продолжения в цепочку ... 148
4.4.5. Ожидание готовности более чем одного фьючерса ... 151
4.4.6. Ожидание первого фьючерса в наборе с when_any ... 153
4.4.7. Защелки и барьеры в Concurrency TS ... 156
4.4.8. Базовый тип защелки std::experimental::latch ... 156
4.4.9. Основной барьер std::experimental::barrier ... 157
4.4.10. Барьер std::experimental::flex_barrier — более гибкий соратник барьера std::experimental::barrier ... 159
Резюме ... 161

Глава 5. Модель памяти C++ и операции над атомарными типами ... 162
5.1. Основы модели памяти ... 163
5.1.1. Объекты и их размещение в памяти ... 163
5.1.2. Объекты, области памяти и конкурентность ... 165
5.1.3. Очередность внесения изменений ... 166
5.2. Атомарные операции и типы в C++ ... 166
5.2.1. Стандартные атомарные типы ... 167
5.2.2. Операции над std::atomic_flag ... 171

5.2.3. Операции над std::atomic<bool> ... 173
5.2.4. Операции над std::atomic<T*>: арифметика указателей ... 176
5.2.5. Операции над стандартными атомарными целочисленными типами ... 177
5.2.6. Шаблон первичного класса std::atomic<> ... 178
5.2.7. Свободные функции для атомарных операций ... 180
5.3. Синхронизация операций и принудительное упорядочение ... 183
5.3.1. Отношение «синхронизируется с» ... 184
5.3.2. Отношение «происходит до» ... 185
5.3.3. Упорядочение доступа к памяти для атомарных операций ... 187
5.3.4. Последовательности освобождений и отношения «синхронизируется с» ... 207
5.3.5. Барьеры ... 209
5.3.6. Упорядочение неатомарных операций с помощью атомарных операций ... 211
5.3.7. Упорядочение неатомарных операций ... 212
Резюме ... 216

Глава 6. Разработка конкурентных структур данных с блокировками ... 217
6.1. Что означает разработка структур данных для конкурентного доступа ... 218
6.1.1. Рекомендации по разработке структур данных для конкурентного доступа ... 219
6.2. Конкурентные структуры данных с блокировками ... 220
6.2.1. Потокобезопасный стек, использующий блокировки ... 221
6.2.2. Потокобезопасная очередь с использованием блокировок и условных переменных ... 224
6.2.3. Потокобезопасная очередь с использованием подробной детализации блокировок и условных переменных ... 228
6.3. Разработка более сложных структур данных с использованием блокировок ... 240
6.3.1. Создание потокобезопасной поисковой таблицы с использованием блокировок ... 240
6.3.2. Создание потокобезопасного списка с использованием блокировок ... 246
Резюме ... 250

Глава 7. Разработка конкурентных структур данных без блокировок ... 251
7.1. Определения и выводы ... 252
7.1.1. Типы структур данных, не подвергаемых блокировкам ... 252
7.1.2. Структуры данных, свободные от блокировок ... 254
7.1.3. Структуры данных, свободные от ожиданий ... 254
7.1.4. Все за и против создания структур данных, свободных от блокировок ... 255
7.2. Примеры структур данных, свободных от блокировок ... 256
7.2.1. Создание потокобезопасного стека без блокировок ... 257

7.2.2. Устранение утечек: управление памятью в структурах данных, свободных от блокировок ... 261
7.2.3. Определение узлов, не подлежащих утилизации, с помощью указателей опасности ... 266
7.2.4. Определение используемых узлов путем подсчета ссылок ... 274
7.2.5. Применение модели памяти к свободному от блокировок стеку ... 281
7.2.6. Создание потокобезопасной очереди без блокировок ... 285
7.3. Рекомендации по созданию структур данных без блокировок ... 298
7.3.1. Для создания прототипа используйте std::memory_order_seq_cst ... 298
7.3.2. Воспользуйтесь схемой утилизации памяти, свободной от блокировок ... 299
7.3.3. Остерегайтесь проблемы ABA ... 299
7.3.4. Выявляйте циклы активного ожидания и организуйте помощь другому потоку ... 300
Резюме ... 301

Глава 8. Разработка конкурентного кода ... 302
8.1. Способы распределения работы между потоками ... 303
8.1.1. Распределение данных между потоками до начала обработки ... 304
8.1.2. Рекурсивное распределение данных ... 305
8.1.3. Распределение работы по типам задач ... 309
8.2. Факторы, влияющие на производительность конкурентного кода ... 312
8.2.1. А сколько у нас процессоров? ... 313
8.2.2. Конкуренция при обращении к данным и перебрасывание данных в кэш-памяти процессоров ... 314
8.2.3. Ложное совместное использование памяти ... 317
8.2.4. Насколько близко расположены по отношению друг к другу ваши данные? ... 318
8.2.5. Переоценка вычислительных возможностей и чрезмерное количество переключений задач ... 319
8.3. Разработка структур данных для высокопроизводительных многопоточных приложений ... 320
8.3.1. Распределение элементов массива для сложных операций ... 320
8.3.2. Схемы доступа к данным в других структурах данных ... 322
8.4. Дополнительные факторы, учитываемые при разработке конкурентных программ ... 324
8.4.1. Безопасность исключений в параллельных алгоритмах ... 325
8.4.2. Масштабируемость и закон Амдала ... 332
8.4.3. Компенсация потерь на ожидание за счет применения нескольких потоков ... 333
8.4.4. Повышение отзывчивости за счет конкурентности ... 334

8.5. Разработка конкурентного кода на практике ... 336
8.5.1. Параллельная реализация std::for_each ... 337
8.5.2. Параллельная реализация std::find ... 339
8.5.3. Параллельная реализация std::partial_sum ... 345
Резюме ... 354

Глава 9. Усовершенствованное управление потоками ... 355
9.1. Пулы потоков ... 356
9.1.1. Простейший пул потоков ... 356
9.1.2. Ожидание завершения задач, переданных пулу потоков ... 358
9.1.3. Задачи, ожидающие завершения других задач ... 362
9.1.4. Предотвращение конкуренции при обращении к очереди работ ... 365
9.1.5. Перехват работы ... 367
9.2. Прерывание потоков ... 371
9.2.1. Запуск и прерывание другого потока ... 371
9.2.2. Обнаружение того, что поток был прерван ... 373
9.2.3. Прерывание ожидания на условной переменной ... 374
9.2.4. Прерывание ожидания на std::condition_variable_any ... 377
9.2.5. Прерывание других блокирующих вызовов ... 379
9.2.6. Обработка прерываний ... 380
9.2.7. Прерывание фоновых задач при выходе из приложения ... 381
Резюме ... 382

Глава 10. Алгоритмы параллельных вычислений ... 383
10.1. Перевод стандартных библиотечных алгоритмов в режим параллельных вычислений ... 383
10.2. Политики выполнения ... 384
10.2.1. Общие последствия от задания политики выполнения ... 385
10.2.2. std::execution::sequenced_policy ... 386
10.2.3. std::execution::parallel_policy ... 387
10.2.4. std::execution::parallel_unsequenced_policy ... 388
10.3. Параллельные алгоритмы из стандартной библиотеки C++ ... 388
10.3.1. Примеры использования параллельных алгоритмов ... 391
10.3.2. Подсчет посещений ... 393
Резюме ... 395

Глава 11. Тестирование и отладка многопоточных приложений ... 396
11.1. Типы ошибок, связанных с конкурентностью ... 397
11.1.1. Нежелательная блокировка ... 397
11.1.2. Состояния гонки ... 398

11.2. Приемы обнаружения ошибок, связанных с конкурентностью ... 399
11.2.1. Просмотр кода с целью поиска возможных ошибок ... 400
11.2.2. Обнаружение ошибок, связанных с конкурентностью, путем тестирования ... 402
11.2.3. Разработка кода с прицелом на удобство тестирования ... 404
11.2.4. Приемы тестирования многопоточного кода ... 406
11.2.5. Структурирование многопоточного тестового кода ... 408
11.2.6. Тестирование производительности многопоточного кода ... 411
Резюме ... 412

Приложения

Приложение А. Краткий справочник по некоторым функциям языка C++11 ... 414
A.1. Ссылки на r-значения ... 414
A.1.1. Семантика перемещений ... 415
A.1.2. Ссылки на r-значения и шаблоны функций ... 418
A.2. Удаленные функции ... 419
A.3. Функции по умолчанию ... 420
A.4. constexpr-функции ... 424
A.4.1. constexpr и типы, определенные пользователем ... 425
A.4.2. constexpr-объекты ... 428
A.4.3. Требования к constexpr-функциям ... 428
A.4.4. constexpr и шаблоны ... 430
A.5. Лямбда-функции ... 430
A.5.1. Лямбда-функции, ссылающиеся на локальные переменные ... 432
A.6. Вариативные шаблоны ... 436
A.6.1. Расширение пакета параметров ... 437
A.7. Автоматическое выведение типа переменной ... 440
A.8. Локальные переменные потока ... 441
A.9. Выведение аргументов шаблона класса ... 442
Резюме ... 443

Приложение Б. Краткое сравнение библиотек для написания конкурентных программ ... 444

Приложение В. Среда передачи сообщений и полный пример программы управления банкоматом ... 446

Приложение Г. Справочник по C++ Thread Library ... 463
Г.1. Заголовок <chrono> ... 463
Г.1.1. Шаблон класса std::chrono::duration ... 464
Г.1.2. Шаблон класса std::chrono::time_point ... 474

Г.1.3. Класс std::chrono::system_clock ... 477
Г.1.4. Класс std::chrono::steady_clock ... 479
Г.1.5. Псевдоним типа std::chrono::high_resolution_clock ... 481
Г.2. Заголовок <condition_variable> ... 481
Г.2.1. Класс std::condition_variable ... 481
Г.2.2. Класс std::condition_variable_any ... 490
Г.3. Заголовок <atomic> ... 499
Г.3.1. Псевдонимы типа std::atomic_xxx ... 501
Г.3.2. Макросы ATOMIC_xxx_LOCK_FREE ... 501
Г.3.3. Макрос ATOMIC_VAR_INIT ... 502
Г.3.4. Перечисление std::memory_order ... 502
Г.3.5. Функция std::atomic_thread_fence ... 503
Г.3.6. Функция std::atomic_signal_fence ... 504
Г.3.7. Класс std::atomic_flag ... 504
Г.3.8. Шаблон класса std::atomic ... 508
Г.3.9. Специализации шаблона std::atomic ... 520
Г.3.10. Специализации std::atomic<integral-type> ... 521
Г.4. Заголовок <future> ... 539
Г.4.1. Шаблон класса std::future ... 540
Г.4.2. Шаблон класса std::shared_future ... 546
Г.4.3. Шаблон класса std::packaged_task ... 552
Г.4.4. Шаблон класса std::promise ... 558
Г.4.5. Шаблон функции std::async ... 565
Г.5. Заголовок <mutex> ... 566
Г.5.1. Класс std::mutex ... 567
Г.5.2. Класс std::recursive_mutex ... 570
Г.5.3. Класс std::timed_mutex ... 572
Г.5.4. Класс std::recursive_timed_mutex ... 577
Г.5.5. Класс std::shared_mutex ... 582
Г.5.6. Класс std::shared_timed_mutex ... 586
Г.5.7. Шаблон класса std::lock_guard ... 594
Г.5.8. Шаблон класса std::scoped_lock ... 596
Г.5.9. Шаблон класса std::unique_lock ... 598
Г.5.10. Шаблон класса std::shared_lock ... 608
Г.5.11. Шаблон функции std::lock ... 619
Г.5.12. Шаблон функции std::try_lock ... 620
Г.5.13. Класс std::once_flag ... 621
Г.5.14. Шаблон функции std::call_once ... 621

Г.6. Заголовок <ratio> ... 622
Г.6.1. Шаблон класса std::ratio ... 623
Г.6.2. Псевдоним шаблона std::ratio_add ... 624
Г.6.3. Псевдоним шаблона std::ratio_subtract ... 624
Г.6.4. Псевдоним шаблона std::ratio_multiply ... 625
Г.6.5. Псевдоним шаблона std::ratio_divide ... 626
Г.6.6. Шаблон класса std::ratio_equal ... 626
Г.6.7. Шаблон класса std::ratio_not_equal ... 627
Г.6.8. Шаблон класса std::ratio_less ... 627
Г.6.9. Шаблон класса std::ratio_greater ... 628
Г.6.10. Шаблон класса std::ratio_less_equal ... 628
Г.6.11. Шаблон класса std::ratio_greater_equal ... 628
Г.7. Заголовок <thread> ... 628
Г.7.1. Класс std::thread ... 629
Г.7.2. Пространство имен this_thread ... 639

Посвящается Ким, Хью и Эрин

Предисловие

С понятием многопоточного кода я столкнулся на своей первой работе, на которую устроился по окончании колледжа. Мы разрабатывали приложение для обработки данных, предназначенное для наполнения базы данных поступающими записями. Данных было много, но записи были независимы друг от друга, и требовалось немного обработать их перед добавлением. Чтобы по максимуму нагрузить наш компьютер UltraSPARC, имеющий десять центральных процессоров, мы запускали код в нескольких потоках и каждый поток обрабатывал собственный набор поступающих записей. Мы написали код на C++, используя потоки POSIX, допустили кучу ошибок — многопоточность для всех нас была в новинку, — но все же справились. Кроме того, работая над проектом, я узнал о существовании Комитета по стандартизации C++ и только что вышедшем стандарте C++.

С тех пор я очень заинтересовался многопоточностью и конкурентностью[1]. В том, что другим представлялось чем-то весьма сложным, запутанным и приносящим одни проблемы, я видел весьма эффективное средство, позволяющее программе за-

---

[1] В русскоязычной литературе нередко путаются термины «параллелизм» и «конкурентность». Оба термина означают одновременность процессов, но первый — на физическом уровне (параллельное исполнение нескольких процессов, нацеленное только на повышение скорости исполнения за счет использования соответствующей аппаратной поддержки), а второй — на логическом (парадигма проектирования систем, идентифицирующая процессы как независимые, что в том числе позволяет их исполнять физически параллельно, но в первую очередь нацелено на упрощение написания многопоточных программ и повышение их устойчивости) (источник: «Википедия»). В этой книге термин concurrency переводится как «конкурентность» и подразумевает одновременные вычисления, обеспечиваемые средствами конкретного языка и среды выполнения. Под одновременностью понимается не только и не столько выполнение сразу нескольких вычислений в одно и то же физическое время (что больше похоже на параллелизм), но и выполнение разных взаимосвязанных вычислений за один и тот же период времени в рамках одного приложения с использованием программных средств синхронизации (которая не нужна параллельным вычислениям в силу их независимости друг от друга). — *Примеч. ред.*

действовать все доступное оборудование и увеличить скорость своей работы. Чуть позже я освоил приемы использования этой технологии для повышения отзывчивости и производительности приложений даже на одноядерном оборудовании за счет применения нескольких потоков, позволяющих не замечать ожидания завершения таких продолжительных процессов, как ввод-вывод данных. Я также разобрался в том, как это все работает на уровне операционной системы и как центральные процессоры Intel справляются с переключением задач.

Интересуясь C++, я начал общаться с участниками Ассоциации пользователей языков C и C++ (ACCU), а затем и с представителями Комиссии по стандартизации C++ при Институте стандартов Великобритании (BSI), а также с разработчиками библиотеки Boost. Я с интересом наблюдал за началом разработки Boost Thread Library, а когда автор забросил проект, не упустил своего шанса принять участие в этом процессе. На протяжении нескольких лет я остаюсь основным разработчиком и сопровождающим Boost Thread Library.

По мере того как Комитет по стандартизации C++ перешел от устранения недочетов в существующем стандарте к выработке предложений по стандарту C++11 (обозначенному C++0x в надежде, что его разработка завершится до 2009-го, а позже официально названному C++11 по причине окончательной публикации в 2011 году), я еще больше втянулся в работу BSI и начал вносить собственные предложения. Как только стало ясно, что многопоточность приобрела особую актуальность, я всецело отдался ее продвижению и стал автором и соавтором многих предложений, касающихся многопоточности и конкурентности, сформировавших соответствующую часть стандарта. Совместно с группой по конкурентности участвовал в работе над внесением изменений для выработки стандарта C++17, спецификации Concurrency TS и предложений на будущее. Мне повезло в том, что таким образом удалось объединить две основные сферы моих компьютерных интересов — C++ и многопоточность.

В этой книге отражен весь мой опыт как в программировании на C++, так и в применении многопоточности, она предназначена для обучения других разработчиков программных средств на C++ приемам безопасного и эффективного использования C++17 Thread Library и спецификации Concurrency TS. Я также надеюсь заразить читателей своим энтузиазмом по части решения рассматриваемых проблем.

Благодарности

Прежде всего хочу сказать огромное спасибо своей жене Ким за ее любовь и поддержку, которую она мне оказывала, когда я писал книгу. Работа над первым изданием отнимала изрядную долю моего свободного времени на протяжении четырех лет. Над вторым изданием также пришлось немало потрудиться. Без терпения, поддержки и понимания Ким я бы не справился.

Далее хочу поблагодарить коллектив издательства Manning, сделавший возможным выпуск данной книги: издателя Марьяна Бэйса (Marjan Bace), соиздателя Майкла Стивенса (Michael Stephens), моего редактора-консультанта Синтию Кейн (Cynthia Kane), редактора-рецензента Александара Драгосавлевича (Aleksandar Dragosavljević), моих корректоров, компанию Safis Editing и Хайди Уорд (Heidi Ward), корректора Мелоди Долаб (Melody Dolab). Без их участия вы бы сейчас эту книгу не читали.

Хочу также поблагодарить представителей Комитета по стандартизации C++, составлявших документы по многопоточным средствам: Андрея Александреску (Andrei Alexandrescu), Пита Беккера (Pete Becker), Боба Блейнера (Bob Blainer), Ханса Бёма (Hans Boehm), Бемана Доуса (Beman Dawes), Лоуренса Кроула (Lawrence Crowl), Питера Димова (Peter Dimov), Джеффа Гарланда (Jeff Garland), Кевлина Хенни (Kevlin Henney), Ховарда Хиннанта (Howard Hinnant), Бена Хатчингса (Ben Hutchings), Яна Кристофферсона (Jan Kristofferson), Дага Ли (Doug Lea), Пола Маккенни (Paul McKenney), Ника Макларена (Nick McLaren), Кларка Нельсона (Clark Nelson), Билла Пью (Bill Pugh), Рауля Сильвера (Raul Silvera), Херба Саттера (Herb Sutter), Детлефа Фольманна (Detlef Vollmann) и Майкла Вонга (Michael Wong), а также всех, кто комментировал документы, обсуждал их на заседаниях Комитета и иными путями помогал настроить поддержку многопоточности и конкурентности в C++11, C++14, C++17 и спецификации Concurrency TS.

И наконец, хочу с благодарностью отметить тех, чьи предложения позволили существенно улучшить книгу: доктора Джейми Оллсопа (Jamie Allsop), Питера Димова (Peter Dimov), Говарда Хиннанта (Howard Hinnant), Рика Моллоя (Rick Molloy), Джонатана Уэйкли (Jonathan Wakely) и доктора Рассела Уиндера (Russel

Winder). В особенности Рассела за его подробные рецензии и Фредерика Флайоля (Frédéric Flayol), технического корректора, в процессе производства тщательно проверившего окончательный вариант рукописи на явные ошибки. (Разумеется, все оставшиеся в тексте ляпы целиком на моей совести.) Кроме того, хочу поблагодарить группу рецензентов второго издания книги: Ала Нормана (Al Norman), Андрея де Араужо Формигу (Andrei de Araújo Formiga), Чада Брюбейкера (Chad Brewbaker), Дуайта Уилкинса (Dwight Wilkins), Хьюго Филипе Лопеса (Hugo Filipe Lopes), Виейру Дурана (Vieira Durana), Юру Шикина (Jura Shikin), Кента Р. Спилнера (Kent R. Spillner), Мариа Джемини (Maria Gemini), Матеуша Малента (Mateusz Malenta), Маурицио Томази (Maurizio Tomasi), Ната Луенгнарюмитчая (Nat Luengnaruemitchai), Роберта С. Грина II (Robert C. Green II), Роберта Траусмута (Robert Trausmuth), Санчира Картиева (Sanchir Kartiev) и Стивена Парра (Steven Parr). Спасибо также читателям предварительного издания, не пожалевшим времени и указавшим на ошибки или отметившим текст, требующий пояснения.

О книге

Эта книга представляет собой подробное руководство по средствам поддержки конкурентности и многопоточности из нового стандарта C++, начиная от базового использования классов `std::thread`, `std::mutex` и `std::async` и заканчивая сложностями атомарных операций и модели памяти.

Структура издания

В главах 1–4 дается введение в различные библиотечные средства и порядок их возможного применения.

В главе 5 рассматриваются низкоуровневые подробности модели памяти и атомарные операции, включая порядок использования атомарных операций для наложения ограничений на порядок доступа к памяти со стороны другого кода. Ею завершаются вводные главы.

В главах 6 и 7 начинается изучение программирования на высоком уровне и приводится ряд примеров того, как использовать основные средства для создания более сложных структур данных — основанных на блокировках (в главе 6) и без блокировок (в главе 7).

В главе 8 продолжается рассмотрение высокоуровневых тем и даются рекомендации по разработке многопоточного кода. Здесь охвачены факторы, влияющие на производительность, и приведены примеры реализации различных параллельных алгоритмов.

В главе 9 речь идет об управлении потоками — пулах потоков, очередях работ и операциях, прерывающих выполнение потоков.

В главе 10 рассматриваются новые аспекты поддержки параллелизма, появившиеся в C++17 и представленные в виде дополнительных переопределений для многих алгоритмов стандартной библиотеки.

В главе 11 разговор идет о тестировании и отладке — типах ошибок, приемах обнаружения мест возникновения ошибок, порядке проведения тестирования на наличие ошибок и т. д.

В приложения включено краткое описание некоторых новых средств языка, введенных новым стандартом и имеющих отношение к многопоточности. Здесь также рассмотрены детали библиотеки передачи сообщений, упомянутой в главе 4, и приведен полный справочник по C++17 Thread Library.

Для кого предназначена эта книга

Эта книга для тех, кто создает многопоточные приложения на языке C++. Если вы пользуетесь новыми средствами многопоточности из стандартной библиотеки C++, то издание послужит руководством по основным вопросам. Если вы работаете с другими библиотеками многопоточности, то описанные рекомендации и приемы тоже могут стать полезным подспорьем.

Предполагается, что читатели обладают практическими навыками программирования на языке C++, возможно не зная о новых свойствах языка — их описание дано в приложении A. Также не требуются знания и навыки в области многопоточного программирования, хотя они были бы нелишними.

Порядок чтения

Если ранее вам не приходилось создавать многопоточный код, советую читать книгу последовательно от начала до конца, возможно, опуская некоторые детали из главы 5. Материал главы 7 основан на информации, изложенной в главе 5, поэтому, если вы ее пропустите, то изучение главы 7 нужно отложить до тех пор, пока не будет изучена глава 5.

Если ранее вам не приходилось пользоваться новыми возможностями языка, появившимися в стандарте C++11, то, возможно, сначала стоит просмотреть приложение A, чтобы убедиться, что вам понятны примеры, приводимые в книге. Впрочем, в основном тексте упоминания о новых средствах графически выделены, поэтому, встретив что-то незнакомое, вы всегда можете обратиться к приложению.

Даже при наличии большого опыта создания многопоточного кода в других средах начальные главы все же стоит прочитать, чтобы увидеть, как уже известные вам понятия переносятся на те средства, что вошли в новый стандарт языка C++. Если есть намерение поработать с низкоуровневыми средствами с применением атомарных переменных, нужно обязательно изучить главу 5. А главу 8 стоит просмотреть, чтобы убедиться, что вы знакомы с такими вопросами, как безопасность выдачи исключений в многопоточном коде на C++. Если перед вами стоит конкретная задача, то быстро найти соответствующий раздел поможет оглавление.

Даже после освоения C++ Thread Library вам все равно пригодится материал приложения Г, например для поиска конкретных сведений о каждом классе или вызове функции. Также можно будет время от времени заглядывать в основные главы, освежая в памяти конкретные конструкции или просматривая примеры кода.

Условные обозначения и загрузка кода

Исходный код, приводимый в листингах и в тексте книги, набран моноширинным шрифтом. Многие листинги сопровождаются примечаниями, выделяющими важные понятия. Иногда примечания сопровождаются пронумерованными метками, фигурирующими в пояснениях, которые приводятся сразу за листингами.

Исходный код всех работоспособных примеров, приводимых в книге, доступен для загрузки с веб-сайта издательства по адресу www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition. Исходный код можно загрузить также из хранилища GitHub по адресу https://github.com/anthonywilliams/ccia_code_samples.

Требования к программным средствам

Чтобы воспользоваться кодом, приведенным в книге, не внося в него каких-то изменений, понадобится самая последняя версия компилятора C++, поддерживающая средства языка C++17, перечисленные в приложении A, кроме того, нужна копия C++ Standard Thread Library.

Во время написания книги с реализацией C++17 Standard Thread Library уже поставлялись самые последние версии g++, clang++ и Microsoft Visual Studio. Они поддерживали большинство особенностей языка, рассмотренных в приложении A, а поддержка остальных возможностей ожидалась в ближайшее время.

Моя компания Just Software Solutions Ltd продает полную реализацию C++11 Standard Thread Library для ряда более старых компиляторов, а также реализацию спецификации Concurrency TS для новых версий clang, gcc и Microsoft Visual Studio[1]. Эта реализация использовалась для тестирования примеров, приведенных в книге.

Boost Thread Library[2], переносимая на многие платформы, предоставляет API, основанный на предложениях, касающихся развития C++11 Standard Thread Library. В большинство примеров из книги можно внести изменения для работы с Boost Thread Library, для чего нужно аккуратно заменить префикс `std::` на `boost::` и воспользоваться соответствующими директивами `#include`. Есть ряд средств, которые в Boost Thread Library либо не поддерживаются (например, `std::async`), либо называются иначе (например, `boost::unique_future`).

Форум, посвященный книге

Купив второе издание книги, вы получите свободный доступ к закрытому веб-форуму, организованному издательством Manning Publications, где можно оставить комментарий, касающийся текста, задать технические вопросы и получить помощь от автора или других пользователей. Для этого нужно перейти по адресу www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition. Дополнительная информация

[1] Реализация just::thread библиотеки C++ Standard Thread Library, http://www.stdthread.co.uk.

[2] Коллекция библиотеки Boost C++, http://www.boost.org.

о форумах издательства Manning и правилах поведения на них содержится по адресу https://forums.manning.com/forums/about.

В соответствии с обязательствами издательства Manning читателям предоставляется площадка, где они могут вести содержательный диалог между собой и с автором. При этом автор не обязан участвовать в диалоге, для него эта деятельность добровольная (и неоплачиваемая). Чтобы заинтересовать автора в разговоре с вами, задавайте ему сложные вопросы.

Форум и архивы предыдущих обсуждений будут доступны на сайте издательства на весь период допечатывания книги.

Об авторе



Энтони Уильямс (Anthony Williams) — британский разработчик, консультант и преподаватель с более чем 20-летним опытом программирования на C++. С 2001 года он принимает активное участие в деятельности группы по выработке стандартов BSI C++ и является автором или соавтором многих документов Комитета по стандартизации C++, благодаря которым библиотека потоков была включена в стандарт C++11. Он продолжает работать над новыми функциями, позволяющими усовершенствовать инструментарий для конкурентности на C++, а также над предложениями по стандартам и реализацией соответствующих средств расширения just::thread Pro для библиотеки потоков C++ от компании Just Software Solutions Ltd. Энтони живет на крайнем западе Англии, в графстве Корнуолл.

Об иллюстрации на обложке

Рисунок на обложке книги называется «Традиционный костюм японской женщины»[1]. Это репродукция из четырехтомника Томаса Джефферис (Thomas Jefferys) «Коллекция костюмов разных народов»[2], изданного в Лондоне в 1757–1772 годах. Коллекция включает отпечатанные с медных пластин и раскрашенные вручную гравюры с изображением одежды народов со всего мира. Издание существенно повлияло на дизайн театральных костюмов. Разнообразие рисунков в коллекции ярко свидетельствует о великолепии костюмов на Лондонской сцене более 200 лет назад. Можно было увидеть традиционную историческую и современную одежду людей, живших в разное время в разных странах, сделав их понятнее и ближе зрителям, посещавшим лондонские театры.

С тех пор стиль одежды сильно изменился и исчезло разнообразие, свойственное различным областям и странам. Теперь трудно различить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистичной точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду более насыщенной личной жизни или более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну компьютерную книгу от другой, издательство Manning проявляет инициативу и деловую сметку, украшая обложки книг изображениями, которые показывают богатое разнообразие жизни в регионах два века назад.

---

1 Habit of a Lady of Japan.

2 Collection of the Dress of Different Nations.

От издательства

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

Мы будем рады узнать ваше мнение!

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

1 Здравствуй, мир конкурентности в C++!

В этой главе

- Что подразумевается под конкурентностью и многопоточностью.
- Зачем использовать конкурентность и многопоточность в ваших приложениях.
- История поддержки конкурентности в C++.
- Как выглядит простая многопоточная программа на C++.

Для пользователей C++ наступили интересные времена. Спустя 13 лет после опубликования стандарта C++ (это произошло в 1998 году) Комитет по стандартизации C++ полностью переработал язык и поддерживающую его библиотеку. Новый стандарт C++, обозначаемый C++11 или C++0x, был опубликован в 2011 году и принес с собой ряд изменений, сделавших работу с языком более простой и продуктивной. Комитет также обязался разработать новую «модель передач» выпусков, согласно которой новый стандарт C++ будет публиковаться каждые три года. Пока вышли две такие публикации: C++14 в 2014 году и C++17 — в 2017-м, а также несколько технических спецификаций с описаниями расширений стандарта C++.

Из самых значимых новшеств в стандарте C++11 стоит отметить поддержку многопоточных программ. Впервые Комитет признал существование многопоточных приложений на этом языке и добавил в библиотеку средства для написания многопоточных приложений. Появилась возможность писать на C++ многопоточные

программы с гарантированным поведением, не полагаясь на платформенные расширения. Все это было очень вовремя, так как программисты, стремясь повысить производительность приложений, все чаще стали обращать внимание на конкурентность в целом и многопоточное программирование в частности. Стандарты C++14 и C++17 обеспечивают поддержку создания многопоточных программ на C++, но уже при наличии соответствующих технических спецификаций. Есть техническая спецификация для расширений, призванных выполнять конкурентные вычисления, и еще одна спецификация для параллельных вычислений (включена только в C++17).

В этой книге рассматриваются создание конкурентных программ на C++ с использованием нескольких потоков, а также свойства языка C++ и библиотечные средства, обеспечивающие соответствующие возможности. Сначала я объясню, что подразумевается под конкурентностью и многопоточностью и что может побудить вас использовать конкурентность в своих приложениях. После краткого обзора причин, по которым вы можете отказаться от ее применения в приложениях, будет вкратце рассмотрена поддержка конкурентности в C++, а в завершение главы приведу пример простой конкурентной программы на языке C++. Читатели, имеющие опыт разработки многопоточных приложений, могут пропустить первые разделы. В последующих главах мы рассмотрим более сложные примеры и подробно изучим средства библиотеки. В конце книги будут детально описаны все возможности стандартной библиотеки C++, позволяющие реализовать многопоточность и конкурентность.

Итак, что же понимается под *конкурентностью* (concurrency) и *многопоточностью* (multithreading)?

1.1. Что такое конкурентность

На самом элементарном уровне конкурентность — это одновременное выполнение двух и более отдельных задач. За примерами из обычной жизни далеко ходить не надо: мы можем одновременно идти и разговаривать, а еще выполнять разные действия левой и правой руками; каждый из нас живет своей жизнью независимо от других людей — пока я буду плавать, вы можете смотреть футбол и т. д.

1.1.1. Конкурентность в компьютерных системах

Когда речь заходит о конкурентности в контексте компьютеров, подразумевается отдельно взятая система, выполняющая несколько независимых задач в параллельном, а не последовательном (одна за другой) режиме. Эта идея не нова. Многозадачные операционные системы, позволяющие настольному компьютеру запускать одновременно несколько приложений с помощью переключения задач были обычным явлением на протяжении многих лет, как и высокопроизводительные серверы с несколькими процессорами, обеспечивавшие действительно параллельные

вычисления. Новым является массовое распространение компьютеров, способных по-настоящему выполнять несколько задач в параллельном режиме, а не создавать иллюзию этого процесса.

Исторически сложилось так, что у большинства настольных компьютеров был один процессор с одним вычислительным блоком или ядром, да и сегодня для многих настольных компьютеров ничего в этом смысле не изменилось. Такая машина в конкретный момент времени выполняет только одну задачу, но может переключаться между задачами множество раз в секунду. При выполнении небольшой части одной задачи, затем небольшой части другой и т. д. создается впечатление, что задачи решаются одновременно. Это называется *переключением задач*. Имея в виду такие системы, мы все же говорим о *конкурентности*, поскольку переключение задач происходит так быстро, что невозможно сказать, в какой момент задача может быть приостановлена из-за переключения процессора на выполнение другой задачи. Переключение между задачами создает иллюзию конкурентности как для пользователя, так и для самих приложений. Поскольку это всего лишь иллюзия таких вычислений, поведение приложений при выполнении в среде переключения задач с одним процессором может немного отличаться от их поведения при выполнении в среде с истинным параллелизмом. В частности, неверные допущения о модели памяти (рассматриваемой в главе 5) могут не проявляться в такой среде. Более подробно этот вопрос обсуждается в главе 10.

Компьютеры с несколькими процессорами в течение многих лет использовались в качестве серверов и для решения вычислительных задач, для которых требуется высокопроизводительное оборудование, а компьютеры на базе процессоров с несколькими ядрами на одном кристалле (многоядерные процессоры) теперь получают все большее распространение в качестве настольных. Независимо от того, имеют компьютеры несколько процессоров или несколько ядер в процессоре (или и то и другое), они способны выполнять несколько задач в параллельном режиме. Это называется *аппаратной конкурентностью*.

На рис. 1.1 схематически показан сценарий работы компьютера с двумя задачами, каждая из которых разделена на десять одинаковых этапов. На двухъядерной машине (имеющей два вычислительных ядра) каждая задача может выполняться в собственном ядре. На одноядерном компьютере, переключающем задачи, этапы задач чередуются. Кроме того, они немного разнесены по времени, что показано серыми разделительными полосами, более толстыми по сравнению с разделительными полосами на схеме для двухъядерной машины. Чтобы выполнить чередование, система при каждом переходе от одной задачи к другой должна *переключать контекст*, на что требуется время. Чтобы переключить контекст, операционная система должна сохранить состояние центрального процессора и указатель команд для выполняемой в данный момент задачи, определить, на какую задачу переключиться, и перезагрузить состояние центрального процессора для задачи, на которую происходит переключение. В таком случае центральному процессору, возможно, придется загружать в кэш-память инструкции и данные для новой задачи, что может помешать ему выполнять любые другие инструкции и привести к еще большей задержке.

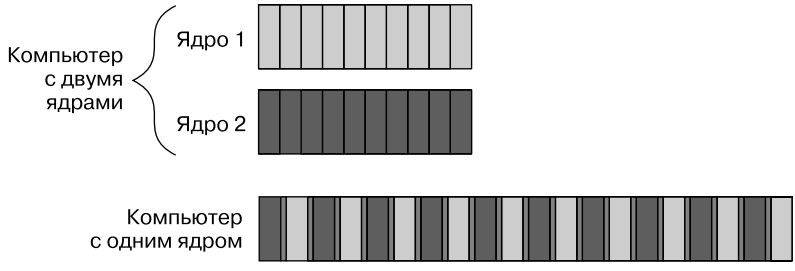


Рис. 1.1. Два подхода к конкурентности: параллельное выполнение на компьютере с двумя ядрами и переключение задач на одноядерной машине

Хотя возможности конкурентности нагляднее проявляются в многопроцессорных или многоядерных системах, некоторые процессоры могут выполнять несколько потоков и на одном ядре. Следует учитывать очень важный фактор — количество *аппаратных потоков*, то есть то количество независимых задач, которое оборудование может выполнить действительно одновременно. В то же время в системе с истинным аппаратным параллелизмом количество задач может превышать количество ядер, и тогда будет применяться механизм переключения задач. Например, на обычном настольном компьютере могут выполняться сотни задач, работающих в фоновом режиме, даже если машина номинально простаивает. Именно благодаря переключению эти задачи могут выполняться параллельно и можно одновременно открыть текстовый процессор, компилятор, редактор и браузер (или любую комбинацию приложений). На рис. 1.2 показано такое переключение между четырьмя задачами на двухъядерном компьютере, опять же для идеализированного сценария, где задачи делятся на этапы одинаковой длительности. На практике в силу многих причин деление не будет таким равномерным, а переключение — таким распределенным. Некоторые из этих причин рассматриваются в главе 8 при изучении факторов, влияющих на производительность конкурентного кода.

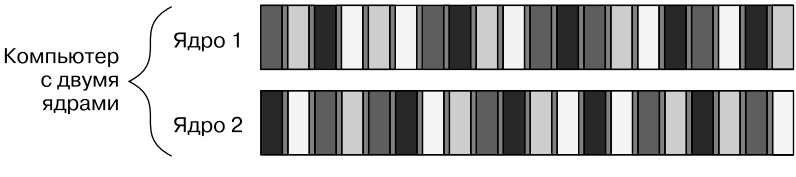


Рис. 1.2. Переключение при решении четырех задач на двух ядрах

Все технические приемы, функции и классы, рассматриваемые в данной книге, можно задействовать независимо от того, на каком компьютере выполняется ваше приложение — с одним одноядерным процессором или с несколькими многоядерными процессорами. И неважно, как именно реализуется конкурентность — путем переключения задач или за счет настоящего аппаратного параллелизма. Но, наверное, понятно, что способ обеспечения конкурентности в вашем приложении может зависеть от доступного оборудования. Соответствующие вопросы

рассматриваются в главе 8 при изучении задач, связанных с разработкой на C++ конкурентного кода.

1.1.2. Подходы к конкурентности

Представьте двух программистов, работающих вместе над одним программным проектом. Если они находятся в разных кабинетах, то могут работать спокойно, не мешая друг другу и пользуясь каждый своим набором справочников. Но вот общаться им непросто. Вместо того чтобы развернуться лицом друг к другу и поговорить, они вынуждены звонить по телефону, писать по электронной почте или же вставать и идти в другой кабинет к коллеге. Кроме того, приходится нести затраты на аренду двух помещений и приобретение нескольких экземпляров справочников.

А теперь представьте, что разработчики сидят в одном кабинете. Они могут свободно общаться, обсуждать проект приложения, рисовать схемы на бумаге или доске, обмениваться своими идеями.

Теперь нужно содержать только один кабинет, и зачастую достаточно будет только одного набора ресурсов. В то же время разработчикам может быть труднее сосредоточиться, и вполне реальны проблемы с общими ресурсами («Куда опять подевался справочник?»).

Эти два способа организации труда разработчиков позволяют проиллюстрировать два основных подхода к конкурентности. Каждый разработчик представляет поток, а каждый кабинет — процесс. Первый подход состоит в том, чтобы иметь несколько однопоточных процессов, что похоже на работу каждого разработчика в собственном кабинете, а второй подход предполагает наличие нескольких потоков в одном процессе, что похоже на работу двух разработчиков в одном помещении.

На такой основе можно составлять какие угодно комбинации и иметь несколько процессов, часть которых являются многопоточными, а часть — однопоточными, но действуют по одним и тем же принципам. Теперь кратко рассмотрим эти два подхода к конкурентности в приложении.

Конкурентность с использованием нескольких процессов

Первый способ применения параллелизма в приложении — разделение приложения на несколько конкурентных процессов с одним потоком. Например, так происходит при одновременном запуске браузера и текстового процессора. В дальнейшем отдельные процессы могут пересылать сообщения по всем обычным каналам межпроцессного обмена данными (используя сигналы, сокеты, файлы, конвейеры и т. д.) (рис. 1.3). Один из недостатков способа заключается в том, что либо такую связь между процессами сложно установить, либо она слишком медленно работает, либо наблюдается и то и другое. Так происходит из-за того, что операционные системы при обмене данными между процессами обычно предпринимают множество защитных мер, чтобы один процесс не мог случайно изменить данные, принадлежащие другому процессу. Еще одним недостатком является то, что выполнение нескольких процессов сопряжено с накладными расходами: на

запуск процесса требуется время, операционная система для управления им должна выделять внутренние ресурсы и т. д.

Но не все так плохо: операционные системы с дополнительной защитой обычно обеспечивают взаимодействие процессов и механизмов обмена данными более высокого уровня, а это значит, что конкурентный код может быть проще создавать с процессами, а не с потоками. И действительно, в среде, подобной той, что предусмотрена для языка программирования Erlang (www.erlang.org/), в качестве основного механизма конкурентности весьма эффективно используются именно процессы.

У использования отдельных процессов для реализации конкурентности есть и дополнительное преимущество — процессы можно запускать на разных компьютерах, входящих в одну сеть. Хотя это и увеличивает затраты на обмен данными, в качественно спроектированной системе способ может стать экономически эффективным для повышения степени конкурентности и увеличения производительности.

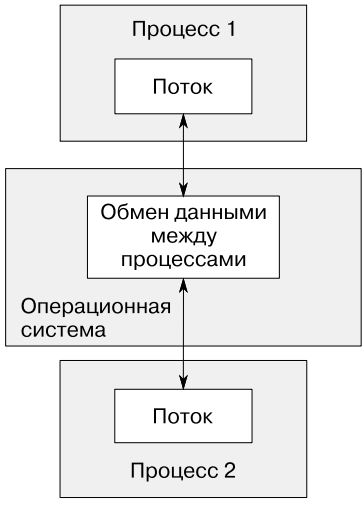


Рис. 1.3. Обмен данными между двумя одновременно запущенными процессами

Конкурентность с применением нескольких потоков

Альтернативный подход к конкурентности состоит в выполнении нескольких потоков в одном процессе. Потоки очень похожи на облегченные процессы: каждый из них реализуется независимо от других и может выполнять собственную последовательность инструкций. Но все потоки в процессе используют общее адресное пространство, и непосредственный доступ к основной части данных можно получить из всех потоков. При этом глобальные переменные остаются глобальными, а указатели или ссылки на объекты или данные можно передавать между потоками. Для процессов можно организовать доступ к разделяемой памяти, но эта функция трудно поддается настройке и управлению, потому что адреса размещения в памяти одних и тех же элементов не обязательно будут одинаковыми для разных процессов. На рис. 1.4 показаны два потока внутри процесса, обменивающиеся данными через общую память.

Наличие общего адресного пространства и отсутствие защиты данных между потоками существенно сокращают издержки, связанные с использованием нескольких потоков, по сравнению с издержками при выполнении нескольких процессов, так как снижается нагрузка на операционную систему. Но гибкость, обусловленная применением общей памяти, сопряжена и с затратами: если доступ к данным получают несколько потоков, программист должен убедиться, что представление данных,

видимых каждым потоком, согласовано при каждом обращении к этим данным. Вопросы, связанные с обменом данными между потоками, используемые для их решения инструменты, а также рекомендации, которых следует придерживаться во избежание возникновения проблем, рассматриваются в книге повсеместно, особенно в главах 3–5 и 8. Проблемы вполне решаемы при условии, что при создании кода вы будете осмотрительны, а это означает, что обмен данными между потоками следует тщательно продумать.

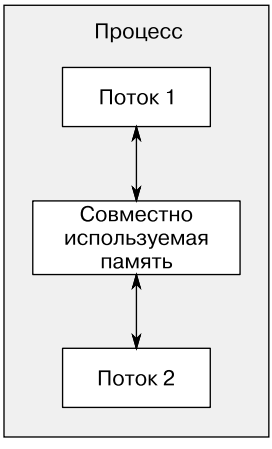


Рис. 1.4. Обмен данными между двумя потоками, запущенными одновременно в одном процессе

Низкие издержки на запуск потоков внутри процесса и обмен данными 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, стандарт C++ не предусматривает встроенной поддержки обмена данными между процессами, из-за чего приложениям с несколькими процессами придется полагаться на API, зависящие от платформы. Поэтому данная книга посвящена исключительно конкурентности на основе многопоточности, и в дальнейшем при любом упоминании конкурентности предполагается, что она реализуется использованием нескольких потоков.

Для многопоточного кода часто употребляется другое слово — *«параллелизм»*. Давайте уточним, в чем здесь различия.

1.1.3. Сравнение конкурентности и параллелизма

Для многопоточного кода значения понятий «конкурентность» и «параллелизм» в значительной степени пересекаются. Действительно, для многих они означают одно и то же. Разница в первую очередь заключается в нюансах, фокусе и намерениях. Оба термина относятся к одновременному выполнению нескольких задач с использованием доступного оборудования, но параллелизм более явно ориентирован на производительность. О *параллелизме* говорят, когда главной заботой становится использование имеющегося оборудования для повышения производительности массовой обработки данных, тогда как при *конкурентности* основная забота — разделение задач, требующих одновременного решения, или оперативность реагирования. Такая двойственность до сих пор существует, и значения этих двух понятий все еще существенно совпадают, но знать, в чем их различие, полезно при обсуждении рассматриваемых вопросов. В этой книге будут встречаться примеры использования обоих понятий.

Уточнив, что именно подразумевается под конкурентностью и параллелизмом, давайте выясним, зачем нужна конкурентность в приложениях.

1.2. Зачем используется конкурентность

Существует две основные причины применения параллелизма в приложении: разделение задач и повышение производительности. Возьму на себя смелость заявить, что это почти *единственные* причины использования параллелизма, все остальное, если присмотреться, сводится либо к одному, либо к другому (или даже и к тому и к другому, за исключением таких причин, как «потому что мне так хочется»).

1.2.1. Конкурентность для разделения неотложных задач

Разделение задач при создании программных средств — это практически всегда весьма рациональный подход. Сгруппировав связанные фрагменты кода и обособив несвязанные, проще понять и протестировать свои программы, к тому же так можно снизить вероятность появления ошибок. Конкурентность можно задействовать для разделения функциональных областей, даже если операции в этих отдельных областях должны выполняться одновременно. Без явного использования конкурентности пришлось бы либо создавать среду переключения задач, либо в ходе операции постоянно вызывать не связанные с этой операцией области кода.

Рассмотрим приложение, интенсивно обрабатывающее данные и имеющее пользовательский интерфейс, например приложение DVD-плеера для настольного компьютера. У него в основном два набора обязанностей. Оно должно не только считывать данные с диска, декодировать изображения и звук и своевременно отправлять их на монитор и звуковую карту, не допуская сбоев в воспроизведении DVD, но и принимать данные от пользователя, например, когда он щелкает на кнопках паузы, возврата в меню или даже выхода. В одном потоке во время воспроизведения приложение должно через равные промежутки времени проверять, что вводит пользователь, смешивая код воспроизведения DVD с кодом интерфейса пользователя. Используя многопоточность для разделения этих задач, можно избавиться от тесного переплетения кода пользовательского интерфейса и кода воспроизведения DVD: один поток может обрабатывать пользовательский интерфейс, а другой — воспроизведение DVD. Между ними должно быть организовано взаимодействие, например, когда пользователь нажимает кнопку паузы, но теперь эти моменты взаимодействия напрямую связаны с поставленной задачей.

Это создает иллюзию восприимчивости, поскольку поток пользовательского интерфейса обычно способен тут же ответить на запрос пользователя, даже если во время передачи запроса потоку, выполняющему работу, ответ должен отображать курсор в виде значка занятости или сообщение с просьбой подождать. Точно так же отдельные потоки часто используются для запуска задач, которые должны выполняться непрерывно в фоновом режиме, например отслеживания изменений файловой системы в приложении поиска на настольном компьютере. Как правило, подобное использование потоков существенно упрощает логику в каждом из них, поскольку взаимодействие между потоками может быть ограничено строго опреде-

ленными моментами, не переплетаясь с логикой различных задач. В данном случае количество потоков не зависит от количества доступных ядер центрального процессора, так как разделение на потоки основано на концептуальной схеме, а не на попытке увеличения пропускной способности.

1.2.2. Конкурентность для повышения производительности: параллелизм задач и данных

История многопроцессорных систем насчитывает десятилетия, но до недавнего времени они встречались в основном в суперкомпьютерах, мейнфреймах и крупных серверных системах. Производители микросхем все чаще отдают предпочтение многоядерным разработкам с 2, 4, 16 и более процессорами на одном кристалле, а не повышению производительности микропроцессоров с одним ядром. Следовательно, в наши дни все большее распространение получают многоядерные настольные компьютеры и даже многоядерные встраиваемые устройства. Вычислительная мощность этих машин увеличивается не за счет более быстрого выполнения одной задачи, а в результате параллельного выполнения сразу нескольких задач. В прошлом программисты могли сидеть сложа руки и смотреть, как без каких-либо усилий с их стороны с каждым новым поколением процессоров созданные ими программы работают быстрее. Но теперь, как сказал Херб Саттер (Herb Sutter), «бесплатных обедов больше не будет». Чтобы программы могли воспользоваться преимуществами увеличенной вычислительной мощности, их следует разрабатывать с прицелом на одновременное выполнение сразу нескольких задач. Поэтому программистам нужно принять это к сведению, а тем, кто до сих пор игнорировал конкурентность, — стремиться добавить ее в свой арсенал.

Существует два способа применения конкурентности для повышения производительности. Первый и наиболее очевидный — разбить одну задачу на части и выполнять их параллельно, сокращая тем самым общее время выполнения. Это *параллелизм задач*. На словах все довольно просто, но процесс может оказаться совсем не простым, потому что, возможно, между различными частями программы существует множество зависимостей. Разбиение на части возможно либо относительно обработки: один поток выполняет одну часть алгоритма, другой поток — другую, либо относительно данных: все потоки выполняют одну и ту же операцию с разными частями данных. Последний подход называется *параллелизмом данных*.

Алгоритмы, легко поддающиеся параллелизму, зачастую называют *сугубо параллельными*. Не стоит смущаться легкости распараллеливания такого кода, в ней нет ничего плохого. Мне встречались и другие названия таких алгоритмов: *естественно параллельные* и *удобно параллельные*. Сугубо параллельные алгоритмы обладают высокой масштабируемостью — по мере увеличения количества доступных аппаратных потоков параллелизм в алгоритме может быть соответственно увеличен. Подобный алгоритм идеально характеризуется изречением: «Одна голова хорошо,

а две — лучше». Те его части, которые не подпадают под определение сугубо параллельных, можно разделить на фиксированное (а следовательно, немасштабируемое) количество параллельно выполняемых задач. Методы разбиения задач на потоки рассматриваются в главах 8 и 10.

Второй способ применения конкурентности для повышения производительности — воспользоваться доступным параллелизмом для решения более сложных задач. Например, в зависимости от ситуации одновременно обрабатывать не один, а два, десять или двадцать файлов. Это не что иное, как *параллелизм данных*, то есть одновременное выполнение одной и той же операции с несколькими наборами данных. Есть и другой фокус. Обработка одного фрагмента данных по-прежнему занимает одно и то же время, но теперь в течение него можно обрабатывать больше данных. Очевидно, что у такого подхода есть ограничения и он не всегда будет приносить выгоду, но обусловленное им повышение пропускной способности может открыть новые возможности, например увеличится разрешение при обработке видео, если различные области изображения будут обрабатываться параллельно.

1.2.3. Когда не нужна конкурентность

Понимать, когда конкурентностью пользоваться не стоит, не менее важно. По сути, единственной причиной отказа от ее использования является отрицательный баланс выгод и затрат. Код, реализующий конкурентность, зачастую сложнее понять, поэтому создание и поддержка многопоточного кода сопряжены с прямыми интеллектуальными затратами, а дополнительная сложность может вызвать увеличение количества ошибок. Если потенциальный прирост производительности недостаточно велик или разбиение задач выражено нечетко и это не может оправдать дополнительные затраты времени на разработку, необходимые для достижения успеха, а также дополнительные затраты, связанные с поддержкой многопоточного кода, использовать конкурентность не имеет смысла.

Кроме того, прирост производительности может не оправдать ожиданий, ведь с запуском потока связаны некоторые издержки, поскольку операционная система должна выделить соответствующие ресурсы ядра и пространство стека, а затем добавить новый поток в диспетчер потоков, а на все это нужно время. Если выполняемая в потоке задача завершается довольно быстро, то время, затрачиваемое на саму задачу, может быть нивелировано издержками на запуск потока, что способно ухудшить общую производительность приложения, по сравнению с выполнением задачи в том потоке, который породил поток конкретно под нее.

Кроме того, потоки являются ограниченным ресурсом. Одновременный запуск нескольких потоков потребляет ресурсы операционной системы и может замедлить работу системы в целом. При этом использование слишком большого количества потоков может исчерпать доступную память или адресное пространство процесса, поскольку каждый поток требует отдельного пространства стека. Это особенно актуально для 32-разрядных процессов с плоской архитектурой, где имеется ограничение 4 Гбайт доступного адресного пространства. Если каждый поток имеет

стек 1 Мбайт (во многих системах именно так), то адресное пространство будет использоваться с 4096 потоками, не оставляя места для кода, статических данных или данных кучи. Хотя 64-разрядные или более крупные системы не имеют данного прямого ограничения адресного пространства, их ресурсы по-прежнему ограниченны: если запускается слишком много потоков, то в конечном итоге возникают проблемы. Несмотря на то что для ограничения количества потоков можно воспользоваться пулами потоков (см. главу 9), панацеей они не являются, так как и у них есть проблемы.

Если серверная часть клиент-серверного приложения запускает отдельный поток для каждого подключения, то для небольшого числа подключений будет обеспечена нормальная работа. Но если этот метод используется для сервера запросов, обрабатывающего множество подключений, то при запуске слишком большого количества потоков весьма высока вероятность исчерпания системных ресурсов. В таком случае обеспечить оптимальную производительность поможет аккуратное использование пулов потоков (см. главу 9).

И наконец, чем больше потоков запущено, тем больше переключений контекста должна выполнять операционная система. Каждое такое переключение занимает время, которое можно было бы потратить на полезную работу, поэтому в какой-то момент добавление дополнительного потока снизит общую производительность приложения, а не увеличит ее. Следовательно, при попытке получить максимально возможную производительность системы необходимо настроить количество запускаемых потоков, чтобы учесть доступный аппаратный параллелизм или его отсутствие.

Использование конкурентности для повышения производительности похоже на любую другую стратегию оптимизации: оно может существенно повысить производительность приложения, но способно и усложнить код, затруднить его понимание и повысить вероятность ошибок. Поэтому применять конкурентность стоит только для тех критически важных для производительности частей приложения, где можно получить ощутимую выгоду. Конечно, если потенциал прироста производительности менее важен, чем ясность конструкции или разделение задач, все же стоит воспользоваться многопоточной конструкцией.

Предположим, вы уже решили распараллелить свое приложение либо для повышения производительности, либо для разделения задач, либо просто «в рамках кампании по применению многопоточности». Что это будет означать для программистов на C++?

1.3. Конкурентность и многопоточность в C++

Стандартная поддержка конкурентности с помощью многопоточности относительно нова для C++. Возможность создать многопоточный код, не прибегая к расширениям для конкретной платформы, появилась только с выходом стандарта C++11. Чтобы понять, чем обоснованы многие решения в стандартной библиотеке потоков C++, важно знать историю их появления.

1.3.1. История поддержки многопоточности в C++

В стандарте C++ 1998 года существование потоков не признается, а действия различных элементов языка описываются в понятиях последовательной абстрактной машины. Более того, модель памяти формально не определена, поэтому создавать многопоточные приложения без характерных для компилятора расширений стандарта C++ 1998 года невозможно.

Поставщики компилятора могут свободно добавлять расширения языка, а распространенность API языка C для многопоточности, например, в стандарте POSIX C и API Windows компании Microsoft побудила многих поставщиков компилятора C++ поддерживать многопоточность с различными расширениями, специфичными для конкретных платформ. Эта поддержка компилятора, как правило, ограничивается при наличии нескольких потоков разрешением использовать соответствующий API языка C для платформы и обеспечением работы библиотеки времени выполнения C++ (например, кода для механизма обработки исключений). Хотя формальная многопоточная модель памяти была предоставлена немногими поставщиками компиляторов, поведение компиляторов и процессоров позволяло создать множество многопоточных программ на C++.

Не удовлетворившись применением платформенно-зависимых API языка C для работы с многопоточностью, программисты на C++ захотели, чтобы в используемых ими библиотеках классов были реализованы объектно-ориентированные средства для написания многопоточных программ. Среды разработки, например MFC, и универсальные библиотеки C++, такие как Boost и ACE, накопили в себе наборы классов C++, в которые заключались базовые API для конкретных платформ и которые предоставляли высокоуровневые средства, упрощающие решение задач применения многопоточности. Хотя конкретные детали библиотек классов существенно различаются, особенно в области запуска новых потоков, в целом форма классов имеет много общего. Одной из важнейших конструктивных особенностей, общей для многих библиотек классов C++ и дающей программисту существенные преимущества, является использование способа Resource Acquisition Is Initialization (RAII) (получение ресурса через инициализацию с блокировками), чтобы гарантировать разблокировку мьютексов при выходе из соответствующей области.

Во многих случаях поддержка многопоточности существующих компиляторов C++ в сочетании с доступностью API для конкретных платформ и библиотек классов, не зависящих от конкретной платформы, таких как Boost и ACE, обеспечивает прочную основу для написания многопоточного кода на C++. Как следствие, возможно, существуют миллионы строк кода на C++, написанных в качестве части многопоточных приложений. Но отсутствие стандартной поддержки означает, что в ряде случаев при отсутствии модели памяти с поддержкой потоков у программистов возникают проблемы. Особенно у тех, кто пытается повысить производительность, используя знания об аппаратных особенностях процессора, или пишет кроссплатформенный код, в котором поведение компиляторов варьируется в зависимости от используемой платформы.

1.3.2. Поддержка конкурентности в стандарте C++11

С выпуском стандарта C++11 все изменилось. В нем есть не только модель памяти, ориентированная на использование многопоточности, но и стандартная библиотека C++, включающая классы для управления потоками (см. главу 2), защиту совместно используемых данных (см. главу 3), синхронизацию операций между потоками (см. главу 4) и низкоуровневые атомарные операции (см. главу 5).

Библиотека потоков C++11 базируется на предыдущем опыте, накопленном благодаря использованию упомянутых ранее библиотек классов C++. В частности, в качестве модели, на которой основана новая библиотека, задействовалась библиотека потоков Boost, причем имена и структуры многих классов перекликались с именами и структурами соответствующих классов из Boost. Развитие стандарта превратило этот процесс в двусторонний, в рамках которого изменялась и сама библиотека потоков Boost, чтобы во многих отношениях соответствовать стандарту C++. Поэтому пользователям, ранее применявшим Boost, многое должно быть уже знакомо.

Как упоминалось в начале главы, с выходом стандарта C++11 в язык было внесено множество улучшений, облегчающих жизнь программистов, и поддержка конкурентности — одно из них. Хотя другие изменения зачастую не относятся к тематике данной книги, некоторые из них непосредственно повлияли на библиотеку потоков и способы ее использования. Краткий обзор этих функций языка дается в приложении A.

1.3.3. Расширение поддержки конкурентности и параллелизма в C++14 и C++17

Единственным добавлением в C++14, касавшимся конкретной поддержки конкурентности и параллелизма, стал новый тип мьютекса для защиты совместно используемых данных (см. главу 3). С выходом C++17 новинок стало значительно больше: для начинающих появился полный набор параллельных алгоритмов (см. главу 10). Оба стандарта улучшили основной язык и стандартную библиотеку в целом, и эти улучшения могут существенно упростить создание многопоточного кода.

Как уже упоминалось, для конкурентности имеется техническая спецификация, в которой дается описание расширений функций и классов, предоставляемых стандартом C++, особенно имеющих отношение к синхронизации операций между потоками (см. главу 4).

Поддержка непосредственно в C++ атомарных операций позволяет программистам создавать эффективный код с определенной семантикой, не требуя применения языка ассемблера конкретной платформы. Это настоящая находка для тех, кто стремится писать эффективный переносимый код, причем компилятор не только обращает внимание на специфику платформы, но и дает возможность создать оптимизатор, учитывающий семантику операций, что способно оптимизировать программу в целом.

1.3.4. Эффективность, обеспеченная библиотекой потоков C++

У разработчиков, занимающихся высокопроизводительными вычислениями, зачастую вызывает вопросы эффективность языка C++ в целом и, в частности, тех классов C++, которые служат оболочкой для низкоуровневых средств, имеющихся в новой стандартной библиотеке потоков C++. Стремясь достичь максимального уровня производительности, важно понимать различие между затратами на реализацию с применением любых средств высокого уровня и реализацию непосредственного использования базовых низкоуровневых средств. Эти затраты являются *платой за абстракцию*.

Специалисты Комитета по стандартизации C++ знали об этом, разрабатывая стандартную библиотеку C++ в целом и стандартную библиотеку потоков C++ в частности. Одной из целей проектирования библиотек было сведение практически к нулю выгоды от непосредственного использования низкоуровневых API там, где должно быть предоставлено аналогичное средство. Поэтому библиотека была разработана с прицелом на обеспечение эффективной реализации (и низкой платы за абстракцию) на большинстве основных платформ.

Другой целью Комитета по стандартизации C++ было добиться того, чтобы язык C++ обеспечивал достаточные возможности низкоуровневого программирования для всех, кто захочет для достижения максимальной производительности вести разработку с учетом конкретного оборудования. Поэтому наряду с новой моделью памяти поставляется и комплексная библиотека атомарных операций, допускающая непосредственное управление отдельными битами и байтами, а также синхронизацию между потоками и отображение любых изменений. Теперь эти атомарные типы и соответствующие операции можно задействовать во многих местах, где разработчики ранее предпочли бы перейти на язык ассемблера конкретной платформы. Код, использующий новые стандартные типы и операции, отличается большей степенью переносимости и простотой поддержки.

Стандартная библиотека C++ предоставляет также высокоуровневые абстракции и средства, которые облегчают создание многопоточного кода и уменьшают вероятность возникновения в нем ошибок. Порой использование этих средств приводит к снижению производительности, поскольку приходится выполнять дополнительный код. Но это не обязательно становится следствием более высокой платы за абстракцию. В целом ущерб не выше, чем при создании эквивалентных функциональных средств вручную, а встроить объемный дополнительный код компилятор может и в любом другом случае.

В ряде случаев высокоуровневые средства предоставляют дополнительные функциональные возможности, выходящие за рамки того, что может понадобиться в конкретном случае. Чаще всего это не вызывает никаких проблем: вы не несете особых расходов на то, что не используете. Но в редких случаях эти неиспользуемые функциональные возможности влияют на производительность другого кода. Когда нужен высокий уровень производительности, а стоимость применения стандартных средств слишком высока, возможно, стоит создать нужные функциональные сред-

ства вручную, собрав их из объектов более низкого уровня. В подавляющем большинстве случаев усложнение кода и вероятность увеличения количества ошибок существенно перевешивают потенциальные выгоды от незначительного прироста производительности. Даже если профилирование показывает, что узким местом являются средства стандартной библиотеки C++, это может быть связано с неудачной конструкцией приложения, а не с плохой реализацией библиотеки. Например, то, что за мьютекс конкурирует слишком много потоков, негативно влияет на производительность. Вероятно, выгоднее было бы не пытаться сэкономить незначительное время на операциях мьютекса, а реструктурировать приложение, чтобы уменьшить число конфликтов, связанных с мьютексом. Разработка приложений с прицелом на уменьшение количества конфликтов рассматривается в главе 8.

В тех редких случаях, когда стандартная библиотека C++ не обеспечивает нужные производительность или поведение, может потребоваться использовать средства, ориентированные на применение конкретной платформы.

1.3.5. Средства, ориентированные на использование конкретной платформы

Хотя библиотека потоков C++ дает довольно широкие возможности для применения многопоточности и конкурентности, на любой отдельно взятой платформе будут средства, зависящие только от нее и выходящие за рамки предлагаемого библиотекой. Чтобы обеспечить возможность получить легкий доступ к этим средствам, не отказываясь от преимуществ задействования стандартной библиотеки потоков C++, типы в библиотеке потоков C++ предусматривают компонентную функцию `native_handle()`, позволяющую работать на уровне API для конкретной платформы. По своей природе любые операции, выполняемые с помощью `native_handle()`, полностью зависят от платформы и выходят за рамки тематики данной книги (и самой стандартной библиотеки C++).

Прежде чем начать присматриваться к возможностям использования средств, учитывающих особенности конкретной платформы, важно понять, что именно обеспечивает стандартная библиотека, поэтому начнем с примера.

1.4. Приступаем к практической работе

Итак, у вас в наличии великолепный компилятор C++11/C++14/C++17. Что дальше? Как выглядит многопоточная программа на C++? Она похожа на любую другую программу на C++ с обычным сочетанием переменных, классов и функций. Единственное реальное отличие заключается в том, что некоторые функции могут выполняться одновременно, поэтому необходимо обеспечить безопасность совместно используемых данных при конкурентном доступе к ним в соответствии с инструкциями, изложенными в главе 3. Чтобы одновременно запустить функции для управления различными потоками, необходимо воспользоваться особыми функциями и объектами.

1.4.1. Здравствуй, мир конкурентности

Начнем с классического примера — программы для вывода фразы Hello World. Далее показана простая программа Hello World, которая выполняется в одном потоке и служит базой для перехода к использованию нескольких потоков:

```
#include <iostream>
int main()
{
    std::cout<<"Hello World\n";
}
```

В этой программе в стандартный поток вывода записывается фраза Hello World. Сравним ее с простой программой Hello Concurrent World, показанной в листинге 1.1 и запускающей для отображения сообщения отдельный поток.

Листинг 1.1. Простая программа Hello Concurrent World

```
#include <iostream>
#include <thread>
void hello()
{
    std::cout<<"Hello Concurrent World\n";
}
int main()
{
    std::thread t(hello);
    t.join();
}
```

Первым ее отличием является использование директивы `#include <thread>`. Объявления, предназначенные для поддержки многопоточности в стандартной библиотеке C++, представлены в новых заголовках: функции и классы для управления потоками объявляются в `<thread>`, а функции и классы для защиты совместно используемых данных — в других заголовках.

Второе отличие — перемещение кода для записи сообщения в отдельную функцию. Это связано с тем, что каждый поток должен иметь *исходную функцию*, с которой начинается новый поток выполнения. Для начального потока в приложении это `main()`, а для любого другого потока такая функция указывается в конструкторе объекта `std::thread`. В данном случае это объект `std::thread` с именем `t`, имеющий в качестве исходной новую функцию `hello()`.

И в этом заключается следующее отличие: вместо непосредственной записи в стандартный вывод или вызова `hello()` из `main()` эта программа запускает для выполнения своей задачи новый поток, в результате чего потоков становится два — исходный поток, который начинается с `main()`, и новый, начинающийся с `hello()`.

После запуска нового потока исходный продолжает выполняться. Если бы он не ждал завершения нового потока, то запросто продолжил бы работу до конца `main()` и завершил программу, возможно, еще до того, как смог бы запуститься новый поток. Вот почему в соответствии с положениями, которые рассматриваются в главе 2, мы

добавили вызов `join()` – он заставляет вызывающий поток (`main()`) ожидать тот поток, что связан с объектом `std::thread`, — в данном случае поток `t`.

Если вам показалось, что для элементарного вывода сообщения в стандартный вывод работы слишком много, то так оно и есть, — в подразделе 1.2.3 выше мы говорили, что обычно для решения такой простой задачи нет смысла создавать несколько потоков, особенно если главному потоку в это время нечего делать. Далее в книге будут рассмотрены примеры сценариев, в которых выгода от применения нескольких потоков не вызывает никаких сомнений.

Резюме

В этой главе мы выяснили, что подразумевается под конкурентностью и многопоточностью, и определили причины, по которым стоит или не стоит использовать их в создаваемых вами приложениях. Была также рассмотрена история применения многопоточности в C++, начиная с полного отсутствия поддержки в стандарте 1998 года, появления в дальнейшем различных расширений для конкретных платформ и заканчивая надлежащей поддержкой многопоточности в стандарте C++11 и стандартах C++14 и C++17, а также выработкой технической спецификации по конкурентности. Эта поддержка оказалась весьма своевременной. Она позволила программистам воспользоваться преимуществами аппаратного параллелизма, которые стали доступны в современных процессорах, поскольку их производители пошли по пути наращивания мощности за счет реализации нескольких ядер, а не увеличивая быстродействие одного ядра.

Кроме этого, на примерах из раздела 1.4 вы увидели, как просто использовать классы и функции из стандартной библиотеки C++. Применение в C++ нескольких потоков не представляет особой сложности. Труднее разработать код, который будет вести себя именно так, как задумано.

После рассмотрения примеров из раздела 1.4 пришло время для чего-то более существенного. В главе 2 поговорим о классах и функциях, доступных для управления потоками.

2 Управление потоками

В этой главе

- Запуск потоков и различные способы задания кода, исполняемого в новом потоке.
- Ожидать завершения потока или разрешить ему оставаться запущенным?
- Уникальная идентификация потоков.

Итак, вы решили воспользоваться конкурентностью для своего приложения. В частности, задействовать несколько потоков. И что дальше? Как запустить эти потоки, убедиться в их завершении и проследить за ними? Благодаря применению стандартной библиотеки C++ большинство задач управления потоками решить довольно легко, поскольку почти все управление осуществляется с использованием объекта `std::thread`, связанного, как будет показано, с заданным потоком. Для задач с более сложным решением библиотека допускает проявление гибкости, позволяя создавать из базовых строительных блоков нужные вам конструкции.

Сначала в этой главе будут рассмотрены основы: запуск потока, ожидание его завершения или запуск в фоновом режиме. Затем поговорим о передаче дополнительных параметров функции `thread` при ее запуске и передаче права владения потоком от одного объекта `std::thread` другому. И в завершение рассмотрим порядок выбора количества используемых потоков и идентификации конкретных потоков.

2.1. Основы управления потоками

У каждой программы на C++ есть как минимум один поток, запускаемый средой выполнения C++, — поток, выполняющий функцию `main()`. Затем программа может запустить дополнительные потоки, точкой входа в которые служит другая функция. После чего эти потоки и начальный поток выполняются одновременно. Аналогично завершению программы при выходе из `main()` поток завершается при возвращении из функции, указанной в качестве точки входа. Далее будет показано, что при наличии для потока объекта `std::thread` можно дождаться завершения этого потока. Однако сначала его нужно запустить, поэтому посмотрим на запуск потоков.

2.1.1. Запуск потока

В главе 1 было показано, что потоки запускаются созданием объекта `std::thread`, в котором определяется выполняемая в потоке задача. В простейшем случае эта задача представляет собой обычную функцию, которая возвращает `void`-значение, и не принимает никаких параметров. Эта функция выполняется в собственном потоке, пока не вернет управление, после чего поток останавливается. В иных случаях задача может быть функциональным объектом, принимающим дополнительные параметры и выполняющим ряд независимых операций, которые задаются посредством некоей системы обмена сообщениями в ходе его выполнения. Поток останавливается только при получении соответствующего сигнала, опять же через некую систему сообщений. Что бы ни собирался делать поток и откуда бы он ни запускался, его запуск с использованием стандартной библиотеки C++ всегда сводится к созданию объекта `std::thread`:

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

Проще, наверное, и быть не может. Конечно, чтобы компилятор мог видеть определение класса `std::thread`, нужно убедиться, что в программу включен заголовок `<thread>`. Как и основная часть содержимого стандартной библиотеки C++, `std::thread` работает с любым вызываемым типом, поэтому конструктору `std::thread` можно также передать экземпляр класса с оператором вызова функции:

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
background_task f;
std::thread my_thread(f);
```

В данном случае предоставленный функциональный объект копируется в хранилище, принадлежащее вновь созданному потоку выполнения, и вызывается оттуда. Поэтому важно, чтобы копия действовала аналогично оригиналу, иначе результат может не соответствовать ожидаемому.

При передаче функционального объекта в конструктор потока следует избегать так называемого самого неприятного случая синтаксического анализа C++. Если передается временная, а не именованная переменная, синтаксис может быть таким же, как и при объявлении функции. В этом случае компилятор интерпретирует его именно таким образом, а не как определение объекта. Например, код:

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

объявляет функцию `my_thread`, принимающую единственный параметр наподобие указателя на функцию, не принимающую параметры и возвращающую объект фоновой задачи `background_task`, и вместо запуска нового потока возвращает объект `std::thread`. Такого развития событий можно избежать, присвоив функциональному объекту имя, как было показано ранее, с помощью дополнительных скобок или нового унифицированного синтаксиса инициализации, например:

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

В первом примере дополнительные скобки не позволяют интерпретировать код как объявление функции, разрешая объявить `my_thread` переменной типа `std::thread`. Во втором примере используется новый унифицированный синтаксис инициализации с фигурными, а не круглыми скобками, также позволяющий объявить переменную.

Одним из типов вызываемого объекта, позволяющего избегать возникновения рассматриваемой проблемы, является лямбда-выражение. Эта новая особенность, появившаяся в стандарте C++11, допускает создание локальной функции с возможным захватом некоторых локальных переменных, избавляя от необходимости передачи дополнительных аргументов (см. раздел 2.2). Более подробно лямбда-выражения рассматриваются в разделе A.5. С помощью лямбда-выражения предыдущий пример можно записать следующим образом:

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

Научившись запускать поток, нужно принять однозначное решение, ждать ли его завершения (присоединив его — см. подраздел 2.1.2) или пустить его на самотек (путем его отсоединения — см. подраздел 2.1.3). Если не принять решение до уничтожения объекта `std::thread`, то программа завершится (деструктор `std::thread` вызовет метод `std::terminate()`). Поэтому даже при выдаче исключений необходимо убедиться, что поток правильно присоединен или отсоединен. Методы, позволяющие справиться с таким сценарием, рассмотрены в подразделе 2.1.3. Учтите, что

решение нужно принимать до того, как объект `std::thread` будет уничтожен. Сам же поток вполне мог бы завершиться задолго до его присоединения или отсоединения. Если его отсоединить, то при условии, что он все еще выполняется, он и будет выполняться, и этот процесс может продолжаться еще долго и после уничтожения объекта `std::thread`. Выполнение будет прекращено, только когда в конце концов произойдет возвращение из функции потока.

Если не дожидаться завершения потока, необходимо убедиться, что данные, к которым он обращается, будут действительны, пока он не закончит работать с ними. Эта проблема не отличается новизной — даже в однопоточном коде поведение приложения при доступе к объекту после его уничтожения характеризуется неопределенностью, но при использовании потоков вероятность столкнуться с подобными проблемами времени жизни объектов возрастает.

К примеру, похожие проблемы могут возникнуть, если функция потока содержит указатели или ссылки на локальные переменные, а поток при выходе из функции не завершается. Пример такого сценария показан в листинге 2.1.

Листинг 2.1. Функция, возвращающая управление, в то время как в потоке сохраняется обращение к локальным переменным

```
struct func
{
    int& i;
    func(int& i_):i(i_){}
    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};
void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

Потенциальное обращение к недействительной ссылке

Отсутствие ожидания завершения потока

Возможно, новый поток все еще выполняется

В данном случае новый поток, связанный с `my_thread`, при выходе из oops, вероятно, все еще будет выполняться, поскольку, судя по вызову `detach()`, было принято решение не дожидаться его завершения. Если поток все еще выполняется, все идет по сценарию, показанному в табл. 2.1: следующий вызов `do_something(i)` будет обращаться к уже уничтоженной переменной. Все это похоже на обычный однопоточный код, в котором сохранять указатель или ссылку на локальную переменную после выхода из функции непозволительно, но вероятность допустить такую ошибку при использовании многопоточного кода значительно выше, поскольку ее негативные проявления видны не сразу.

Таблица 2.1. Обращение к локальной переменной в отсоединенном потоке после того, как она была уничтожена

Основной поток	Новый поток
Создание my_func со ссылкой на some_local_state	
Запуск нового потока my_thread	
	Запуск
	Вызов func::operator()
Отсоединение my_thread	Запуск func::operator(); возможен вызов do_something со ссылкой на some_local_state
Уничтожение some_local_state	Продолжение выполнения
Выход из oops	Продолжение выполнения func::operator(); возможен вызов do_something со ссылкой на some_local_state => неопределенное поведение

Справиться с подобным сценарием можно одним из распространенных способов: сделать функцию потока автономной и вместо разделения данных скопировать их в поток. При использовании вызываемого объекта для функции потока этот объект копируется в поток, поэтому исходный объект можно тут же уничтожить. Тем не менее следует опасаться объектов, содержащих указатели или ссылки, как, к примеру, в листинге 2.1. В частности, не рекомендуется создавать внутри функции поток, имеющий доступ к ее локальным переменным, если только не будет обеспечено завершение потока до выхода из этой функции.

Или же, *присоединив* поток до выхода из функции, можно убедиться в завершении его выполнения.

2.1.2. Ожидание завершения потока

Дождаться завершения потока можно, вызвав `join()` для связанного экземпляра `std::thread`. Что касается кода в листинге 2.1, то замены вызова `my_thread.detach()` перед закрывающей фигурной скобкой тела функции вызовом `my_thread.join()` было бы достаточно, чтобы гарантировать завершение потока до выхода из функции и, следовательно, до уничтожения локальных переменных. В данном случае это означало бы, что запускать функцию в отдельном потоке нет смысла, поскольку первый поток во время ожидания не будет выполнять никакой полезной работы. Но в реальном коде исходный поток будет либо выполнять полезную работу, либо запускать несколько потоков для выполнения полезной работы, прежде чем входить в режим ожидания их завершения.

Применение метода `join()` — простой и незамысловатый прием: либо вы ждете завершения потока, либо нет. Если нужен более утонченный контроль над ожиданием потока, допустим, с проверкой завершения потока или ожиданием в течение строго определенного времени, придется воспользоваться альтернативными механизмами, например условными переменными и фьючерсами, которые будут рассматриваться в главе 4. Вызов `join()` также приводит к очистке любого хранилища,

связанного с потоком, поэтому объект `std::thread` больше не связан с завершенным потоком. Мало того, он не связан ни с одним потоком. Это означает, что `join()` можно вызвать для конкретного потока только один раз: как только вызван метод `join()`, объект `std::thread` утрачивает возможность присоединения, а метод `joinable()` вернет значение `false`.

2.1.3. Ожидание в исключительных обстоятельствах

Как упоминалось ранее, следует убедиться, что вызов либо `join()`, либо `detach()` был выполнен еще до уничтожения объекта `std::thread`. При отсоединении потока метод `detach()` можно вызвать сразу же после запуска этого потока, так что это не проблема. Но если есть намерение дожидаться завершения потока, то место в коде, где будет выполнен вызов метода `join()`, нужно выбирать с особой тщательностью, поскольку при выдаче исключения после запуска потока, но до вызова `join()` вызов этого метода можно пропустить.

Чтобы избежать завершения приложения при выдаче исключения, необходимо решить, что делать в таком случае. Вообще, если вы намеревались вызвать `join()` при нормальном выполнении программы, вам, чтобы избежать проблем, связанных с временем жизни, необходимо вызвать `join()` и при выдаче исключения. Простой код, выполняющий именно такие действия, показан в листинге 2.2.

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();
        throw;
    }
    t.join();
}
```

Код в листинге 2.2 содержит блок `try-catch`, позволяющий гарантировать, что поток с доступом к локальному состоянию завершится до выхода из функции независимо от того, завершается ли функция нормально или из-за выдачи исключения. Использование блоков `try-catch` растягивает код, повышая вероятность получения неправильной области видимости, поэтому данный сценарий далек от идеала. Если есть веские причины обеспечить завершение потока до выхода из функции, скажем, из-за наличия ссылки на другие локальные переменные или

чего-нибудь еще, важно убедиться, что учтены все возможные пути выхода, как нормальные, так и исключительные, и желательно обеспечить для этого простой и лаконичный механизм.

Кроме иных способов, это можно сделать с использованием стандартного способа RAII и, как показано в листинге 2.3, предоставить класс, выполняющий метод `join()` в своем деструкторе. Посмотрите, насколько при этом упрощается функция `f()`.

Листинг 2.3. Использование RAII для ожидания завершения потока

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
    ~thread_guard()
    {
        if(t.joinable())
        {
            t.join();
        }
    }
    thread_guard(thread_guard const&)=delete;
    thread_guard& operator=(thread_guard const&)=delete;
};
struct func;
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);
    do_something_in_current_thread();
}
```

См. определение в листинге 2.1

Когда выполнение текущего потока достигает конца функции `f`, локальные объекты уничтожаются в порядке, обратном порядку их построения. Следовательно, сначала уничтожается объект `g` типа `thread_guard`, а в его деструкторе происходит присоединение к потоку. Это наблюдается даже при завершении выполнения функции, поскольку `do_something_in_current_thread` выдает исключение.

Перед вызовом метода `join()` деструктор `thread_guard` в листинге 2.3 сначала проверяет с помощью метода `joinable()`, является ли объект `std::thread` присоединяемым. Важность этого действия обусловливается тем, что метод `join()` для конкретного потока выполнения можно вызвать только один раз, поэтому, если поток уже был присоединен, его вызов станет ошибкой.

Копирующий конструктор и операторы копирующего присваивания помечают спецификатором `=delete`, чтобы запретить компилятору автоматически предоставлять их. Копирование или присваивание такого объекта было бы опасным, поскольку тогда время его жизни могло бы превысить время жизни присоединяемого потока.

При объявлении их удаленными любая попытка скопировать объект `thread_guard` будет приводить к выдаче ошибки компиляции. Более подробно удаленные функции рассматриваются в разделе A.2.

Если ждать завершения потока не нужно, проблемы безопасности исключений можно избежать, *отсоединив* этот поток. Тем самым разрывается его связь с объектом `std::thread` и гарантируется, что метод `std::terminate()` не будет вызван при удалении объекта `std::thread`, даже если поток все еще выполняется в фоновом режиме.

2.1.4. Запуск потоков в фоновом режиме

Вызов метода `detach()` для объекта `std::thread` позволяет потоку выполняться в фоновом режиме, непосредственное взаимодействие с ним не требуется. Возможность дождаться завершения этого потока исчезает: если поток отсоединяется, получить ссылающийся на него объект `std::thread` невозможно, поэтому такой поток больше нельзя присоединить. Отсоединенные потоки фактически выполняются в фоновом режиме, владение и управление ими передаются в библиотеку среды выполнения C++, которая гарантирует правильное высвобождение ресурсов, связанных с потоком, при выходе из него.

Отсоединенные потоки часто называются *потоками-демонами* по аналогии с существующей в UNIX концепцией *процесса-демона*, который выполняется в фоновом режиме без какого-либо явного пользовательского интерфейса. Как правило, такие потоки являются весьма продолжительными, работая в течение практически всего времени жизни приложения и выполняя фоновую задачу, например отслеживая состояние файловой системы, удаляя неиспользуемые записи из кэш-памяти объектов или оптимизируя структуры данных. В то же время есть смысл использовать отсоединенный поток, если существует иной механизм определения факта завершения потока или когда нужно запустить задачу и «забыть» о ней.

Как было показано в подразделе 2.1.2, поток отсоединяется путем вызова для объекта `std::thread` компонентной функции `detach()`. После завершения вызова объект `std::thread` больше не связан с фактическим потоком выполнения, поэтому данный поток больше нельзя присоединить:

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

Чтобы отсоединить поток от объекта `std::thread`, должен существовать отсоединяемый поток: метод `detach()` нельзя вызывать для объекта `std::thread`, не имеющего связанного с ним потока выполнения. Это требование аналогично тому, которое предъявляется к вызову метода `join()`, и проверку можно провести точно таким же образом — вызывать для объекта `t` типа `std::thread` метод `t.detach()` возможно, только если метод `t.joinable()` вернет значение `true`.

Рассмотрим приложение, к примеру текстовый процессор, позволяющий одновременно редактировать несколько документов. Есть множество способов

справиться с такой задачей как на уровне пользовательского интерфейса, так и на уровне внутренних механизмов. Один из них, получающий сейчас все более широкое распространение, — использование нескольких независимых окон верхнего уровня, по одному для каждого редактируемого документа. Хотя эти окна представляются полностью независимыми и каждое имеет собственные меню, они работают в одном и том же экземпляре приложения. Один из способов справиться с этим за счет внутренних механизмов — запустить все окна редактирования документа в собственных потоках. Каждый поток будет выполнять один и тот же код, но с разными данными, относящимися к редактируемому документу и соответствующими свойствам окна. Следовательно, открытие нового документа потребует запуска нового потока. Поток, обрабатывающий запрос, не станет дожидаться завершения этого другого потока, потому что он работает над другим, независимым документом, что делает новый поток основным кандидатом для запуска в отсоединенном режиме.

В листинге 2.4 показано общее представление кода реализации рассматриваемого подхода.

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);
            t.detach();
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

Если пользователь решает открыть новый документ, ему предлагается сделать это, после чего для открытия этого документа запускается новый поток, который затем отсоединяется. Поскольку новый поток выполняет ту же операцию, что и текущий, но в другом файле, той же функцией (`edit_document`) можно воспользоваться повторно, взяв в качестве аргумента новое выбранное имя файла.

В этом примере также показана ситуация, когда есть смысл передать аргументы в функцию, используемую для запуска потока: вместо простой передачи имени функции в конструктор `std::thread` ему передается параметр `filename`. Для этого можно использовать и другие механизмы, например вместо обычной функции с параметрами задействовать функциональный объект с компонентными данными, для чего стандартная библиотека C++ предоставляет очень простой способ.

2.2. Передача аргументов функции потока

В листинге 2.4 было показано, что передача аргументов вызываемому объекту или функции сводится к простой передаче дополнительных аргументов конструктору `std::thread`. Но важно учесть, что по умолчанию аргументы копируются во внутреннее хранилище, где к ним может получить доступ вновь созданный поток выполнения, а затем передаются вызываемому объекту или функции как *r*-значения (rvalues), как будто они временные. Так делается, даже если соответствующий параметр в функции ожидает ссылку. Рассмотрим пример:

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

В результате создается новый поток выполнения, связанный с `t`, который вызывает функцию `f(3,"hello")`. Обратите внимание: даже если `f` в качестве второго параметра принимает `std::string`, строковый литерал передается как `char const *` и преобразуется в `std::string` только в контексте нового потока. Это становится особенно важным, когда, как показано далее, предоставленный аргумент является указателем на автоматическую переменную:

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

Здесь это указатель на буфер локальной переменной, который передается в новый поток. И высока вероятность того, что выход из функции `oops` произойдет, прежде чем буфер будет в новом потоке преобразован в `std::string`, что вызовет неопределенное поведение. Решением является приведение к типу `std::string` перед передачей буфера в конструктор `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 предотвращает обращение к недействительному указателю

В данном случае причиной возникновения проблемы является надежда на подразумеваемое преобразование указателя на буфер в объект `std::string`, ожидаемый в качестве параметра функции, но это преобразование происходит слишком поздно, поскольку конструктор `std::thread` копирует предоставленные значения как есть, не преобразуя в ожидаемый тип аргумента.

Обратный сценарий получить невозможно: объект скопирован и вам нужна не-`const`-ссылка, поскольку она не скомпилируется. Все это можно попробовать, если поток обновляет структуру данных, переданную по ссылке, например:

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

Хотя `update_data_for_widget` ожидает, что второй параметр будет передан по ссылке, конструктор `std::thread` не знает об этом, он не обращает внимания на типы аргументов, которые ожидает функция, и слепо копирует предоставленные значения. Но внутренний код передает скопированные аргументы в качестве *r*-значений, чтобы работать с типами, предназначенными только для перемещений, и пытается таким образом вызвать `update_data_for_widget` с *r*-значением. Этот код не пройдет компиляцию, так как нельзя передать *r*-значение функции, ожидающей не-`const`-ссылку. Для тех, кто знаком с `std::bind`, решение будет очевидным: аргументы, которые должны быть ссылками, следует заключать в `std::ref`. В этом случае при изменении вызова потока на:

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

`update_data_for_widget` будет корректно передана ссылка на данные, а не временная копия данных и код успешно пройдет компиляцию.

Если работать с `std::bind` уже приходилось, то в семантике передачи параметров не будет ничего нового, поскольку и операция конструктора `std::thread`, и операция `std::bind` определены в рамках одного и того же механизма. То есть можно, например, передать указатель на компонентную функцию в виде функции при условии, что в качестве первого аргумента предоставлен подходящий указатель на объект:

```
class X
{
public:
    void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work,&my_x);
```

Этот код вызовет `my_x.do_lengthy_work()` в новом потоке, поскольку в качестве указателя на объект предоставляется адрес `my_x`. Такому вызову компонентной функции можно также предоставлять аргументы: третий аргумент конструктора `std::thread` будет первым аргументом компонентной функции и т. д.

Еще один интересный сценарий предоставления аргументов применяется, когда аргументы нельзя скопировать, а можно только *переместить*: данные, содержащиеся в одном объекте, передаются в другой, и исходный объект остается пустым.

Примером может послужить тип `std::unique_ptr`, обеспечивающий автоматическое управление памятью для динамически выделяемых объектов. В одно и то же время на данный объект может указывать только один экземпляр `std::unique_ptr`, и когда этот экземпляр уничтожается, объект, на который он указывал, удаляется. *Перемещающий конструктор* и *перемещающий оператор присваивания* позволяют передавать права владения объектом между экземплярами `std::unique_ptr` (более подробно семантика перемещения рассматривается в подразделе А.1.1. В результате этого исходный объект остается с нулевым указателем. Такое перемещение значений позволяет принимать объекты данного типа в качестве параметров функции или возвращать их из функций. Если исходный объект временный, перемещение выполняется автоматически, но если источником является именованное значение, передача должна быть запрошена напрямую путем вызова метода `std::move()`. В следующем примере показано использование `std::move` для передачи потоку права владения динамическим объектом:

```
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 при вызове конструктора `std::thread` указан метод `std::move(p)`, право владения `big_object` сначала передается внутреннему хранилищу вновь созданного потока, а затем переходит к `process_big_object`.

Некоторые классы в стандартной библиотеке C++ имеют ту же семантику прав владения, что и `std::unique_ptr`, и одним из них является `std::thread`. Хотя экземпляры `std::thread` не владеют, как `std::unique_ptr`, динамическим объектом, они владеют ресурсом: каждый экземпляр отвечает за управление потоком выполнения. Это право собственности разрешается передавать между экземплярами, потому что экземпляры `std::thread` можно переместить, даже если их нельзя скопировать. Тем самым гарантируется, что в любой момент связь с конкретным потоком выполнения будет только у одного объекта, что позволяет программистам передавать это право собственности между объектами.

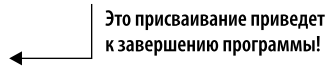
2.3. Передача права владения потоком

Предположим, что нужно написать функцию, которая создает поток для выполнения в фоновом режиме, но не ожидает его завершения, а передает право собственности на новый поток обратно вызывающей функции, или же сделать обратное: создать поток и передать владение им некоторой функции, которая должна дождаться его завершения. В любом случае нужно передать право собственности из одного места в другое.

Именно здесь и пригодится поддержка перемещения, имеющаяся в `std::thread`. Как говорилось в предыдущем разделе, перемещаться, но не копироваться могут многие типы стандартной библиотеки C++, владеющие ресурсами, например `std::ifstream` и `std::unique_ptr`, и одним из таких типов является `std::thread`.

А это значит, что владение конкретным потоком выполнения можно перемещать между экземплярами `std::thread`, что и показано в следующем примере. В нем создаются два потока выполнения, а права собственности на них передаются между тремя экземплярами `std::thread` — `t1`, `t2` и `t3`:

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```



Сначала запускается новый поток, который связывается с `t1`. Затем при создании `t2` путем вызова `std::move()` для явного перемещения владельца право собственности передается `t2`. В этот момент у `t1` больше нет связанного потока выполнения, и поток, выполняющий `some_function`, теперь связан с `t2`.

Затем запускается новый поток, связанный с временным объектом типа `std::thread`. Последующая передача права собственности в `t1` не требует вызова `std::move()` для его явного перемещения, потому что владелец является временным объектом, а передача права собственности временными объектами подразумевается и выполняется автоматически.

Объект `t3` создан с установками по умолчанию, то есть без связанного с ним потока выполнения. Право собственности на поток, который в данный момент связан с `t2`, передается `t3`, для чего опять применяется явный вызов `std::move()`, поскольку `t2` — именованный объект. После всех этих перемещений `t1` ассоциируется с потоком, выполняющим функцию `some_other_function`, у `t2` связанного потока нет, а `t3` связан с потоком, выполняющим `some_function`.

Последнее перемещение переносит владение потоком, выполняющим `some_function`, обратно в `t1`, откуда он начинался. Но в данном случае объект `t1` уже имел связанный с ним поток, который выполнял `some_other_function`, поэтому для завершения программы вызывается метод `std::terminate()`.

Это делается для обеспечения согласованности с деструктором `std::thread`. В подразделе 2.1.1 было показано, что нужно ожидать явного завершения или отсоединения потока перед уничтожением, то же самое относится и к присваиванию: нельзя просто удалить поток, присвоив управляющему им объекту `std::thread` новое значение.

Реализованная в `std::thread` поддержка перемещения означает, что право собственности можно легко передать из функции (листинг 2.5).

Листинг 2.5. Возвращение 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 владение должно быть передано функции, то, как показано далее, она может принять экземпляр `std::thread` по значению в качестве одного из параметров:

```
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 поддержки перемещения `std::thread` — возможность надстроить класс `thread_guard`, показанный в листинге 2.3, и принудить его к владению потоком. Это позволит избежать неприятных последствий в том случае, если объект `thread_guard` переживет поток, на который ссылается. Это также означает, что никто другой не может присоединить или отсоединить поток после того, как право собственности на него передано объекту. Поскольку цель всего этого — в первую очередь обеспечить завершение потоков до выхода из области видимости, я назвал данный класс `scoped_thread`. Реализация и простой пример показаны в листинге 2.6.

Листинг 2.6. Класс scoped_thread и пример его применения

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_):
        t(std::move(t_))
    {
        if(!t.joinable())
        throw std::logic_error(“No thread”);
    }
    ~scoped_thread()
    {
        t.join();
    }
    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))};
    do_something_in_current_thread();
}
```

Код примера похож на приведенный в листинге 2.3, но новый поток вместо создания для него отдельной именованной переменной передается непосредственно в `scoped_thread`. Когда исходный поток достигает конца `f`, объект `scoped_thread` уничтожается, а затем этот поток присоединяется к потоку, переданному конструктору. В ходе работы с классом `thread_guard` из листинга 2.3 деструктор должен был проверить, что поток все еще является присоединяемым, а здесь это можно сделать в конструкторе, выдав в случае отрицательного результата исключение.

Одно из предложений для C++17 касалось класса `joining_thread`, который был бы похож на `std::thread`, за исключением того, что автоматически включался бы в деструктор так же, как это делает `scoped_thread`. Но к согласию в Комитете прийти не удалось, и это предложение не было включено в стандарт (хотя в виде класса `std::jthread` оно все еще актуально для перспективного стандарта C++20), но его относительно несложно создать. Одна из возможных реализаций показана в листинге 2.7.

Листинг 2.7. Класс joining_thread

```
class joining_thread
{
    std::thread t;
public:
    joining_thread() noexcept=default;
    template<typename Callable,typename … Args>
    explicit joining_thread(Callable&& func,Args&& … args):
        t(std::forward<Callable>(func),std::forward<Args>(args)…)
    {}
    explicit joining_thread(std::thread t_) noexcept:
        t(std::move(t_))
    {}
    joining_thread(joining_thread&& other) noexcept:
        t(std::move(other.t))
    {}
    joining_thread& operator=(joining_thread&& other) noexcept
    {
        if(joinable())
            join();
        t=std::move(other.t);
        return *this;
    }
    joining_thread& operator=(std::thread other) noexcept
    {
        if(joinable())
            join();
        t=std::move(other);
        return *this;
    }
    ~joining_thread() noexcept
    {
        if(joinable())
            join();
```

```
    }
    void swap(joining_thread& other) noexcept
    {
        t.swap(other.t);
    }
    std::thread::id get_id() const noexcept{
        return t.get_id();
    }
    bool joinable() const noexcept
    {
        return t.joinable();
    }
    void join()
    {
        t.join();
    }
    void detach()
    {
        t.detach();
    }
    std::thread& as_thread() noexcept
    {
        return t;
    }
    const std::thread& as_thread() const noexcept
    {
        return t;
    }
};
```

Поддержка перемещения в `std::thread` допускает также использование контейнеров объектов `std::thread`, если эти контейнеры поддерживают перемещения, например применение обновленного `std::vector<>`. Это позволяет создавать код из листинга 2.8 — он порождает несколько потоков, а затем ожидает их завершения.

Листинг 2.8. Порождение 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)
    {
        threads.emplace_back(do_work,i);  <-- Порождение потоков
    }
    for(auto& entry: threads)  <-- Поочередный вызов join() для каждого потока
        entry.join();
}
```

Если потоки, как часто требуется, используются для разделения работы алгоритма, то перед возвратом управления вызывающему коду все они должны быть

завершены. Простая структура листинга 2.8 подразумевает, что работа, выполняемая потоками, автономна, а результатом их операций являются только побочные эффекты для совместно используемых 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 после анализа разделяемых данных по завершении всех потоков. Альтернативные схемы передачи результатов операций между потоками рассматриваются в главе 4.

Помещение объектов `std::thread` в `std::vector` — шаг к автоматизации управления этими потоками: вместо создания для них отдельных переменных и непосредственного соединения с ними можно рассматривать их как группу. Или пойти дальше, создав динамическое количество потоков, определяемое в ходе выполнения программы, а не фиксированное, как в листинге 2.8.

2.4. Выбор количества потоков в ходе выполнения программы

Одна из функций стандартной библиотеки C++, помогающая решить данную задачу, — `std::thread::hardware_concurrency()`. Она возвращает то количество потоков, которые действительно могут работать одновременно в ходе выполнения программы. Например, в многоядерной системе оно может быть увязано с числом ядер центрального процессора. Функция дает всего лишь подсказку и может вернуть `0`, если информация недоступна, но ее данные способны принести пользу при разбиении задачи на потоки.

В листинге 2.9 показана простая реализация параллельной версии `std::accumulate`. В реальном коде потребуется, скорее всего, не самостоятельно создавать эту функцию, а использовать параллельную версию `std::reduce`, рассматриваемую в главе 10, но данная реализация служит иллюстрацией основной идеи. Она распределяет работу между потоками с минимальным количеством элементов на каждый из них, чтобы избежать высоких издержек на слишком большое количество потоков. Заметьте, здесь предполагается, что ни одна из операций не выдаст исключение, даже если они возможны. Конструктор `std::thread`, к примеру, будет выдавать исключение, если не сможет запустить новый поток выполнения. В этом простом примере обработка исключений в подобном алгоритме не предусмотрена (она рассматривается в главе 8).

Листинг 2.9. Простейшая параллельная версия 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)
        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);
    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);
        threads[i]=std::thread(
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;
    }
    accumulate_block<Iterator,T>()(
        block_start,last,results[num_threads-1]);

    for(auto& entry: threads)
        entry.join();
    return std::accumulate(results.begin(),results.end(),init);
}
```

Несмотря на большой объем кода, в этой функции нет ничего сложного. Если входной диапазон пуст, возвращается исходное значение, указанное в качестве значения параметра `init`. В противном случае в диапазоне есть хотя бы один элемент, поэтому, чтобы получить максимальное количество потоков, количество обрабатываемых элементов можно разделить на минимальный размер блока. Это сделано для того, чтобы не создавать 32 потока на 32-ядерном компьютере, если в диапазоне только пять значений.

Количество запускаемых потоков — это минимальное значение из полученного в результате расчетов максимума и количества аппаратных потоков. Не следует запускать больше потоков, чем может поддерживать имеющееся оборудование (приводя к появлению так называемого *превышения лимитов*), поскольку переключение контекста при превышении количества потоков повлечет за собой снижение производительности. Если при вызове `std::thread::hardware_concurrency()` возвращается `0`, количество выбирается по вашему усмотрению, в данном случае я выбрал 2. Запускать слишком много потоков нежелательно, поскольку на одноядерном компьютере

это замедлит работу, но нежелательно запускать и слишком мало потоков, потому что будет не реализована возможность использования доступной конкурентности.

Количество элементов, обрабатываемых каждым потоком, равно длине диапазона, разделенной на количество потоков. Если число не делится без остатка, не стоит беспокоиться, с этой ситуацией мы справимся чуть позже.

После определения количества существующих потоков можно создать `std::vector<T>` для промежуточных результатов и `std::vector<std::thread>` для потоков. Следует отметить, что нужно запустить на один поток меньше, чем значение `num_threads`, поскольку один поток уже имеется.

Запуск потоков представлен простым циклом: итератор `block_end` сдвигается в конец текущего блока, и для накопления результатов для данного блока запускается новый поток. Окончание текущего блока служит началом следующего.

После того как вы запустили все потоки, данный поток может обработать последний блок. Здесь учитывается любое деление с остатком: известно, что за окончанием последнего блока ничего быть не должно, и количество элементов в этом блоке уже не имеет значения.

После того как накоплены результаты для последнего блока, можно дождаться завершения всех потоков, созданных, как в листинге 2.8, с помощью `std::for_each`, а затем сложить результаты с финальным вызовом `std::accumulate`.

Прежде чем распрощаться с этим примером, стоит отметить, что там, где оператор сложения для типа `T` не является ассоциативным (например, для `float` или `double`), результаты этого `parallel_accumulate` могут отличаться от результатов `std::accumulate` из-за разбиения диапазона на блоки. Кроме того, к итераторам предъявляются несколько более жесткие требования: они должны быть по меньшей мере *однонаправленными*, `std::collectulate` может работать с однопроходными *входными итераторами*, а `T` должен *допускать конструирование по умолчанию*, чтобы можно было создать вектор результатов `results`. Изменения требований такого рода общие для параллельных алгоритмов: чтобы стать параллельными, они по своей природе должны быть другими, что отражается на результатах и требованиях. Более подробно реализация параллельных алгоритмов рассматривается в главе 8, а глава 10 посвящена стандартным средствам, имеющимся в C++17 (эквивалентом рассмотренного здесь `parallel_accumulate` является параллельная форма `std::reduce`). Стоит также отметить, что невозможность возвращать значение непосредственно из потока вынуждает передавать ссылку на соответствующую запись в векторе результатов `results`. Альтернативные способы возвращения результатов из потоков с помощью фьючерсов рассматриваются в главе 4.

В данном случае вся информация, необходимая каждому потоку, включая место, где будет храниться результат его вычисления, была передана при запуске потока. Но так бывает не всегда, и порой в процессе работы необходимо каким-то образом идентифицировать потоки. Можно, как в листинге 2.8, передать идентификационный номер в виде значения `i`, но если функция, нуждающаяся в идентификаторе, находится в стеке вызовов глубже на несколько уровней и может быть вызвана из любого потока, то делать это таким образом неудобно. Эта необходимость была нами предусмотрена при разработке стандартной библиотеки C++, поэтому уникальный идентификатор имеется у каждого потока.

2.5. Идентификация потоков

Идентификаторы потоков относятся к типу `std::thread::id`, и их можно получить двумя способами. Во-первых, идентификатор потока можно получить из связанного с ним объекта `std::thread`, вызвав компонентную функцию `get_id()`. Если у объекта `std::thread` нет связанного с ним потока выполнения, вызов `get_id()` возвращает созданный по умолчанию объект `std::thread::id`, который означает «не поток». Во-вторых, идентификатор можно получить, если вызвать функцию `std::this_thread::get_id()`, которая также определена в заголовке `<thread>`.

Объекты типа `std::thread::id` можно свободно копировать и сравнивать, иначе в качестве идентификаторов от них не было бы никакой пользы. Если два объекта типа `std::thread::id` равны, значит, они представляют один и тот же поток или оба имеют значение «не поток». Если два объекта не равны, они представляют разные потоки или один представляет поток, а другой имеет значение «не поток».

Стандартная библиотека C++ не ограничивает возможности лишь проверкой того, одинаковы идентификаторы потоков или нет: объекты типа `std::thread::id` предлагают полный набор операторов сравнения, предоставляющих весь спектр возможностей по упорядочению всех отличающихся друг от друга значений. Поэтому их можно использовать в качестве ключей в ассоциативных контейнерах, сортировать или сравнивать любым другим способом, который программист сочтет нужным. Операторы сравнения обеспечивают полноценное упорядочение всех неравных значений `std::thread::id`, поэтому они ведут себя интуитивно предсказуемым образом: если `a<b` и `b<c`, то `a<c` и т. д. Стандартная библиотека также предоставляет класс `std::hash<std::thread::id>`, поэтому значения типа `std::thread::id` можно использовать в качестве ключей в новых неупорядоченных ассоциативных контейнерах.

Экземпляры `std::thread::id` часто применяются для проверки необходимости выполнения потоком какой-либо операции. Например, если потоки, как в листинге 2.9, используются для разделения работы, то исходному потоку, запустившему другие потоки, может потребоваться выполнить свою работу в середине алгоритма немного иначе. В данном случае он может сохранить результат `std::this_thread::get_id()` перед запуском других потоков, а затем основная часть алгоритма, которая является общей для всех потоков, может проверить собственный идентификатор потока на предмет соответствия сохраненному значению:

```
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();
}
```

Как вариант, `std::thread::id` текущего потока может храниться в структуре данных как часть операции. Затем последующие операции с той же структурой

данных могут сравнить сохраненный идентификатор на предмет соответствия идентификатору потока, выполняющего операцию, чтобы определить, какие операции разрешены или необходимы.

Точно так же идентификаторы потока можно применять в качестве ключей в ассоциативных контейнерах, где конкретные данные должны быть связаны с потоком, а альтернативные варианты, такие как локальное хранилище потока, для этого не подходят. Такой контейнер может, к примеру, использоваться управляющим потоком для хранения информации о каждом из потоков, находящихся под его управлением, или для передачи информации между ними.

Суть в том, что в большинстве случаев `std::thread::id` в качестве общего идентификатора для потока вполне достаточно, альтернативный вариант может понадобиться, только если идентификатор должен иметь связанное с ним семантическое значение, например, должен быть индексом в массиве. Экземпляр `std::thread::id` можно даже записать в выходной поток, например в `std::cout`:

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

Конкретный вывод полностью зависит от реализации, стандарт лишь гарантирует, что идентификаторы потоков, определяемые при сравнении как одинаковые, выдают одинаковый вывод, а неодинаковые — различный. Следовательно, в первую очередь это пригодится при отладке и ведении учетных записей, но значения не несут никакой семантической нагрузки, и сделать на их основе дополнительные выводы невозможно.

Резюме

В этой главе мы рассмотрели основы управления потоками с применением стандартной библиотеки C++: запуск потоков, ожидание их завершения и работу без такого ожидания, если нужно, чтобы они выполнялись в фоновом режиме. Было также показано, как при запуске потока передавать аргументы в функцию потока, а ответственность за управление потоком — из одной части кода в другую и как группы потоков можно использовать для разделения работы. И наконец, мы обсудили механизм идентификации потоков, позволяющий ассоциировать с потоком данные или поведение в тех случаях, когда использовать другие средства неудобно. Конечно же, многое можно сделать и с полностью независимыми потоками, каждый из которых будет работать с отдельными данными, но порой желательно, чтобы запущенные потоки могли работать с общими для них данными. В главе 3 будут обсуждаться вопросы, связанные с разделением данных непосредственно между потоками, а в главе 4 — более общие вопросы, связанные с синхронизацией операций с совместно используемыми данными и без них.

3 Совместное использование данных несколькими потоками

В этой главе

- Проблемы, связанные с совместным использованием данных несколькими потоками.
- Защита данных с помощью мьютексов.
- Альтернативные средства защиты совместно используемых данных.

Одним из ключевых преимуществ применения потоков для конкурентности является возможность совместного использования (разделения) данных несколькими потоками, поэтому теперь, рассмотрев запуск потоков и управление ими, обратимся к вопросам, связанным с разделением данных.

Представьте на минуту, что вы живете в одной квартире с приятелем. У вас одна кухня и одна ванная на двоих. Обычно ванной не пользуются одновременно несколько человек, и то, что сосед слишком долго плещется в воде, вынуждая вас дожидаться своей очереди, не может не раздражать. Возможно, одному из вас захочется запечь в духовке колбаски, в то время как у другого там готовятся кексы, и из этого тоже не выйдет ничего хорошего. Ну и всем знакомо чувство досады, когда при совместно используемом оборудовании вы на полпути к решению какой-нибудь задачи вдруг обнаруживаете, что кто-то взял что-то нужное вам в данный момент или что-то изменил, а вы рассчитывали, что все останется в прежнем состоянии или на своих местах.

То же самое происходит и с потоками. Если они совместно используют данные, для них нужны правила, определяющие, какой поток и к какому биту данных может получить доступ, когда и как любые обновления данных будут передаваться другим потокам, интересующимся этими данными. Но не стоит радоваться, что можно так легко разделять данные между потоками в одном процессе. Некорректное применение совместных данных — одна из основных причин ошибок, связанных с конкурентностью, и последствия могут быть куда хуже, чем кексы со вкусом колбасы.

Эта глава посвящена безопасной совместной работе с данными нескольких потоков в программах на C++, а также исключению возможных проблем и извлечению максимальных выгод.

3.1. Проблемы совместного использования данных несколькими потоками

Когда дело доходит до совместной работы с данными нескольких потоков, то все проблемы возникают из-за последствий изменения этих данных. *Если все совместно используемые данные доступны только для чтения, проблем не будет, поскольку данные, считываемые одним потоком, не зависят от того, читает другой поток те же данные или нет*. Но если один или несколько потоков, совместно использующих данные, начинают вносить в них изменения, создаются серьезные предпосылки для возникновения проблем. В таком случае следует обеспечить приемлемость конечных результатов.

Очень широкое распространение получило понятие *инвариантов*, позволяющее программистам рассуждать о коде. Это утверждения относительно конкретной структуры данных, не вызывающие сомнений, например «эта переменная содержит количество элементов в списке». В результате обновлений инварианты часто нарушаются, особенно если у структуры повышенный уровень сложности или обновление требует изменения более чем одного значения.

Рассмотрим список, имеющий две связи, в котором каждый узел содержит указатель как на следующий, так и на предыдущий узел. Один из инвариантов заключается в том, что если с узла *А* пойти по указателю «следующий» к узлу *Б*, то указатель узла *Б* «предыдущий» будет указывать на узел *А*. Чтобы удалить узел из списка, необходимо обновить узлы с обеих сторон, чтобы они указывали друг на друга. Как только один из узлов был обновлен, инвариант нарушается до тех пор, пока так же не будет обновлен узел на другой стороне. После завершения обновления инвариант снова соблюдается.

Этапы удаления записи из такого списка показаны на рис. 3.1.

1. Идентификация удаляемого узла — *N*.
2. Обновление ссылки в узле, предшествующем *N*, чтобы она указывала на узел, находящийся после *N*.

3. Обновление ссылки в узле после *N*, чтобы она указывала на узел, предшествующий узлу *N*.
4. Удаление узла *N*.

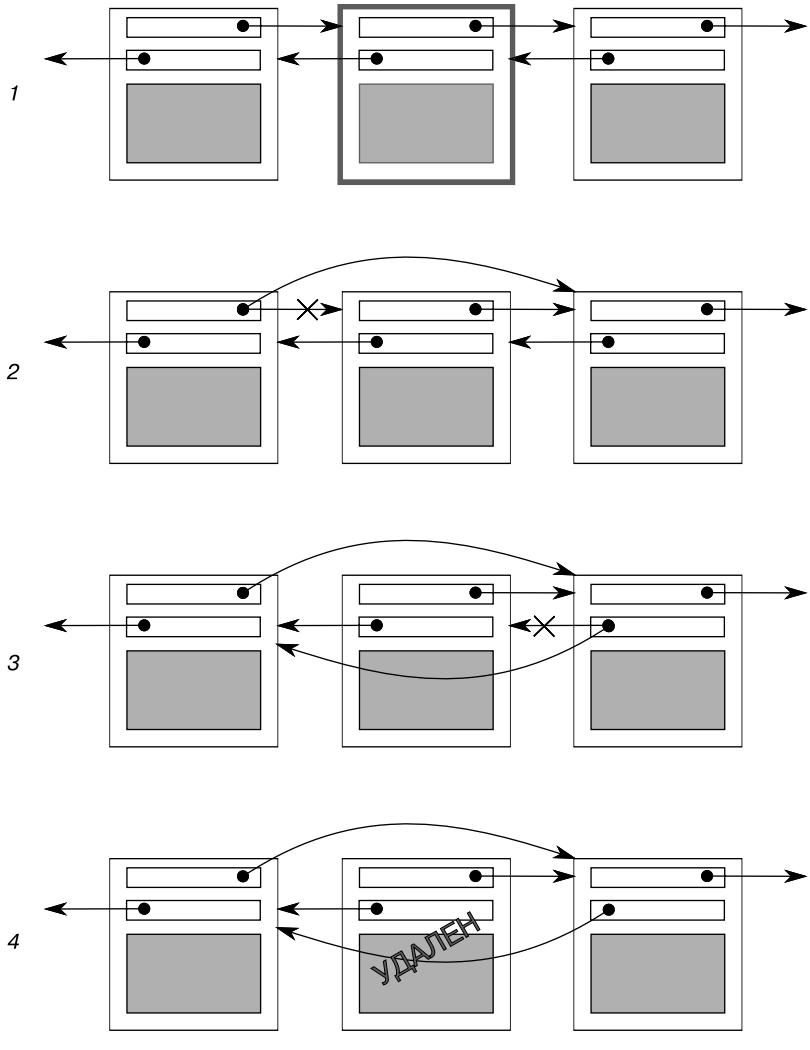


Рис. 3.1. Удаление узла из двусвязного списка

На рис. 3.1 показано, что между этапами *1* и *3* ссылки, указывающие на перемещение в прямом направлении, не согласуются со ссылками, указывающими на перемещение в обратном направлении, то есть в данном случае инвариант нарушен.

Простейшая потенциальная проблема, связанная с данными, совместно используемыми несколькими потоками, — это проблема нарушенных инвариантов.

Если не предпринять ничего, что могло бы гарантировать обратное, то в ситуации, когда один поток читает двусвязный список, а другой удаляет узел, вполне возможно, что читающий поток может увидеть список с удаленным узлом лишь частично (поскольку, как на этапе *2*, была изменена только одна из ссылок), следовательно, инвариант нарушен. Последствия такого нарушения инварианта могут различаться: если на схеме другой поток читает элементы списка слева направо, то он пропустит удаляемый узел. В то же время, если второй поток попытается удалить самый правый узел на схеме, это может привести к необратимому повреждению структуры данных и в конечном итоге к сбою программы. Каким бы ни был результат, это пример одной из наиболее распространенных причин ошибок в конкурентном коде — *состояние гонки*, обусловливающее неопределенность итога.

3.1.1. Состояние гонки

Предположим, вы покупаете билет в кино. Если кинотеатр большой, билеты будут продавать сразу несколько кассиров, обслуживая одновременно несколько человек. Если кто-то в это же время покупает билет на тот же сеанс в другой кассе, то выбор места зависит от того, кто первым его закажет, вы или другой. Если осталось всего несколько мест, очередность может стать решающей: возможна настоящая гонка за последними билетами. Это пример *состояния гонки*: какое место вы получите и получите ли вообще, зависит от порядка двух покупок.

При конкурентности состоянием гонки является все, что зависит от порядка выполнения операций в двух и более потоках относительно друг друга: потоки участвуют в гонке по выполнению соответствующих операций. В большинстве случаев все заканчивается благополучно, поскольку приемлемы все возможные результаты, даже если их порядок меняется. Например, если два потока добавляют элементы в очередь обработки, то при условии сохранения инвариантов системы обычно не имеет значения, какой элемент добавляется первым. Проблема возникает, когда состояние гонки нарушает инварианты, как в приведенном ранее примере двусвязного списка. Когда речь заходит о конкурентности, выражение «состояние гонки» обычно используется для обозначения *проблемного* состояния: благополучно разрешаемое состояние гонки не вызывает интереса и не становится причиной ошибок. В стандарте C++ также определяется понятие *гонки за данными*, обозначающее конкретный тип состояния гонки, возникающий из-за одновременного изменения одного и того же объекта (подробности рассматриваются в подразделе 5.1.2). Гонки за данными вызывают опасное *неопределенное поведение*.

Критические состояния гонки обычно возникают, когда для завершения операции требуется изменение двух и более отдельных частей данных, таких как две ссылки в рассмотренном ранее примере. Поскольку операция должна получить доступ к двум отдельным частям данных, они должны быть изменены в отдельных инструкциях, и потенциально получить доступ к структуре данных, когда завершена только одна из них, может другой поток. Из-за узости окна возможностей состояние гонки зачастую трудно определить и сложно воспроизвести. Если изменения вносятся последовательными инструкциями центрального процессора, вероятность возникновения проблемы при любом проходе невелика, даже если к структуре дан-

ных в то же самое время обращается другой поток. По мере увеличения нагрузки на систему и количества выполнения операции возрастает и вероятность возникновения проблемной последовательности выполнения. Проблемы практически неизменно появляются в самое неподходящее время. Поскольку обычно состояние гонки возникает в конкретный период времени, при запуске приложения под отладчиком оно вообще может исчезать, поскольку отладчик, пусть и незначительно, но все же влияет на время выполнения программы.

При написании многопоточных программ состояние гонки может стать настоящим мучением, основные сложности создания программ, использующих конкурентность, связаны с поисками способов обхода проблемных состояний гонок.

3.1.2. Пути обхода проблемных состояний гонок

Есть несколько способов, позволяющих справиться с проблемными состояниями гонок. Самый простой вариант — заключить структуру данных в механизм защиты, чтобы гарантировать, что промежуточные состояния, в которых нарушены инварианты, будут видны только потоку, выполняющему изменения. С позиции других потоков, обращающихся к этой же структуре данных, такие изменения либо еще не начнутся, либо уже завершатся. Стандартная библиотека C++ предоставляет несколько таких механизмов, рассматриваемых в данной главе.

Есть еще один вариант — изменить конструкцию структуры данных и ее инвариантов так, чтобы модификации вносились в виде серии неделимых изменений, каждая из которых сохраняет инварианты. Обычно эта процедура называется *программированием без блокировок* (lock-free programming), и реализовать ее нелегко. В ходе работы на этом уровне возможны сложности с особенностями модели памяти и определением того, каким именно потокам какие наборы значений могут быть видны. Модель памяти рассматривается в главе 5, а программирование без блокировки — в главе 7.

Еще один способ, позволяющий справиться с состоянием гонки, заключается в обновлении структуры данных в форме *транзакции*, так же как это делается при обновлениях баз данных. Требуемая последовательность изменений и считываний данных сохраняется в журнале транзакций, а затем фиксируется в рамках единой операции. Если зафиксировать ее невозможно из-за того, что структура данных была изменена другим потоком, транзакция перезапускается. Такой способ называется *программной транзакционной памятью* (software transactional memory, STM), на момент написания данной книги в этой области велись весьма активные исследования. Непосредственная поддержка STM в языке C++ отсутствует (хотя есть техническая спецификация для транзакционных расширений памяти — Transactional Memory Extensions[1]), и в книге этот способ рассматриваться не будет. Но к идее выполнения каких-либо действий в частном порядке с последующим одномоментным завершением мы еще вернемся.

---

[1] ISO/IEC TS 19841:2015 — Technical Specification for C++ Extensions for Transactional Memory http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=66343.

Основным механизмом защиты совместно используемых данных, обеспеченным стандартом C++, является *мьютекс*, поэтому сначала будет рассмотрен именно он.

3.2. Защита совместно используемых данных с применением мьютексов

Итак, имеется совместно используемая структура данных, например связанный список из предыдущего раздела, и его нужно защитить от состояния гонки и возможных нарушений инвариантов. Наверное, неплохо было бы получить возможность помечать все фрагменты кода, обращающиеся к структуре данных, как *взаимоисключающие*, чтобы при выполнении одного из них каким-либо потоком любой другой поток, пытающийся получить доступ к этой структуре данных, был бы вынужден ждать, пока первый поток не завершит выполнение такого фрагмента. Тогда поток не смог бы увидеть нарушенный инвариант, кроме тех случаев, когда он сам выполнял бы модификацию.

Данное желание не из разряда сказочных, именно это будет получено при использовании примитива синхронизации под названием «*мьютекс*», означающего *взаимное исключение* (*mut*ual *ex*clusion). Перед получением доступа к совместно используемой структуре данных мьютекс, связанный с ней, *блокируется*, а когда доступ к ней заканчивается, *блокировка с него снимается*. Библиотека потоков гарантирует, что, как только один поток заблокирует определенный мьютекс, все остальные потоки, пытающиеся его заблокировать, должны будут ждать, пока поток, который успешно заблокировал мьютекс, его не разблокирует. Тем самым гарантируется, что все потоки видят непротиворечивое представление совместно используемых данных без нарушенных инвариантов.

Мьютексы — главный механизм защиты данных, доступный в C++, но панацеей от всех бед их не назовешь: важно структурировать код таким образом, чтобы защитить нужные данные (см. подраздел 3.2.2) и избежать состояний гонки, присущих используемым интерфейсам (см. подраздел 3.2.3). У мьютексов имеются и собственные проблемы в виде *взаимной блокировки* (см. подраздел 3.2.4) и защиты либо слишком большого, либо слишком малого объема данных (см. подраздел 3.2.8).

Начнем с основ.

3.2.1. Использование мьютексов в C++

Мьютекс в C++ получается при создании экземпляра `std::mutex`, блокируется вызовом компонентной функции `lock()`, а разблокируется вызовом компонентной функции `unlock()`. Но напрямую вызывать компонентные функции не рекомендуется, поскольку при этом нужно помнить о необходимости вызова `unlock()` на всех путях кода из функции, в том числе возникающих из-за выдачи исключений. Вместо этого стандартной библиотекой C++ предоставляется опекун блокировки — шаблон класса `std::lock_guard`, который реализует для мьютекса способ RAII. Он блокиру-

ет предоставляемый мьютекс при работе конструктора и разблокирует при работе деструктора, гарантируя правильную разблокировку в любом случае. В листинге 3.1 показано, как с помощью `std::mutex`, а также `std::lock_guard` защитить список, к которому могут получить доступ сразу несколько потоков. Оба класса объявлены в заголовке `<mutex>`.

Листинг 3.1. Защита списка с помощью мьютекса

```
#include <list>
#include <mutex>
#include <algorithm>
std::list<int> some_list;                                    ❶
std::mutex some_mutex;                                       ❷
void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);           ❸
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex);           ❹
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

В листинге 3.1 есть одна глобальная переменная ❶, она защищена соответствующим глобальным экземпляром `std::mutex` ❷. Использование `std::lock_guard<std::mutex>` в `add_to_list()` ❸ и еще раз в `list_contains()` ❹ означает, что доступ к списку из этих функций является взаимоисключающим: `list_contains()` никогда не увидит список на промежуточном этапе его изменения функцией `add_to_list()`.

В C++17 появилась новая возможность, называемая *выводом аргументов шаблона класса* (class template argument deduction), которая означает, что для простых шаблонов классов, таких как `std::lock_guard`, список аргументов шаблона зачастую можно не указывать. Строки кода ❸ и ❹ в компиляторе C++17 можно сократить до следующего вида:

```
std::lock_guard guard(some_mutex);
```

В подразделе 3.2.4 будет показано, что в C++17 также введена улучшенная версия опекуна блокировки под названием `std::scoped_lock`, поэтому в среде C++17 этот код можно записать следующим образом:

```
std::scoped_lock guard(some_mutex);
```

Для того, чтобы код был понятнее, и для совместимости с прежними компиляторами в других фрагментах кода по-прежнему будет использоваться шаблон класса `std::lock_guard` и указываться его аргументы.

Иногда в таком использовании глобальных переменных есть определенный смысл, однако в большинстве случаев мьютекс и защищенные данные помещаются в один класс, а не в глобальные переменные. Это соответствует стандартным

правилам объектно-ориентированного проектирования: помещение их в один класс служит признаком связанности друг с другом, позволяя инкапсулировать функциональность и обеспечить защиту. В данном случае `add_to_list` и `list_contains` станут компонентными функциями класса, а мьютекс и защищенные данные — закрытыми компонентами класса, что значительно упростит определение того, какой код имеет доступ к данным и, следовательно, какой код должен заблокировать мьютекс. Если все компонентные функции класса блокируют мьютекс перед доступом к любым другим компонентным данным и разблокируют его по завершении доступа, данные будут надежно защищены от любого обращающегося к ним кода.

Самый проницательный читатель мог бы заметить, что это не совсем так: если одна из компонентных функций возвращает указатель или ссылку на защищенные данные, то обстоятельства блокировки мьютекса всеми компонентными функциями строго по очереди уже не будут иметь значения, потому что тем самым в защите будет проделана большая дыра. *Теперь обратиться к защищенным данным и, возможно, их изменить, не блокируя мьютекс, сможет любой код, имеющий доступ к этому указателю или ссылке*. Поэтому защита данных с помощью мьютекса требует тщательной проработки интерфейса, чтобы гарантировать, что мьютекс заблокирован до того, как появится доступ к защищенным данным, и к ним не будет никаких черных ходов.

3.2.2. Структуризация кода для защиты совместно используемых данных

Как видите, защита данных с помощью мьютекса не сводится лишь к добавлению объекта `std::lock_guard` в каждую компонентную функцию: один случайный указатель или ссылка — и вся эта защита рушится. На одном и том же уровне проверка на наличие случайных указателей или ссылок довольно проста: данные находятся в безопасности до тех пор, пока ни одна из компонентных функций не возвращает указатель или ссылку на защищенные данные вызывающему коду посредством возвращаемого значения или выходного параметра. Если же копнуть немного глубже, то все, как всегда, становится гораздо сложнее. Помимо проверки того, что компонентные функции не передают указатели или ссылки вызывающему их коду, важно также убедиться, что они не передают эти указатели или ссылки тем функциям, которые вызываются ими и не контролируются вами. Такая передача не менее опасна: эти функции могут хранить указатель или ссылку в том месте, где их позже можно использовать без защиты, предоставляемой мьютексом. В этом смысле особенно опасны функции, которые предоставляются во время выполнения программы в виде аргументов функций или иными способами (листинг 3.2).

Листинг 3.2. Случайная передача за пределы защиты ссылки на защищаемые 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();
}
```

❶ Передача «защищенных» данных функции, предоставленной пользователем

❷ Передача во вредоносную функцию

❸ Незащищенный доступ к защищенным данным

В этом примере код в `process_data`, казалось бы, не вызывает опасений и выглядит вполне защищенным с помощью `std::lock_guard`, но вызов предоставленной пользователем функции `func` ❶ означает, что `foo` может передать вредоносную функцию `malicious_function` для обхода защиты ❷, а затем вызвать `do_something()` уже без блокировки мьютекса ❸.

По сути, проблема кода состоит в невыполнении им ваших намерений пометить все фрагменты кода, обращающиеся к структуре данных, как *взаимоисключающие*. В данном случае он пропустил код в `foo()`, который вызывает `unprotected->do_something()`. К сожалению, помочь справиться с проблемой такого рода библиотека потоков C++ не в состоянии, так как задача блокировки нужного мьютекса для защиты данных возлагается на программиста. В то же время можно воспользоваться рекомендацией, которая поможет в подобных случаях: *не передавайте указатели и ссылки на защищенные данные за пределы блокировки никаким способом: ни возвращая их из функции, ни сохраняя во внешне видимой памяти и ни передавая в качестве аргументов функциям, предоставленным пользователем.*

Хотя данная ошибка при попытке использования мьютексов для защиты совместно используемых данных распространена наиболее широко, она далеко не единственно возможная. В следующем разделе будет показано, что даже при защите данных мьютексом вероятность возникновения состояния гонки все еще сохраняется.

3.2.3. Обнаружение состояния гонки, присущего интерфейсам

Применение мьютекса или другого механизма для защиты совместно используемых данных не дает полной гарантии защищенности от состояния гонки, нужно еще убедиться в защищенности соответствующих данных. Вернемся к примеру двусвязного списка. Чтобы поток мог безопасно удалить узел, необходимо убедиться в том, что предотвращен конкурентный доступ к трем узлам: удаляемому узлу и узлам по обе стороны от него. Если защищать доступ к указателям каждого узла по отдельности, ситуация будет не лучше, чем при использовании кода, в котором мьютексов нет, поскольку от вероятности состояния гонки избавиться не удастся — нужно защищать не отдельные узлы на отдельных этапах, а всю структуру данных для всей операции удаления. В данном случае самое простое решение — применить один мьютекс, защищающий, как в листинге 3.1, весь список.

Безопасность отдельных операций над списком не решает общей проблемы, состояние гонки может возникнуть даже при использовании простого интерфейса. Рассмотрим структуру данных стека на примере применения адаптера контейнера `std::stack`, показанного в листинге 3.3. Помимо конструкторов и функции `swap()`, есть только пять операций, проводимых над `std::stack`: можно поместить в стек новый элемент, применив функцию `push()`, извлечь элемент из стека, использовав `pop()`, прочитать верхний элемент с помощью `top()`, проверить, не является ли стек пустым, задействовав `empty()`, и прочитать количество элементов стека, применив `size()`. Даже если изменить функцию `top()`, заставив ее возвращать копию, а не ссылку (в соответствии с указаниями из подраздела 3.2.2), и защитить внутренние данные с помощью мьютекса, этот интерфейс все равно не будет застрахован от возникновения гонки. Такая проблема характерна не только для реализаций на основе мьютексов, она свойственна и интерфейсу, поэтому состояния гонки будут возникать и при реализации кода 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&&);
    template <class... Args> void emplace(Args&&... args); <-- Новое в C++14
};
```

Проблема в том, что полагаться на результаты работы функций `empty()` и `size()` нельзя. Хотя на момент вызова они, вероятно, и были достоверными, но после возврата из функции любой другой поток может обратиться к стеку и затолкнуть в него новые элементы (`push()`), либо вытолкнуть существующие (`pop()`), причем до того, как поток, вызывающий `empty()` или `size()`, сможет воспользоваться этой информацией.

В частности, если экземпляр стека *не является совместно используемым*, то с помощью следующего кода вполне безопасно проверить его на пустоту методом `empty()`, а затем вызвать `top()` для доступа к верхнему элементу:

```
stack<int> s;
if(!s.empty())   <-- ❶
{
    int const value=s.top();   <-- ❷
    s.pop();
    do_something(value);       <-- ❸
}
```

Для однопоточного кода такой вариант не только безопасен, но и вполне ожидаем, поскольку вызов `top()` при пустом стеке вызывает неопределенное поведение. При совместно используемом объекте стека *эта последовательность вызовов перестает быть безопасной* из-за возможности вызова `pop()` из другого потока, в результате чего между вызовами `empty()` ❶ и `top()` ❷ удаляется последний элемент. Следовательно, это классическое состояние гонки, не устраняемое внутренним использованием мьютекса для защиты содержимого стека и являющееся следствием применения данного интерфейса.

Так каким же должно быть решение? Данная проблема является следствием конструкции интерфейса, поэтому решение состоит в ее изменении. Остается ответить на вопрос: какими должны быть изменения? Проще всего объявить, что функция `top()` выдаст исключение, если при ее вызове в стеке не будет никаких элементов. Это позволит полностью решить проблему, однако трудоемкость программирования возрастет, поскольку теперь требуется перехватывать исключения, даже если вызов `empty()` вернул значение `false`. Вызов `empty()` становится не необходимой частью конструкции, а средством оптимизации, позволяющим избежать издержек на создание исключения, если стек уже пуст (хотя, если между вызовами `empty()` и `top()` состояние изменяется, исключение все равно будет выдано).

Если присмотреться к предыдущему фрагменту кода, можно заметить вероятность возникновения еще одного состояния гонки между вызовами `top()` ❷ и `pop()` ❸. Рассмотрим два потока, выполняющие предыдущий фрагмент кода и ссылающиеся на один и тот же объект стека `s`. Это вполне типичная ситуация: при использовании потоков для повышения производительности зачастую достаточно иметь несколько потоков, выполняющих один и тот же код для разных данных, и совместно используемый объект стека идеально подходит для разделения работы между ними (хотя чаще всего для этой цели используется очередь — см. примеры в главах 6 и 7). Предположим, что изначально в стеке имеется два элемента, поэтому не нужно беспокоиться о гонке между `empty()` и `top()` в любом потоке и учитывать возможные шаблоны выполнения.

Если стек внутренне защищен мьютексом, то компонентную функцию стека в любой момент времени может запускать только один поток, поэтому вызовы таких функций нужным образом чередуются, но вызовы `do_something()` могут выполняться одновременно. Один из возможных вариантов выполнения кода показан в табл. 3.1.

Таблица 3.1. Возможный порядок операций над стеком из двух потоков

Поток А	Поток Б
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);

Если это единственные запущенные потоки, то можно заметить, что между двумя вызовами `top()` никаких изменений в стек не вносится, поэтому оба потока будут видеть одно и то же значение. При этом между вызовами `pop()` нет вызовов `top()`. Следовательно, одно из двух значений в стеке отбрасывается непрочитанным, а другое обрабатывается дважды. И это еще одно состояние гонки, гораздо более коварное, чем неопределенное поведение в случае гонки между `empty()` и `top()`. В этом коде вроде бы вообще нет ничего явно неправильного, а последствия ошибки, вероятно, проявятся намного позже причины ее возникновения, хотя вполне очевидно, что они зависят от действий, выполняемых функцией `do_something()`. Здесь требуется более радикальное изменение интерфейса, объединяющего вызовы `top()` и `pop()` под защитой мьютекса. Том Каргилл (Tom Cargill)[1] отметил, что такой объединенный вызов способен вызвать проблемы, если конструктор копирования для объектов в стеке может выдать исключение. А Херб Саттер (Herb Sutter)[2] всесторонне рассмотрел эту проблему с точки зрения безопасности исключений, но вероятность возникновения состояния гонки привносит в нее новые аспекты.

Чтобы разъяснить суть проблемы, рассмотрим стек `<vector <int>>`. Теперь вектор является контейнером динамически изменяемого размера, поэтому при копировании вектора библиотека должна распределить из кучи достаточный объем памяти, позволяющий скопировать содержимое. При высокой загруженности системы или существенном дефиците ресурсов распределение памяти может дать сбой, а конструктор копирования вектора — выдать исключение `std::bad_alloc`. Такое развитие событий вполне вероятно, когда вектор содержит слишком много

---

1 *Cargill T.* Exception Handling: A False Sense of Security // C++ Report 6, 1994. — № 9. Статья также доступна по адресу http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

2 *Sutter H.* Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. — Addison Wesley Professional, 1999.

элементов. Если в определении функции pop() были предусмотрены возврат извлеченного значения и удаление его из стека, то мы получаем предпосылку возникновения проблемы: извлекаемое значение возвращается вызывающему коду только после изменения стека, но копирование данных для их возврата вызывающему коду может выдать исключение. Если это произойдет, извлеченные данные будут потеряны: удаление из стека уже произошло, а при копировании случился сбой! Разработчики интерфейса std::stack любезно разбивают операцию на две части: получение верхнего элемента (top()), а затем удаление его из стека (pop()), чтобы при невозможности безопасного копирования данных элемент оставался в стеке. Если проблема заключалась в дефиците памяти кучи, то, вероятно, приложение в состоянии освободить часть памяти и повторить попытку.

К сожалению, именно такого разбиения мы пытаемся избежать для устранения состояния гонки! К счастью, есть альтернативные варианты, но они сопряжены с определенными затратами.

Вариант 1: передача в ссылке

Первый вариант заключается в передаче функции pop() в виде аргумента ссылки на переменную, в которую желательно получить извлекаемое из стека значение:

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

Этот вариант подходит для многих случаев, но у него есть явный недостаток: он требует от вызывающего кода создания экземпляра типа значения стека еще до вызова, чтобы его можно было передать в качестве получателя. Для некоторых типов это неприемлемо из-за затратности создания экземпляра с точки зрения времени или ресурсов. Другим типам такая возможность предоставляется не всегда, поскольку конструкторам требуются параметры, которые могут быть доступны в данном месте кода. И наконец, нужно, чтобы сохраненный тип допускал присваивание значений. Это очень важное ограничение: многие пользовательские типы не поддерживают присваивание, хотя могут поддерживать конструкции перемещения или даже копирования и допускают возвращение результата в виде значения.

Вариант 2: востребование конструктора копирования без выдачи исключений или конструктора перемещения

Остается решить проблему безопасности при выдаче исключений в функции pop(), возвращающей значение, если возвращение результата в виде значения может выдать исключение. У многих типов имеются конструкторы копирования, не выдающие исключений, а с новой поддержкой ссылок на *r*-значения в стандарте C++ (см. раздел A.1) у многих других типов будут конструкторы перемещения, которые не выдают исключений, даже если они могут выдаваться их конструктором копирования. Один из приемлемых вариантов заключается в ограничении использования вашего потокобезопасного стека теми типами, которые могут безопасно возвращать результаты в виде значений без выдачи исключения.

При всей своей безопасности это решение далеко от идеала. Несмотря на возможность обнаружения в ходе компиляции конструктора копирования или перемещения, не выдающего исключений, при использовании признаков типа `std::is_nothrow_copy_constructible` и `std::is_nothrow_move_constructible` данное решение накладывает слишком большие ограничения. Пользовательских типов, имеющих конструкторы копирования, выдающие исключения, и не имеющих конструкторов перемещений, гораздо больше, чем типов с конструкторами копирования и/или перемещения, не выдающими исключений (хотя по мере того, как все привыкнут к поддержке в C++11 ссылок на `r`-значения, ситуация может измениться). Будет жаль, если такие типы окажется невозможно сохранить в потокобезопасном стеке.

Вариант 3: возвращение указателя на извлекаемый элемент

Третий вариант заключается не в возвращении элемента в виде значения, а в возвращении указателя на извлекаемый элемент. Его преимуществом является свободное копирование указателей без выдачи исключений, что позволяет избежать проблемы исключений, рассмотренных Каргиллом. Недостаток заключается в том, что для возвращения указателя требуются средства управления памятью, распределенной объекту, а для простых типов, например целых чисел, издержки такого управления памятью могут превышать затраты на возвращение типа в виде значения. Для любого интерфейса, использующего данный вариант, при выборе типа для указателя вполне подойдет `std::shared_ptr`, он не только позволит избежать утечек памяти, поскольку объект уничтожается после уничтожения последнего указателя, но и даст возможность библиотеке полностью контролировать схему распределения памяти без использования `new` и `delete`. Это может сыграть важную роль в оптимизации: требование, чтобы каждый объект в стеке распределялся отдельно с помощью `new`, приводило бы к более весомым издержкам по сравнению с исходной непотокобезопасной версией.

Вариант 4: сочетание первого и второго или третьего вариантов

Гибкость исключать не следует, особенно в обобщенном коде. При выборе варианта 2 или 3 сравнительно несложно реализовать и вариант 1, что даст пользователям кода возможность выбрать наиболее подходящий для них вариант с небольшими дополнительными затратами.

Пример определения потокобезопасного стека

В листинге 3.4 показано определение класса для стека, не подверженного состоянию гонки в интерфейсе с реализацией первого и третьего вариантов: в нем есть две перезагружаемые версии функции `pop()`, одна из которых принимает ссылку на место хранения значения, а другая возвращает `std::shared_ptr<>`. Используется довольно простой интерфейс, состоящий всего из двух функций, `push()` и `pop()`.

Листинг 3.4. Схематическое определение класса потокобезопасного стека

```
#include <exception>
#include <memory>
struct empty_stack: std::exception
{
    const char* what() const noexcept;
};
template<typename T>
class threadsafe_stack
{
public:
    threadsafe_stack();
    threadsafe_stack(const threadsafe_stack&);
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};
```

Для std::shared_ptr<>

Оператор присваивания ❶ удален

Упрощение интерфейса позволило добиться максимальной безопасности, ограничены даже операции целиком над всем стеком. Сам стек нельзя присвоить, поскольку удален оператор присваивания ❶ (см. раздел А.2) и отсутствует функция `swap()`. Но его можно скопировать, так как элементы стека разрешено копировать. Если стек пуст, функции `pop()` выдают исключение `empty_stack`, поэтому все должно работать, даже если стек был изменен после вызова функции `empty()`. Как упоминалось в описании варианта 3, применение `std::shared_ptr` позволяет стеку обеспечить распределение памяти и обойтись без лишних обращений к `new` и `delete`. Из пяти операций со стеком осталось только три: `push()`, `pop()` и `empty()`. И даже `empty()` лишняя. Такое упрощение интерфейса позволяет обеспечить более жесткий контроль над использованием данных: теперь можно гарантировать, что мьютекс блокируется для всей совокупности операций. В листинге 3.5 показана простая реализация, служащая оболочкой `std::stack<>`.

Листинг 3.5. Дополненное 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;
    }
    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));
    }
    std::shared_ptr<T> pop()
    {
        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();
        return res;
    }
    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();
    }
};
```

❶ Копирование выполняется в теле конструктора

Проверка стека на пустоту перед попыткой извлечения элемента

Размещение возвращаемого значения перед изменением стека

Эта реализация стека *допускает копирование* — конструктор копирования блокирует мьютекс в исходном объекте, а затем копирует внутренний стек. Чтобы при этом обеспечить удержание мьютекса, копирование выполняется в теле конструктора ❶, а не в списке инициализаторов компонентов.

При рассмотрении функций `top()` и `pop()` становится понятно, что проблемное состояние гонки в интерфейсах возникает из-за блокировки при слишком малой глубине детализации, так как защита не охватывает всю нужную операцию. Проблемы с мьютексами возможны также из-за блокировки при слишком большой глубине детализации: чрезвычайные обстоятельства возникают в результате применения одного глобального мьютекса, защищающего все совместно используемые данные. В системе со значительным объемом совместно используемых данных это обстоятельство способно свести на нет любые преимущества повышения производительности, связанные с конкурентностью, поскольку потоки вынуждены запускаться поодиночке даже при обращении к разным битам данных. Первые версии ядра Linux, предназначенные для работы в многопроцессорных системах, использовали единую

глобальную блокировку ядра. При этом у двухпроцессорной системы производительность обычно оказывалась ниже, чем у двух однопроцессорных систем, а производительность четырехпроцессорной системы была далеко не такой, как у четырех однопроцессорных систем. Вокруг такого ядра велось множество споров, поскольку потоки, запущенные на дополнительных процессорах, не могли выполнять полезную работу. Более поздние версии ядра Linux перешли на схему блокировки с повышенным уровнем детализации, поэтому производительность четырехпроцессорной системы существенно приблизилась к идеальному четырехкратному показателю однопроцессорной системы из-за гораздо меньшего количества конфликтов.

Одна из проблем с детально проработанными схемами блокировки состоит в том, что иногда, чтобы защитить все данные в операции, нужно заблокировать сразу несколько мьютексов. Ранее уже упоминалось, что в некоторых случаях правильное решение заключается в увеличении степени детализации данных, охватываемых мьютексами, так что необходимо заблокировать только один мьютекс. Но такой подход нежелателен, например, когда мьютексы защищают отдельные экземпляры класса. В этом случае блокировка на следующем, более высоком уровне будет означать либо возложение решения о блокировке на пользователя, либо использование единого мьютекса, защищающего все экземпляры этого класса, но ни то ни другое нельзя признать удачным решением.

Если в итоге для той или иной операции придется заблокировать два или более мьютекса, может возникнуть еще одна потенциальная закулисная проблема — *взаимная блокировка*. Она вызвана обстоятельствами, практически противоположными состоянию гонки: потоки не состязаются за первенство, а каждый из них ждет завершения выполнения операции другим, поэтому никакой подвижки не наблюдается ни в одном из потоков.

3.2.4. Взаимная блокировка: проблема и решение

Представьте себе игрушку, к примеру барабан с палочками. Играть с ним можно только при наличии обеих частей, из которых он состоит. А теперь представьте двух малышей, желающих с ним поиграть. Если у одного из них будут и барабан, и палочки, он сможет весело играть, пока не надоест. Если другому тоже захочется поиграть, ему, как ни досадно, придется подождать. Допустим, барабан и палочки валяются по отдельности в коробке с игрушками, а обоим малышам вдруг захотелось поиграть на барабане и они стали рыться в ней. Один нашел барабан, а другой — палочки. Возникла тупиковая ситуация: пока кто-нибудь не уступит и не даст поиграть другому, каждый останется при своем, требуя отдать ему недостающее, при этом никто не сможет играть на барабане.

Теперь представьте, что спорят не малыши из-за игрушек, а потоки из-за блокировок мьютексов: чтобы выполнить некую операцию, каждая пара потоков нуждается в блокировке каждой пары мьютексов, у каждого потока имеется один заблокированный мьютекс и он ожидает разблокировки другого мьютекса. Продолжить выполнение не может ни один из потоков, поскольку каждый ждет, когда другой разблокирует свой мьютекс. Такой сценарий называется *взаимной блокировкой*

и представляет собой самую серьезную проблему при необходимости заблокировать для выполнения одной операции два мьютекса и более.

Общий совет по обходу взаимной блокировки заключается в постоянной блокировке двух мьютексов в одном и том же порядке: если всегда блокировать мьютекс А перед блокировкой мьютекса Б, то взаимной блокировки никогда не произойдет. Иногда это условие выполнить несложно, поскольку мьютексы служат разным целям, но кое-когда все гораздо сложнее, например, когда каждый из мьютексов защищает отдельный экземпляр одного и того же класса. Рассмотрим, к примеру, операцию обмена данными между двумя экземплярами одного и того же класса. Чтобы обеспечить корректный обмен данными и при этом избежать влияния изменений, вносимых в режиме конкурентности, следует заблокировать мьютексы на обоих экземплярах. Но если выбрать определенный порядок, например сначала блокировать мьютекс для экземпляра, переданного в качестве первого параметра, а затем мьютекс для экземпляра, переданного в качестве второго параметра, то можно получить обратный эффект: стоит всего лишь двум потокам попытаться выполнить обмен данными между теми же двумя экземплярами с переставленными местами параметрами, и вы получите взаимную блокировку!

К счастью, в стандартной библиотеке C++ есть средство от этого в виде `std::lock` — функции, способной одновременно заблокировать два и более мьютекса, не рискуя вызвать взаимную блокировку. Пример из листинга 3.6 показывает, как применить эту функцию для простой операции обмена.

Листинг 3.6. Использование функций std::lock() и std::lock_guard в 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);                                          ❶
        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);       ❷
        std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);       ❸
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

Сначала аргументы проверяются на принадлежность к разным экземплярам, поскольку попытки заблокировать уже заблокированный `std::mutex` приводят к неопределенному поведению. (Мьютекс, допускающий несколько блокировок одним потоком, предоставляется в форме `std::recursive_mutex`. Более подробно он рассматривается в подразделе 3.3.3.) Затем вызов `std::lock()` ❶ блокирует два

мьютекса и создаются два экземпляра `std::lock_guard` ❷ и ❸, по одному для каждого мьютекса. В дополнение к мьютексу предоставляется параметр `std::adopt_lock`, чтобы показать объектам `std::lock_guard`, что мьютексы уже заблокированы. Объекты должны овладеть существующей блокировкой мьютекса, а не пытаться заблокировать его в конструкторе.

Тем самым обеспечивается корректная разблокировка мьютекса при выходе из функции в общем случае, когда защищенная операция может выдать исключение, а также разрешается простое возвращение управления. Следует также отметить, что блокировка либо `lhs.m`, либо `rhs.m` внутри вызова `std::lock` может привести к выдаче исключения, в таком случае исключение распространяется из `std::lock`. Если функцией `std::lock` успешно заблокирован один мьютекс, а исключение выдано при попытке заблокировать другой, первый мьютекс разблокируется автоматически: в отношении блокировки предоставленных мьютексов функция `std::lock` обеспечивает семантику «все или ничего».

В C++17 предоставляется дополнительная поддержка этого сценария в виде нового RAII-шаблона `std::scoped_lock<>`. Он практически эквивалентен `std::lock_guard<>`, за исключением того, что является *вариационным шаблоном*, принимающим в качестве параметров шаблона список типов мьютексов, а в качестве аргументов конструктора — список мьютексов. Предоставленные конструктору мьютексы блокируются с использованием такого же алгоритма, как и в `std::lock`, и, когда конструктор завершает работу, они оказываются заблокированными, а затем разблокируются в деструкторе. Операцию `swap()` из листинга 3.6 можно переписать следующим образом:

```
void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::scoped_lock guard(lhs.m,rhs.m); ◄—❶
        swap(lhs.some_detail,rhs.some_detail);
    }
```

В этом примере используется еще одна возможность, добавленная в C++17, — автоматическое выведение параметров шаблона класса. При наличии компилятора C++17, который, скорее всего, будет применяться при использовании `std::scoped_lock`, механизм подразумеваемого выведения параметра шаблона класса, имеющийся в C++17, выберет подходящие типы мьютексов из типов объектов, переданных конструктору в объекте ❶. Эта строка кода эквивалентна полномасштабной версии:

```
std::scoped_lock<std::mutex,std::mutex> guard(lhs.m,rhs.m);
```

Наличие шаблона `std::scoped_lock` означает, что в большинстве случаев там, где до выхода C++17 приходилось использовать `std::lock`, теперь можно будет задействовать `std::scoped_lock`, сократив поле возможностей допущения ошибок, что можно только приветствовать!

Применение `std::lock` и `std::scoped_lock<>` позволяет избавиться от взаимных блокировок, когда нужно завладеть сразу двумя и более блокировками, однако оно

не поможет, если блокировки захватываются разобщенно. В таком случае, чтобы гарантировать обход взаимных блокировок, разработчикам приходится полагаться на самодисциплину. А это не так-то просто: взаимоблокировки относятся к одной из самых неприятных проблем, с которой приходится сталкиваться в многопоточном коде, их возникновение зачастую невозможно предсказать, поскольку в большинстве случаев все работает нормально. И тем не менее существует ряд относительно простых правил, помогающих создавать код, не подверженный взаимным блокировкам.

3.2.5. Дополнительные рекомендации по обходу взаимных блокировок

Чаще всего причинами взаимных блокировок становятся именно блокировки, но это не единственное условие их возникновения. Они могут проявиться при конкурентном выполнении двух потоков и отсутствии блокировок, когда каждый из потоков вызывает функцию `join()` в отношении объекта `std::thread` другого потока. В таком случае дальнейшее выполнение кода невозможно ни в одном из потоков, поскольку каждый из них ожидает завершения работы функции в другом потоке, уподобляясь детям, отнимающим друг у друга игрушку. Простой цикл, когда поток будет ожидать выполнения какого-либо действия другим потоком, а тот одновременно ожидать завершения действия первым потоком, может возникнуть где угодно. И дело может не ограничиться двумя потоками: взаимную блокировку способен вызвать и цикл из трех и более потоков. Все рекомендации по обходу взаимных блокировок сводятся к одному: не ждать завершения операции другим потоком, если есть вероятность, что он также ждет завершения операции текущим потоком. Способы определения и устранения вероятности ожидания другим потоком завершения операции текущим потоком существуют в виде отдельных рекомендаций.

Избегайте вложенных блокировок

И первой рекомендацией будет самая простая: не устанавливайте блокировку, если уже есть какая-либо блокировка. Если придерживаться данной рекомендации, то получить взаимную блокировку по причине использования одних лишь блокировок станет невозможно, поскольку каждый поток будет удерживать только одну блокировку. При этом сохранится вероятность взаимной блокировки по другим причинам, например из-за взаимного ожидания потоками завершения операции. Но, вероятно, самыми распространенными причинами взаимных блокировок будут все же блокировки мьютексов. Если нужно получить несколько блокировок, то с помощью `std::lock` их следует заключить в рамки единого действия, чтобы избежать взаимных блокировок.

При удержании блокировки вызова избегайте кода, предоставленного пользователем

Это простое следствие из предыдущей рекомендации. Поскольку код предоставляется пользователем, неизвестно, что он делает: он может заниматься чем угодно, в том числе устанавливать блокировку. Если при удержании блокировки вызвать

пользовательский код, устанавливающий блокировку, окажется нарушена рекомендация, предписывающая избегать вложенных блокировок, и может возникнуть взаимная блокировка. Но иногда подобное неизбежно: если вы создаете общий код, например стек из подраздела 3.2.3, то код, предоставляемый пользователем, используется при каждой операции с типом или типами параметров. В таком случае нужна новая рекомендация.

Устанавливайте блокировки в фиксированном порядке

Если есть настоятельная необходимость установить две и более блокировки, но в рамках единой операции с помощью `std::lock` это невозможно, лучшее, что можно сделать, — установить их в каждом потоке в одном и том же порядке. В подразделе 3.2.4 этот прием уже рассматривался как один из способов, позволяющих избежать взаимной блокировки при блокировке двух мьютексов, — главное, определить единый порядок, распространяющийся на все потоки. В некоторых случаях сделать это довольно легко. Посмотрите, к примеру, на стек из подраздела 3.2.3 — мьютекс является внутренним для каждого экземпляра стека, а вот операции над записями данных, сохраненными в стеке, требуют вызова кода, предоставленного пользователем. Можно, конечно, добавить ограничение, согласно которому ни одна из операций над элементами данных, хранящимися в стеке, не должна выполнять никаких операций над самим стеком. Это становится дополнительной обузой для пользователя стека, но обстоятельства, при которых данные, хранящиеся в контейнере, получают доступ к этому самому контейнеру, возникают крайне редко, и это довольно заметное событие, так что пользователь способен справиться с ним вполне успешно.

Но бывают случаи, когда не все так просто, как при исследовании операции обмена из подраздела 3.2.4. По крайней мере в этом случае мьютексы можно заблокировать одновременно, но такая возможность предоставляется не всегда. Если вернуться к связанному списку из раздела 3.1, можно увидеть, что одним из способов защиты списка является установка мьютекса на каждом узле. Затем, чтобы получить доступ к списку, потоки должны устанавливать блокировку на каждый интересующий их узел. Чтобы поток удалил запись, он должен установить блокировку на три узла: удаляемый и узлы с обеих сторон от него, поскольку все они так или иначе изменяются. Также, проходя по списку, поток должен удерживать блокировку на текущем узле, устанавливая ее на следующем по порядку узле, чтобы гарантировать, что в это же время не будет изменен следующий указатель. После установки блокировки на следующем узле с первого узла ее можно снять, поскольку надобность в ней уже отпала.

Такой эстафетный стиль блокировки позволяет нескольким потокам обращаться к списку при условии, что все они обращаются к разным узлам. Но во избежание взаимной блокировки узлы неизменно должны блокироваться в одном и том же порядке: если два узла пытаются пройти по списку в противоположных направлениях, используя эстафетные блокировки, то в середине списка они могут заблокировать друг друга. Если узлы А и Б — соседи в списке, то поток, проходящий в одном направлении, будет пытаться удержать блокировку на узле А, а также установить блокировку на узле Б. Поток, проходящий в другом направлении, станет удерживать

блокировку узла Б и пытаться установить блокировку на узле А. Возникнет классический сценарий взаимной блокировки (рис. 3.2).

Поток 1	Поток 2
Блокировка мьютекса главной записи	
Чтение указателя головного узла	
Блокировка мьютекса головного узла	
Разблокировка мьютекса главной записи	
	Блокировка мьютекса главной записи
Чтение указателя от головного к следующему узлу	Блокировка мьютекса хвостового узла
Блокировка мьютекса следующего узла	Чтение указателя от хвостового к предыдущему узлу
Чтение указателя от следующего к очередному узлу	Разблокировка мьютекса хвостового узла
…	…
Блокировка мьютекса узла А	Блокировка мьютекса узла В
Чтение указателя от А к следующему узлу (которым является узел Б)	Чтение указателя от узла В к следующему узлу (которым является узел Б)
	Блокировка мьютекса узла Б
Попытка блокировки мьютекса узла Б	Разблокировка мьютекса узла В
	Чтение указателя от узла Б к следующему узлу (которым является узел А)
	Попытка блокировки мьютекса узла А
Взаимная блокировка!	

Рис. 3.2. Взаимная блокировка потоков, проходящих по списку в противоположных направлениях

Если при удалении узла Б, расположенного между узлами А и В, поток устанавливает блокировку на узле Б до установки блокировки на узлах А и В, то возникает вероятность взаимной блокировки с потоком, проходящим по списку. Такой поток сначала попробует заблокировать либо узел А, либо узел В в зависимости от направления прохода, но потом обнаружит, что не может установить блокировку на узле Б, поскольку она удерживается потоком, выполняющим удаление и пытающимся установить блокировки на узлах А и В.

Один из способов обхода такой взаимной блокировки заключается в определении порядка прохода по списку, чтобы поток всегда был вынужден блокировать А до бло-

кировки Б и Б — до блокировки В. Это устранит вероятность взаимной блокировки ценой запрещения прохода списка в обратном направлении. Такое же соглашение должно зачастую устанавливаться и для других структур данных.

Используйте иерархию блокировок

Являясь частным случаем определения порядка блокировок, иерархия блокировок позволяет обеспечить средство проверки соблюдения соглашения в ходе выполнения программы. Замысел заключается в том, чтобы разбить приложение на уровни и идентифицировать все мьютексы, которые могут быть заблокированы на любом заданном уровне. Когда код пытается заблокировать мьютекс, он не может этого сделать, если в нем уже содержится блокировка из нижнего уровня. Это можно проверить в ходе выполнения программы, назначив номера уровней каждому мьютексу и сохранив записи о том, какие мьютексы заблокированы каждым потоком. Этот шаблон получил очень широкое распространение, но его прямая поддержка в стандартной библиотеке C++ не обеспечивается, поэтому нужно создать собственный тип мьютекса `hierarchical_mutex`, код которого приведен в листинге 3.8.

В листинге 3.7 показаны два потока, задействующих иерархический мьютекс.

Листинг 3.7. Использование иерархии блокировок во избежание взаимной блокировки

```
hierarchical_mutex high_level_mutex(10000); ❶
hierarchical_mutex low_level_mutex(5000); ❷
hierarchical_mutex other_mutex(6000); ❸
int do_low_level_stuff();
int low_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(low_level_mutex); ❹
    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); ❻
    high_level_stuff(low_level_func()); ❺
}
void thread_a()   ❼
{
    high_level_func();
}

void do_other_stuff();
void other_stuff()
{
    high_level_func(); ❿
    do_other_stuff();
}
void thread_b() ❽
{
    std::lock_guard<hierarchical_mutex> lk(other_mutex); ❾
    other_stuff();
}
```

В этом коде имеется три экземпляра `hierarchical_mutex`, ❶, ❷ и ❸, созданные с убывающими номерами иерархии уровней. Поскольку механизм определен таким образом, что при удержании блокировки `hierarchical_mutex` можно установить блокировку только `hierarchical_mutex` с меньшим номером уровня иерархии, это ограничивает возможности кода.

Если предположить, что `do_low_level_stuff` не блокирует никакие мьютексы, то `low_level_func` является дном вашей иерархии и блокирует `low_level_mutex` ❹. Функция `high_level_func` вызывает `low_level_func` ❺, удерживая блокировку `high_level_mutex` ❻, но это вполне допустимо, учитывая, что иерархический уровень мьютекса `high_level_mutex` (❶: `10000`) выше, чем мьютекса `low_level_mutex` (❷: `5000`).

Поток `thread_a()` ❼ соблюдает правила, поэтому работает нормально.

А вот поток `thread_b()` ❽ нарушает правила, что дает сбой в ходе выполнения. Прежде всего он блокирует мьютекс `other_mutex` ❾, значение иерархического уровня которого всего `6000` ❸. Это означает, что он должен располагаться где-то в середине иерархии. Когда `other_stuff()` вызывает `high_level_func()` ❿, это нарушает иерархию: `high_level_func()` пытается установить блокировку на мьютекс `high_level_mutex` со значением `10000`, значительно бóльшим, чем текущее значение уровня иерархии `6000`. Поэтому экземпляр класса `hierarchical_mutex` выдаст отчет об ошибке, возможно, путем выдачи исключения или прерывания выполнения программы. Взаимные блокировки между иерархическими мьютексами невозможны, поскольку порядок блокировки обеспечивается самими мьютексами. Это означает невозможность одновременного удержания двух блокировок, если они относятся к одному и тому же иерархическому уровню, и эстафетная схема блокировки требует, чтобы каждый мьютекс в цепочке имел меньшее значение иерархического уровня, чем предыдущий. Однако в некоторых случаях это условие может оказаться неподходящим.

Этот пример показывает и еще одну возможность — применение шаблона `std::lock_guard<>` с типом мьютекса, определенным пользователем. Класс `hierarchical_mutex` не является частью стандарта, но его создание не представляет особой трудности, простая реализация показана в листинге 3.8. Хотя это тип, определенный пользователем, он может использоваться с `std::lock_guard<>`, поскольку в нем реализованы три компонентные функции, необходимые для соответствия концепции мьютекса: `lock()`, `unlock()` и `try_lock()`. Непосредственное применение `try_lock()` еще не было показано, но в нем нет ничего сложного: если блокировка мьютекса удерживается другим потоком, эта функция возвращает `false`, а не ждет, пока вызвавший ее поток сможет установить блокировку на мьютекс. Она также может использоваться внутри `std::lock()` в качестве части алгоритма предотвращения взаимных блокировок.

В реализации `hierarchical_mutex` задействуется локальная по отношению к потоку переменная, предназначенная для хранения текущего значения уровня иерархии. Данное значение доступно всем экземплярам мьютексов, но оно разное для каждого из потоков. Это позволяет коду проверять поведение каждого потока по отдельности, и код для каждого мьютекса может проверять, разрешено текущему потоку заблокировать этот мьютекс или нет.

Листинг 3.8. Простой иерархический мьютекс

```
class hierarchical_mutex
{
    std::mutex internal_mutex;
    unsigned long const hierarchy_value;
    unsigned long previous_hierarchy_value;
    static thread_local unsigned long this_thread_hierarchy_value; ❶
    void check_for_hierarchy_violation()
    {
        if(this_thread_hierarchy_value <= hierarchy_value) ❷
        {
            throw std::logic_error("mutex hierarchy violated");
        }
    }
    void update_hierarchy_value()
    {
        previous_hierarchy_value=this_thread_hierarchy_value; ❸
        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(); ❹
        internal_mutex.lock();
        update_hierarchy_value(); ❺
    }
    void unlock()
    {
        if(this_thread_hierarchy_value!=hierarchy_value) ❾
            throw std::logic_error("mutex hierarchy violated");
        this_thread_hierarchy_value=previous_hierarchy_value; ❻
        internal_mutex.unlock();
    }
    bool try_lock()
    {
        check_for_hierarchy_violation();
        if(!internal_mutex.try_lock()) ❼
            return false;
        update_hierarchy_value();
        return true;
    }
};
thread_local unsigned long
    hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX); ❽
```

Главное здесь заключается в использовании значения `thread_local`, представляющего собой значение уровня иерархии текущего потока `this_thread_hierarchy_value` ❶. Оно инициализируется максимальным значением ❽, поэтому изначально

заблокирован может быть любой мьютекс. Поскольку это значение объявлено как `thread_local`, собственная копия может быть у каждого потока, поэтому состояние переменной в одном потоке совершенно не зависит от состояния переменной, когда чтение происходит из другого потока. Дополнительные сведения о `thread_local` приводятся в разделе А.8.

Итак, при первой блокировке экземпляра `hierarchical_mutex` значением `this_thread_hierarchy_value` является `ULONG_MAX`. По своей природе это значение больше любого другого, поэтому проверка в `check_for_hierarchy_violation()` ❷ проходит успешно. Раз это происходит, `lock()` делегируется внутреннему мьютексу для блокировки ❹. После успешной установки этой блокировки иерархическое значение можно обновить ❺.

Если теперь заблокировать *другой* экземпляр `hierarchical_mutex`, удерживая блокировку на первом экземпляре, в значении `this_thread_hierarchy_value` отразится иерархическое значение первого мьютекса. Чтобы пройти проверку ❷, значение уровня иерархии этого второго мьютекса теперь должно быть меньше значения уровня иерархии уже удерживаемого мьютекса.

Теперь важно сохранить предыдущее значение уровня иерархии для текущего потока, чтобы можно было восстановить его в `unlock()` ❻, в противном случае не получится снова заблокировать мьютекс с более высоким значением уровня иерархии, даже если поток не удерживает никаких блокировок. Поскольку предыдущее значение уровня иерархии сохраняется, только когда удерживается внутренний мьютекс `internal_mutex` ❸, и его восстановление выполняется до разблокировки внутреннего мьютекса ❻, появляется возможность безопасно хранить его в самом объекте `hierarchical_mutex`, так как он надежно защищен блокировкой внутреннего мьютекса. Во избежание иерархической путаницы из-за разблокировки в неподходящем порядке в строке ❾ выдается ошибка, если разблокированный мьютекс не является самым последним заблокированным мьютексом. Возможно применение и других механизмов, но этот самый простой. Функция `try_lock()` работает так же, как и функция `lock()`, за исключением того, что если вызов `try_lock()` в отношении внутреннего мьютекса `internal_mutex` дает сбой ❼, значит, блокировкой владеет другой поток, поэтому обновления значения уровня иерархии не происходит и вместо `true` возвращается `false`.

Поскольку обнаружение относится к проверкам времени выполнения, оно по крайней мере не зависит от времени — не приходится ждать неких редко возникающих условий, вызывающих взаимную блокировку. К тому же процесс разработки, требующий такого разделения приложения и мьютексов, помогает избавиться от многочисленных вероятностей взаимных блокировок еще до того, как будет создан соответствующий код. Возможно, стоит проработать проект в этом направлении, даже если дело еще не дошло для написания проверок времени выполнения.

Выход рекомендаций за пределы блокировок

В начале данного раздела уже упоминалось, что взаимные блокировки возникают не только при использовании блокировок — они возможны при любой конструкции синхронизации, способной привести к циклу ожидания. Следовательно, рекоменда-

ции нужно распространить и на эти случаи. Например, точно так же, как по возможности стоит избегать вложенных блокировок, не следует дожидаться, когда поток станет выполнять операцию, удерживая при этом блокировку, так как этому потоку, чтобы продолжать выполняться, может потребоваться эту блокировку установить. Аналогично этому, намереваясь дожидаться завершения потока, возможно, стоит определить иерархию потоков, чтобы поток ожидал завершения только тех потоков, которые находятся ниже его по иерархии. Для этого есть очень простой способ — в соответствии с описаниями, приведенными в подразделе 3.1.2 и разделе 3.3, обеспечить объединение потоков в той же функции, которая их запустила.

При разработке кода с прицелом на избавление от взаимных блокировок можно рассчитывать на то, что с помощью `std::lock()` и `std::lock_guard` удастся исключить большинство случаев их возникновения при простых блокировках. Но иногда требуется более гибкий подход. Для таких случаев стандартной библиотекой предоставляется шаблон `std::unique_lock`. Подобно `std::lock_guard`, он представляет собой шаблон класса, параметризированный типом мьютекса и обеспечивающий управление блокировками в том же RAII-стиле, что и `std::lock_guard`, но с немного более гибким подходом.

3.2.6. Гибкая блокировка с применением std::unique_lock

Шаблон `std::unique_lock` обеспечивает немного более гибкий подход, по сравнению с `std::lock_guard`, за счет смягчения инвариантов: экземпляр `std::unique_lock` не всегда владеет связанным с ним мьютексом. Начнем с того, что конструктору в качестве второго аргумента можно передавать не только объект `std::adopt_lock`, заставляющий объект блокировки управлять блокировкой мьютекса, но и объект отсрочки блокировки `std::defer_lock`, показывающий, что мьютекс при конструировании должен оставаться разблокированным. Блокировку можно установить позже, вызвав функцию `lock()` для объекта `std::unique_lock` (но *не* мьютекса) или же передав объект `std:: unique_lock` функции `std::lock()`. Код листинга 3.6 можно легко переписать так, как показано в листинге 3.9, воспользовавшись `std::unique_lock` и `std::defer_lock` ❶ вместо `std::lock_guard` и `std::adopt_lock`. Код имеет то же количество строк и является эквивалентным, за исключением небольшой особенности: `std::unique_lock` занимает немного большее пространство и действует несколько медленнее по сравнению с `std::lock_guard`. За гибкость, заключающуюся в разрешении экземпляру `std::unique_lock` не владеть мьютексом, приходится расплачиваться тем, что эта информация должна быть сохранена и обновлена.

Листинг 3.9. Использование std::lock() и std::unique_lock в 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);
    }
};
```

❶ std::defer_lock оставляет мьютексы разблокированными

❷ Здесь мьютексы блокируются

Поскольку экземпляр `std::unique_lock` в листинге 3.9 предоставляет компонентные функции `lock()`, `try_lock()` и `unlock()`, объекты `std::unique_lock` можно передать `std::lock()` ❷. В них происходит перенаправление к компонентным функциям с таким же именем в основном мьютексе, где выполняется работа и обновляется флаг внутри экземпляра `std::unique_lock`, показывающий, принадлежит ли мьютекс в данный момент этому экземпляру. Этот флаг необходим, чтобы гарантировать правильный вызов функции `unlock()` в деструкторе. Если экземпляр *действительно* владеет мьютексом, деструктор *должен* вызвать функцию `unlock()`, в ином случае — *не должен*. Этот флаг можно запросить, вызвав компонентную функцию `owns_lock()`. Если передача владения блокировкой или какие-то другие действия, требующие `std::unique_lock`, не предусматриваются, лучше все же по возможности воспользоваться вариационным шаблоном `std::scoped_lock` из C++17 (см. подраздел 3.2.4).

Как и следует ожидать, этот флаг должен где-нибудь храниться. Поэтому размер объекта `std::unique_lock` обычно превышает размер объекта `std::lock_guard` и, кроме того, использование `std::unique_lock` вместо `std:: lock_guard` немного ухудшает производительность, поскольку флаг нуждается в обновлении или проверке. Если `std::lock_ guard` вполне вас устраивает, рекомендуется отдавать предпочтение именно этому объекту. И тем не менее бывают случаи, когда `std::unique_lock` лучше вписывается в решение текущей задачи в силу необходимости дополнительной гибкости. В качестве одного примера можно привести уже рассмотренную отложенную блокировку, а в качестве другого — необходимость передачи владения блокировкой из одной области видимости в другую.

3.2.7. Передача владения мьютексом между областями видимости

Поскольку экземпляры `std::unique_lock` не обязаны владеть связанными с ними мьютексами, владение мьютексом может передаваться между экземплярами путем *перемещения*. В некоторых случаях, например при возвращении экземпляра из функции, оно происходит автоматически, а в других случаях его необходимо выполнять явным образом вызовом функции `std::move()`. По сути, все зависит от того, является ли источник *l-значением* — реальной переменной или ссылкой на

таковую — или *r-значением* — неким временным объектом. Владение передается автоматически, если источник является *r*-значением, или же должно передаваться явным образом, если он является *l*-значением, во избежание случайной передачи владения за пределы переменной. Класс `std::unique_lock` — это пример *перемещаемого*, но не *копируемого* типа. Дополнительные сведения о семантике приводятся в подразделе А.1.1.

Один из вариантов возможного использования заключается в разрешении функции заблокировать мьютекс, а затем передать владение этой блокировкой вызывающему коду, который впоследствии сможет выполнить дополнительные действия под защитой этой же самой блокировки. Соответствующий пример показан в следующем фрагменте кода, где функция `get_lock()` блокирует мьютекс, а затем подготавливает данные перед тем, как вернуть блокировку вызывающему коду:

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;     ◄——❶
}
void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock()); ◄——❷
    do_something();
}
```

Поскольку `lk` — автоматическая переменная, объявленная внутри функции, ее можно возвратить напрямую ❶, без вызова функции `std:move()`. О вызове конструктора перемещения позаботится компилятор. Затем функция `process_data()` сможет передать владение непосредственно в собственный экземпляр `std::unique_lock` ❷, а вызов функции `do_something()` может полагаться на правильную подготовку данных без их изменения в промежутке времени со стороны другого потока.

Обычно такой шаблон следует применять, когда блокируемый мьютекс зависит от текущего состояния программы или от аргумента, переданного в функцию, возвращающую объект `std::unique_lock`. Например, так делается, когда блокировка не возвращается напрямую, а является компонентными данными шлюзового класса, используемого для обеспечения корректного доступа к неким заблокированным защищенным данным. В таком случае доступ к данным осуществляется через этот шлюзовой класс: когда нужно получить доступ к данным, приобретается экземпляр шлюзового класса (вызовом функции наподобие `get_lock()`, как в предыдущем примере), который получает блокировку. Затем к данным можно будет обращаться через компонентные функции шлюзового объекта. Когда надобность в шлюзовом объекте отпадает, его уничтожают, в результате чего снимается блокировка и доступ к защищенным данным разрешается *другим* потокам. Такой шлюзовой объект вполне можно сделать перемещаемым, для того чтобы его можно было возвращать из функции, и в таком случае компонентные данные объекта блокировки также должны быть перемещаемыми.

Гибкость, присущая классу `std::unique_lock`, позволяет экземплярам снимать их блокировки до того, как они будут удалены. Сделать это можно с помощью компонентной функции `unlock()` точно так же, как это происходит с мьютексом. Класс `std::unique_lock` поддерживает точно такой же основной набор компонентных функций для блокировки и разблокировки, как и для мьютекса, поэтому его можно использовать с такими обобщенными функциями, как `std::lock`. Возможность снятия блокировки до удаления экземпляра `std::unique_lock` означает допустимость ее выборочного снятия в конкретной ветви кода, если окажется, что она больше не нужна. Это может положительно повлиять на производительность приложения, поскольку удержание блокировки сверх требуемого времени скорее ухудшит производительность из-за того, что снятия блокировки могут ожидать другие потоки и их работа продлится дольше, чем нужно на самом деле.

3.2.8. Блокировка с соответствующей степенью детализации

О степени детализации блокировки уже говорилось в подразделе 3.2.3: это надуманное понятие, касающееся объема данных, защищенного отдельно взятой блокировкой. Блокировка с более подробной детализацией защищает незначительный объем данных, а с менее подробной — большой. Важно не только выбрать детализацию блокировки, которой окажется достаточно для защиты нужных данных, но и обеспечить, чтобы блокировка удерживалась только для тех операций, которым она требуется. Каждому знакомо чувство раздражения, возникающее, когда ты стоишь с тележкой, полной продуктов, в очереди к кассе, а тот, кто рассчитывается, вдруг вспоминает, что забыл взять клюквенный соус, и заставляет всех ждать, пока он за ним сходит. И мы можем предположить, что чувствует кассир, когда он уже готов принять оплату, а клиент только в этот момент начинает искать в сумке кошелек. Ситуация становится куда менее напряженной, если каждый подвозит к кассе все, что хотел, и готов нужным способом оплатить товар.

То же самое применимо и к потокам: если в ожидании одного и того же ресурса находятся сразу несколько потоков (кассиров на рабочем месте) и любой из потоков удерживает блокировку дольше необходимого, возрастает общее время ожидания (не стоит дожидаться подхода своей очереди к кассе, чтобы приступить к поиску клюквенного соуса). По возможности следует блокировать мьютекс только при получении доступа к совместно используемым данным и постараться выполнять любую другую обработку данных вне блокировки. В частности, удерживая блокировку, не следует осуществлять требующие много времени действия наподобие файлового ввода-вывода. Обычно подобный ввод-вывод в сотни, если не в тысячи раз медленнее чтения или записи сопоставимого объема данных из памяти. Если блокировка не предназначена для защиты доступа к файлу, ввод-вывод при удерживаемой блокировке приведет к неоправданной задержке работы других потоков, так

как они будут заблокированы в ожидании получения блокировки, что, возможно, сведет на нет весь выигрыш в производительности, полученный в результате использования нескольких потоков.

В подобной ситуации выгодно задействовать класс `std::unique_lock`, поскольку он позволяет вызвать функцию `unlock()`, когда коду больше не нужно обращаться к совместно используемым данным, а затем опять вызвать функцию `lock()`, если позже в коде это потребуется:

```
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);
}
```

❶ Заблокированный мьютекс при вызове process() не нужен

❷ Повторная блокировка мьютекса для записи результата

При вызове функции `process()` заблокированный мьютекс не нужен, поэтому перед вызовом ❶ он преднамеренно разблокируется, а потом снова блокируется ❷.

Надеюсь, вполне очевидно, что при наличии одного мьютекса защита всей структуры данных не только повышает вероятность блокировки, но и снижает вероятность сокращения времени ее удержания. Последующие этапы операции потребуют блокировки того же мьютекса, поэтому она должна удерживаться дольше. Такое двойное удорожание блокировки является также двойным стимулом для перехода там, где это возможно, к более детализированной блокировке.

Как показывает данный пример, блокировка с соответствующей детализацией касается не только объема блокируемых данных, но и продолжительности удержания блокировки и того, какие операции выполняются в ходе нее. *В основном блокировка должна удерживаться в течение минимально возможного времени, необходимого для выполнения требуемых операций.* Это также означает, что затратные по времени операции, например получение еще одной блокировки (даже если известно, что это не приведет к взаимной блокировке) или ожидание завершения ввода-вывода, не должны выполняться при удерживании блокировки, если это не вызвано крайней необходимостью.

В листингах 3.6 и 3.9 блокировка двух мьютексов требовала операции обмена данными, которой, что вполне очевидно, нужен был конкурентный доступ к обоим объектам. Предположим, что вместо этого понадобилось провести операцию сравнения с участием простого компонентного элемента данных, представляющего собой обычный целочисленный тип `int`. Изменит ли это сложившуюся ситуацию? Значения типа `int` не создают существенных издержек при копировании, поэтому можно легко скопировать данные каждого сравниваемого объекта, удерживая его блокировку, а затем сравнить скопированные значения. Это означало бы, что по времени блокировка удерживается на каждом мьютексе минимально, а также что одна блокировка не удерживается при удержании другой. В листинге 3.10

показаны класс Y, в котором все именно так и происходит, и примерная реализация оператора сравнения.

Листинг 3.10. Блокировка только одного из двух мьютексов с разносом их блокировки по времени 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);  ←❶
        return some_detail;
    }
public:
    Y(int sd):some_detail(sd){}
    friend bool operator==(Y const& lhs, Y const& rhs)
    {
        if(&lhs==&rhs)
            return true;                          ❷
        int const lhs_value=lhs.get_detail();  ←┘
        int const rhs_value=rhs.get_detail();  ←❸
        return lhs_value==rhs_value;  ←┐
    }                                  ❹
};
```

В данном случае оператор сравнения сначала извлекает сравниваемые значения путем вызова компонентной функции `get_detail()` ❷ и ❸. Эта функция извлекает значение, защищая его блокировкой ❶. Затем оператор сравнения выполняет свою задачу с извлеченными значениями ❹. Но следует обратить внимание на то, что наряду с сокращением времени удержания блокировки за счет удержания в любой момент времени только одной блокировки и исключения возможности взаимной блокировки была немного изменена семантика, в отличие от операции сравнения, где обе блокировки удерживались одновременно. В листинге 3.10 показано, что, если оператор возвращает значение `true`, это означает, что значение `lhs.some_detail` в какой-то момент времени равно значению `rhs.some_detail` в другой момент времени. Между двумя операциями считывания эти два значения можно как угодно изменить, например, между выполнением кода в строках ❷ и ❸ программа вообще могла бы поменять их местами, что сделало бы сравнение бесполезным. Сравнение на равенство может возвратить значение `true`, чтобы показать, что значения были равны, даже если ни в один из моментов они не были равны. Поэтому, внося подобные изменения, важно действовать осмотрительно, чтобы семантика операции не менялась, вызывая тем самым проблемы: *если не удерживать требуемые блокировки на весь период проведения операции, возникает вероятность гонки.*

Порой подходящей степени детализации просто не существует, поскольку не все обращения к структуре данных требуют одинакового уровня защиты. В таком случае вместо обычного класса `std::mutex` стоит воспользоваться каким-нибудь альтернативным механизмом.

3.3. Альтернативные средства защиты совместно используемых данных

Хотя мьютексы и представляют собой наиболее универсальный механизм, они не единственные игроки на поле защиты совместно используемых данных, существуют и альтернативные варианты, более подходящие в конкретных обстоятельствах.

Одним из крайних, но весьма распространенных случаев является ситуация, при которой совместно используемые данные нуждаются в защите от конкурентного доступа только во время их инициализации и после этого уже не требуют никакой явной синхронизации. Это может быть связано с тем, что данные создаются только для чтения, и поэтому отсутствуют всяческие проблемы синхронизации, или же с тем, что необходимая защита реализуется по умолчанию как часть операций с данными. В любом случае блокировка мьютекса, выполняемая исключительно для защиты инициализации, становится ненужной после того, как данные инициализированы, и будет неоправданно ухудшать производительность. Именно поэтому в стандарте C++ имеется механизм, предназначенный исключительно для защиты совместно используемых данных во время инициализации.

3.3.1. Защита совместно используемых данных во время инициализации

Предположим, что есть совместно используемый ресурс, создание которого настолько затратно, что заниматься этим хочется лишь в крайней необходимости: возможно, он открывает подключение к базе данных или распределяет слишком большой объем памяти. Подобная *отложенная* (или *ленивая*) *инициализация* (lazy initialization) довольно часто встречается в однопоточном коде — каждая операция, требующая ресурса, сначала проверяет, не был ли он инициализирован, и, если не был, прежде чем воспользоваться этим ресурсом, инициализирует его:

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

Если совместно используемый ресурс безопасен при получении к нему конкурентного доступа, единственной частью, нуждающейся в защите при преобразовании кода в многопоточный, является инициализация ❶. Но слишком простое преобразование, подобное показанному в листинге 3.11, может привести к ненужному последовательному выстраиванию потоков, использующих ресурс. Причина в том, что каждый поток будет вынужден ожидать разблокировки мьютекса, чтобы проверить, не был ли ресурс уже инициализирован.

Листинг 3.11. Безопасная для потоков отложенная инициализация с использованием мьютекса

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;
void foo()
{
    std::unique_lock<std::mutex> lk(resource_mutex);
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);
    }
    lk.unlock();
    resource_ptr->do_something();
}
```

Здесь все потоки выстраиваются в последовательность

В защите нуждается только инициализация

Этот код настолько распространен, а ненужное выстраивание в последовательность вызывает столько проблем, что многие пытались придумать более подходящий способ решения данной задачи, включая небезызвестный *шаблон блокировки с двойной проверкой*: сначала указатель считывается без получения блокировки (в строке ❶ следующего кода), которая устанавливается, только если он имеет значение `NULL`. После получения блокировки указатель проверяется *еще раз* (в строке ❷, следовательно, в этой части получается *двойная проверка*) на тот случай, если между первой проверкой и получением блокировки данным потоком инициализация была выполнена каким-нибудь другим потоком:

```
void undefined_behaviour_with_double_checked_locking()
{
    if(!resource_ptr)   ←❶
    {
        std::lock_guard<std::mutex> lk(resource_mutex);
        if(!resource_ptr)                          ←❷
        {
            resource_ptr.reset(new some_resource); ←❸
        }
    }
    resource_ptr->do_something(); ←❹
}
```

К сожалению, эта схема снискала себе дурную славу, так как в ней имеется потенциальная вероятность вложенного состояния гонки, поскольку чтение за пределами блокировки ❶ не синхронизировано с записью, выполняемой другим потоком под блокировкой ❸. Тем самым создается состояние гонки, распространяющееся не только на сам указатель, но и на указываемый объект: даже если поток видит указатель, записанный другим потоком, он может не увидеть только что созданный экземпляр `some_resource`, в результате чего вызывается функция `do_something()` ❹, работающая с неверными значениями. Это пример такого типа гонки, который определяется стандартом C++ как *гонка за данными* и указывается как вызывающая *неопределенное поведение*. Поэтому абсолютно ясно, что подобной гонки следует избегать. Более подробно модель памяти, включая все составляющие гонки за данными, рассматривается в главе 5.

Комитет по стандартизации C++ также понял важность подобного развития событий, и поэтому, чтобы справиться с данной ситуацией, стандартная библиотека

C++ предоставляет компоненты `std::once_flag` и `std::call_once`. Вместо блокировки мьютекса и явной проверки указателя каждый поток может безопасно воспользоваться функцией `std::call_once`, зная, что к моменту возвращения управления из этой функции указатель будет инициализирован каким-либо потоком (синхронизированным образом). Необходимые для этого данные синхронизации хранятся в экземпляре `std::once_flag`, и каждый экземпляр `std::once_flag` соответствует другой инициализации. Задействование функции `std::call_once` обычно связано с меньшими издержками по сравнению с явным использованием мьютекса, особенно когда инициализация уже была выполнена. Поэтому, если она соответствует функциональным требованиям, предпочтение следует отдавать именно ей. Далее показана операция из листинга 3.11, переделанная для использования `std::call_once`. В данном случае инициализация выполнена путем вызова функции, но это легко можно сделать и с помощью экземпляра класса с оператором вызова функции. Подобно большинству функций, имеющихся в стандартной библиотеке, принимающих в качестве аргумента функции или предикаты, функция `std::call_once` работает с любыми функцией или объектом, допускающими вызов:

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;
void init_resource()
{
    resource_ptr.reset(new some_resource);
}
void foo()
{
    std::call_once(resource_flag,init_resource);
    resource_ptr->do_something();
}
```

❶

Инициализация вызывается только один раз

В этом примере как `std::once_flag` ❶, так и инициализируемые данные являются объектами в области видимости имен, но, как показано в листинге 3.12, функцию `std::call_once()` легко можно использовать для отложенной инициализации компонентов класса.

Листинг 3.12. Потокобезопасная отложенная инициализация компонентов класса с помощью 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)    ❶
```

```
    {
        std::call_once(connection_init_flag,&X::open_connection,this);
        connection.send_data(data
    }
    data_packet receive_data()
    {
        std::call_once(connection_init_flag,&X::open_connection,this);
        return connection.receive_data();
    }
};
```

❷ ❸

В данном примере инициализация выполняется либо при первом вызове `send_data()` ❶, либо при первом вызове `receive_data()` ❸. Использование компонентной функции `open_connection()` для инициализации данных требует также передачи указателя `this`. Как и в других функциях стандартной библиотеки, принимающих вызываемые объекты, такие как конструкторы для `std::thread` и `std::bind()`, это делается передачей `std::call_once()` дополнительного аргумента ❷.

Стоит отметить, что по аналогии с `std::mutex` экземпляры `std::once_flag` нельзя скопировать или переместить, поэтому, если использовать их также в качестве компонента класса, то при необходимости придется определять соответствующие специальные компонентные функции.

Один из сценариев, предполагающих вероятность состояния гонки при инициализации, связан с применением локальной переменной, объявленной с ключевым словом `static`. Инициализация такой переменной определена так, чтобы она выполнялась при первом прохождении потока управления через ее объявление. Это означает, что несколько потоков, вызывающих функцию, в стремлении первыми выполнить определение могут вызвать состояние гонки. На многих компиляторах, предшествующих C++11, состояние гонки создавало реальные проблемы, поскольку на первенство в попытке инициализации переменной могли претендовать сразу несколько потоков или же они могли пытаться использовать ее после начала инициализации, запущенной в другом потоке еще до ее завершения. В C++11 эта проблема была решена: инициализация определена так, чтобы выполняться только в одном потоке, и никакие другие потоки не будут продолжать выполнение до тех пор, пока эта инициализация не будет завершена. Поэтому состояние гонки возникает только вокруг того, какой из потоков выполнит инициализацию, не вызывая более серьезных проблем. Когда нужна только одна глобальная переменная, этим свойством можно воспользоваться в качестве альтернативы `std::call_once`:

```
class my_class;
my_class& get_my_class_instance()
{
    static my_class instance;
    return instance;
}
```

❶ Гарантия потокобезопасности инициализации

Затем сразу несколько потоков могут безопасно вызвать функцию `get_my_class_instance()` ❶, не опасаясь возникновения гонки при инициализации.

Защита данных только для инициализации является частным случаем более общего сценария — структуры данных с редко проводимыми обновлениями. Прак-

тически всегда эта структура данных предназначается только для чтения и допускает при конкурентности чтение сразу несколькими потоками, но изредка она может нуждаться в обновлениях. Здесь нужен защитный механизм, учитывающий данные обстоятельства.

3.3.2. Защита редко обновляемых структур данных

Рассмотрим таблицу, используемую для хранения кэша DNS-записей, с помощью которых выполняется преобразование доменных имен в соответствующие IP-адреса. Во многих случаях заданная DNS-запись не будет изменяться на протяжении длительного периода времени (DNS-записи остаются неизменными в течение многих лет). Хотя временами, по мере того как пользователи станут обращаться к другим веб-сайтам, в таблице могут появляться новые записи и эти данные могут оставаться практически неизменными в течение всего периода своего существования. Важно периодически проверять правильность кэшированных записей, но обновление попрежнему понадобится только при изменении каких-либо деталей.

Хотя обновления происходят крайне редко, это все же случается, и если эта кэш-память предназначена для доступа сразу нескольких потоков, в ходе обновления понадобится соответственная защита, чтобы гарантировать, что ни один из потоков, читающих кэш, не увидит нарушенную структуру данных.

В отсутствие структуры данных, точно соответствующей тому, как она станет использоваться, и специально разработанной для конкурентных обновлений и чтений (таких как в главах 6 и 7), это обновление требует, чтобы выполняющий его поток имел монопольный доступ к структуре данных до завершения операции. После того как обновление структуры данных будет завершено, она снова станет безопасной для конкурентного доступа со стороны нескольких потоков. Поэтому применение `std::mutex` для защиты структуры данных имеет слишком мрачные перспективы, поскольку при этом исключается возможность реализовать конкурентность при чтении структуры данных в тот период, когда она не подвергается модификации, так что нужен другой вид мьютекса. Этот новый тип мьютекса обычно называют мьютексом *чтения — записи*, поскольку он допускает два различных типа использования: монопольный доступ для одного потока записи или общий и конкурентный доступ для нескольких потоков чтения.

Стандартная библиотека C++17 предоставляет два полностью готовых мьютекса такого вида, `std::shared_mutex` и `std::shared_timed_mutex`. C++14 предоставляет лишь `std::shared_timed_mutex`, а C++11 не предоставляет ни один из них. Если по каким-то причинам вы вынуждены применять компилятор версии, предшествующей C++14, можно воспользоваться основанной на исходном предложении реализацией, предоставленной библиотекой Boost. Разница между `std::shared_mutex` и `std::shared_timed_mutex` состоит в том, что `std::shared_timed_mutex` поддерживает дополнительные операции (рассматриваются в разделе 4.3), поэтому `std::shared_mutex` на некоторых платформах может обеспечить выигрыш в производительности, если эти дополнительные операции не нужны.

В главе 8 будет показано, что использование такого мьютекса не является панацеей от всех бед и производительность зависит от количества задействованных

процессоров и относительной рабочей нагрузки на читающие и обновляющие потоки. Поэтому большое значение приобретает профилирование производительности кода на целевой системе, позволяющее убедиться, что от дополнительной сложности будут какие-то преимущества.

Для синхронизации вместо экземпляра `std::mutex` используется экземпляр `std::shared_mutex`. Для операций обновления вместо соответствующих специализаций `std::mutex` можно применять `std::lock_guard<std::shared_mutex>` и `std::unique_lock<std::shared_mutex>`. Они обеспечивают монопольный доступ, как и при использовании `std::mutex`. В потоках, которым не нужно обновлять структуру данных, для получения *совместного* доступа вместо этого можно воспользоваться `std::shared_lock<std::shared_mutex>`. Этот шаблон класса RAII был добавлен в C++14 и применяется так же, как и `std::unique_lock`, за исключением того, что несколько потоков могут одновременно получить общую блокировку на один и тот же мьютекс `std::shared_mutex`. Единственное ограничение заключается в том, что, если какой-либо имеющий общую блокировку поток попытается получить монопольную блокировку, он будет блокироваться до тех пор, пока все другие потоки не снимут свои блокировки. Аналогично, если какой-либо поток имеет монопольную блокировку, никакой другой поток не может получить общую или монопольную блокировку, пока не снимет свою блокировку первый поток.

В листинге 3.13 показана реализация простого кэша DNS, похожего на один из рассмотренных, с использованием `std::map` для хранения кэшированных данных, защищенных с помощью `std::shared_mutex`.

Листинг 3.13. Защита структуры данных с помощью std::shared_mutex

```
#include <map>
#include <string>
#include <mutex>
#include <shared_mutex>
class dns_entry;
class dns_cache
{
    std::map<std::string,dns_entry> entries;
    mutable std::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string const& domain) const
    {
        std::shared_lock<std::shared_mutex> lk(entry_mutex);    ❶
        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<std::shared_mutex> lk(entry_mutex);     ❷
        entries[domain]=dns_details;
    }
};
```

Имеющаяся в листинге 3.13 функция поиска записи `find_entry()` использует в целях защиты совместного доступа только для чтения экземпляр `std::shared_lock<>` ❶, поэтому сразу несколько потоков могут одновременно и без проблем вызвать `find_entry()`. В то же время в функции обновления или добавления записи `update_or_add_entry()` для предоставления монопольного доступа при обновлении таблицы используется экземпляр `std::lock_guard<>` ❷. В вызове `update_or_add_entry()` не только предотвращается обновление от других потоков, но и блокируются даже те потоки, которые вызывают функцию `find_entry()`.

3.3.3. Рекурсивная блокировка

Попытка потока заблокировать мьютекс, которым он уже владеет, приводит при использовании `std::mutex` к ошибке и *неопределенному поведению*. Но порой бывает нужно, чтобы поток многократно получал один и тот же мьютекс, не разблокируя его предварительно. Для этой цели в стандартной библиотеке C++ предусмотрен класс `std::recursive_mutex`. Он работает так же, как и `std::mutex`, за тем лишь исключением, что на один его экземпляр можно из одного и того же потока получить несколько блокировок. Прежде чем мьютекс сможет быть заблокирован другим потоком, нужно будет снять все ранее установленные блокировки, поэтому, если функция `lock()` вызывается три раза, то три раза должна быть вызвана и функция `unlock()`. При правильном применении `std::lock_guard<std::recursive_mutex>` и `std::unique_lock<std::recursive_mutex>` все это будет сделано за вас автоматически.

В большинстве случаев при возникновении желания воспользоваться рекурсивным мьютексом, скорее всего, требуется внести изменения в саму конструкцию. Рекурсивные мьютексы обычно используются там, где класс предназначен для конкурентного доступа из нескольких потоков, поэтому у него имеется мьютекс, защищающий компонентные данные. Каждая открытая компонентная функция блокирует мьютекс, выполняет свою работу, а затем снимает с мьютекса блокировку. Но иногда требуется, чтобы одна открытая компонентная функция вызывала как часть своей работы другую компонентную функцию. В таком случае вторая компонентная функция также попытается заблокировать мьютекс, что приведет к неопределенному поведению. Решить эту задачу, что называется, в лоб можно, заменив мьютекс рекурсивным мьютексом. Это позволит успешно завершить блокировку мьютекса во второй компонентной функции, и она сможет продолжить работу.

Но применять такой вариант не рекомендуется: можно принять легкомысленные решения и разработать неудачную конструкцию. В частности, пока удерживается блокировка, инварианты класса обычно нарушаются, а это означает, что вторая компонентная функция должна работать даже при вызове с нарушенными инвариантами. Лучше выделить новую закрытую компонентную функцию, вызываемую из обеих компонентных функций, которая не блокирует мьютекс (в предположении, что он уже заблокирован). Затем тщательно продумать условия, при которых можно вызвать новую функцию, и то, в каком состоянии будут при этом находиться данные.

Резюме

В этой главе рассматривались роковые последствия проблем, связанных с состоянием гонки в условиях совместного использования данных несколькими потоками, и способы, позволяющие их избежать с помощью `std::mutex` и тщательно продуманной конструкции интерфейса. Было показано, что мьютексы не являются панацеей от всех бед и имеют свои проблемы в виде взаимной блокировки, хотя стандартная библиотека C++ предоставляет инструмент `std::lock()`, позволяющий ее избежать. Затем разговор шел о дополнительных методах предотвращения взаимных блокировок и мы кратко рассмотрели темы, связанные с передачей владения блокировкой, и проблемы, вызываемые выбором соответствующей степени детализации блокировки. И наконец, ознакомились с такими альтернативными средствами защиты данных, как `std::call_once()` и `std::shared_mutex`, применяемыми для решения задач в особых случаях.

Но пока мы не поднимали тему, касающуюся ожидания получения входных данных от других потоков. Если ваш потокобезопасный стек пуст, он выдает исключение, поэтому, если один поток вынужден дожидаться другого, помещающего значение в стек (в чем, собственно, и заключается основное предназначение потокобезопасного стека), то ему придется снова и снова пытаться извлечь значение, повторяя эту попытку после каждой выдачи исключения. Из-за этого ценное процессорное время впустую расходуется на проверку, а не на полезную работу. Более того, постоянная проверка может помешать прогрессу, не разрешая выполняться другим имеющимся в системе потокам. Нужен некий способ, чтобы поток мог ожидать завершения задачи, выполняемой другим потоком, не расходуя на это время центрального процессора. Глава 4, материал которой построен на основе уже рассмотренных средств защиты разделяемых данных, знакомит с различными механизмами синхронизации операций между потоками в C++. А в главе 6 показано, как все это можно применить для создания более крупных, многократно используемых структур данных.

4 Синхронизация конкурентных операций

В этой главе

- Ожидание события.
- Ожидание единичных событий с использованием фьючерсов.
- Ожидание с ограничением по времени.
- Использование синхронизации операций с целью упрощения кода.

В предыдущей главе рассматривались различные способы защиты данных, совместно используемых потоками. Но порой нужно не только защитить данные, но и синхронизировать действия отдельно взятых потоков. Например, одному потоку, возможно, потребуется дождаться завершения задачи в другом потоке, прежде чем он сможет завершить выполнение своей задачи. В большинстве случаев какому-нибудь потоку следует дожидаться наступления конкретного события или создания определенного условия. Конечно, можно было бы обойтись и периодической проверкой флага Задача завершена или чем-то подобным, сохраненным в совместно используемых данных, но такое решение слишком далеко от идеала. Распространенность сценария, при котором возникает потребность в подобной синхронизации операций между потоками, настолько высока, что в стандартной библиотеке C++ имеются соответствующие средства в форме *условных переменных* и *будущих результатов (фьючерсов)*, позволяющие справиться с данной задачей. Арсенал этих средств был расширен в технической спецификации по конкурентности (Concurrency Technical Specification, TS), где

предоставляются дополнительные операции для фьючерсов, а также новые средства синхронизации в форме *защелок* (latches) и *барьеров* (barriers).

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

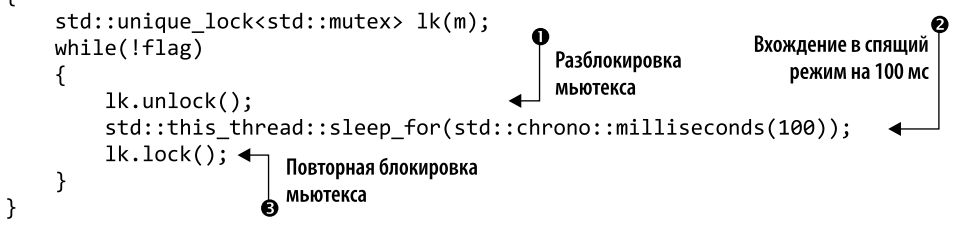
4.1. Ожидание наступления события или создания другого условия

Представьте, что вы едете в ночном поезде. Чтобы гарантированно сойти на нужной станции, придется не спать всю ночь и внимательно отслеживать все остановки. Свою станцию вы не пропустите, но сойдете с поезда уставшим. Но есть и другой способ: заглянуть в расписание, увидеть предполагаемое время прибытия поезда на нужную станцию, поставить будильник на нужное время с небольшим запасом и лечь спать. Этого будет вполне достаточно, и вы не пропустите свою станцию, но, если поезд задержится, пробуждение окажется слишком ранним. Есть также вероятность, что в будильнике сядет батарейка и вы проспите свою станцию. Идеальным решением было бы лечь спать, положившись на то, что кто-нибудь или что-нибудь вас разбудит незадолго до реального прибытия поезда на нужную станцию.

Какое отношение все это имеет к потокам? Если какой-то поток ожидает, пока другой поток завершит выполнение своей задачи, есть несколько вариантов развития событий. Во-первых, первый поток может постоянно проверять состояние флага в совместно используемых данных, защищенных мьютексом, а второй поток будет обязан установить флаг по завершении своей задачи. Это весьма накладно по двум соображениям: постоянно проверяя состояние флага, поток впустую тратит ценное процессорное время, а когда мьютекс заблокирован ожидающим потоком, его нельзя заблокировать никаким другим потоком. Оба обстоятельства наносят вред ожидающему потоку: если он запущен, это ограничивает исполнительные ресурсы, доступные для запуска ожидаемого потока, а на период блокирования мьютекса, защищающего флаг, ожидающим потоком с целью проверки состояния флага ожидаемый поток не может заблокировать мьютекс для установки флага по завершении своей задачи. Это похоже на то, чтобы не спать всю ночь, разговаривая с машинистом поезда: ему приходится вести поезд медленнее, поскольку вы его постоянно отвлекаете, поэтому ваш путь удлиняется. Аналогично этому ожидающий поток потребляет ресурсы, которые могли бы использовать другие потоки, имеющиеся в системе, и ожидание может неоправданно затянуться.

Второй вариант предполагает введение ожидающего потока в спящий режим на короткий промежуток времени между проверками с помощью функции `std::this_thread::sleep_for()` (см. раздел 4.3):

```
bool flag;
std::mutex m;
void wait_for_flag()
```



В данном цикле функция снимает блокировку с мьютекса ❶ перед вхождением в спящий режим ❷ и опять устанавливает на нем блокировку по выходе из спящего режима ❸. Таким образом, другой поток получает шанс заблокировать мьютекс для установки флага.

Это уже гораздо лучше, поскольку поток, находясь в спящем режиме, не тратит процессорное время впустую, но рациональный период пребывания в нем подобрать довольно трудно. Слишком короткий период спячки между проверками — и поток по-прежнему тратит впустую время процессора на слишком частые проверки, слишком длинный период спячки — и поток не выйдет из нее, даже когда задача, завершения которой он ожидал, уже будет выполнена, что приведет к ненужной задержке. Позднее пробуждение напрямую влияет на работу программы довольно редко, но в динамичной игре это может выразиться в пропущенных кадрах, а в приложении реального времени — в превышении отпущенного лимита времени.

Третьим и наиболее предпочтительным вариантом является использование средств из стандартной библиотеки C++, предназначенных для ожидания наступления самого события. Наиболее основательным механизмом ожидания события, инициируемого другим потоком (например, наличие дополнительной работы в упомянутом ранее конвейере), является *условная переменная*. Концептуально она связана с событием или другим *условием*, и один или несколько потоков могут ожидать выполнения условия. Когда поток обнаружит, что условие выполнено, он может *известить* об этом один или несколько потоков, ожидающих условную переменную, чтобы разбудить их и позволить продолжить работу.

4.1.1. Ожидание выполнения условий с применением условных переменных

Стандартная библиотека C++ предоставляет не одну, а *две* реализации условной переменной: `std::condition_variable` и `std::condition_variable_any`. Обе они объявлены в библиотечном заголовке `<condition_variable>`. В обоих случаях для соответствующей синхронизации им нужно работать с мьютексом: первая реализация ограничивается работой только с `std::mutex`, а вторая может работать со всем, что отвечает минимальным критериям схожести с мьютексом, о чем свидетельствует суффикс `_any`. Поскольку `std::condition_variable_any` более универсальна, вероятны более высокие издержки в областях размеров, производительности или ресурсов

операционной системы, поэтому, пока не потребуется дополнительная гибкость, предпочтение следует отдавать реализации `std::condition_variable`.

Как же воспользоваться `std::condition_variable`, чтобы справиться с задачей, приведенной в качестве примера во вступлении? Как позволить потоку, дожидающемуся результатов работы, находиться в спячке, пока идет обработка данных? Один из способов решения этой задачи с помощью условной переменной показан в листинге 4.1.

Листинг 4.1. Ожидание завершения обработки данных с помощью std::condition_variable

```
std::mutex mut;
std::queue<data_chunk> data_queue;  <-- ❶
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);            <-- ❷
        }
        data_cond.notify_one();  <-- ❸
    }
}
void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut); <-- ❹
        data_cond.wait(
            lk,[]{return !data_queue.empty();}); <-- ❺
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();        <-- ❻
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Прежде всего здесь есть очередь ❶, используемая для передачи данных между двумя потоками. Когда данные готовы, поток, подготавливавший данные, блокирует мьютекс, защищающий очередь, с помощью `std::lock_guard` и помещает данные в очередь ❷. Затем он вызывает в отношении экземпляра `std::condition_variable` компонентную функцию `notify_one()` для оповещения ожидающего потока, если таковой имеется ❸. Обратите внимание: код для помещения данных в очередь находится в более узкой области видимости, поэтому условную переменную уведомляют после разблокировки мьютекса, чтобы при немедленном пробуждении ожидающего потока ему не пришлось бы снова быть заблокированным, ожидая разблокировки мьютекса.

По другую сторону препятствия имеется обрабатывающий поток. Первым делом он блокирует мьютекс, но теперь уже с помощью `std::unique_lock`, а не `std::lock_guard` ❹, почему — скоро узнаете. Затем поток вызывает в отношении `std::condition_variable` функцию `wait()`, передавая ей объект-блокировку и лямбда-функцию, которая выражает ожидаемое условие ❺. Лямбда-функции — это новое средство, появившееся в C++11 и позволяющее создавать безымянные функции в качестве части другого выражения. Они идеально подходят для указания предикатов для таких функций из стандартной библиотеки, как `wait()`. В данном случае простая лямбда-функция `[]{return !data_queue.empty();}` проверяет, не является ли очередь `data_queue` пустой, `empty()`, иначе говоря, есть ли в ней какие-либо данные, готовые к обработке. Более подробно лямбда-функции рассматриваются в разделе A.5.

Затем реализация функции `wait()` проверяет условие, вызывая предоставленную лямбда-функцию, и возвращает управление, если условие соблюдено (лямбда-функция вернула `true`). Если условие не соблюдено (лямбда-функция вернула `false`), функция `wait()` снимает блокировку с мьютекса и вводит поток в состояние заблокированности или ожидания. Когда условная переменная уведомлена путем вызова функции `notify_one()` из потока, занимающегося подготовкой данных, поток пробуждается (разблокируется), повторно получает блокировку мьютекса и снова проверяет условие, возвращаясь из `wait()` со все еще заблокированным мьютексом, если условие выполнено. Если условие не выполнено, поток снимает блокировку с мьютекса и возобновляет ожидание. Именно поэтому нужен `std::unique_lock`, а не `std::lock_guard` — ожидающий поток должен разблокировать мьютекс во время ожидания и после него заблокировать его снова, а `std::lock_guard` этой гибкости не обеспечивает. Если мьютекс остается заблокированным на период спячки потока, поток подготовки данных не сможет заблокировать мьютекс, чтобы добавить элемент в очередь, и ожидающий поток никогда не сможет увидеть его условие выполненным.

В листинге 4.1 используется весьма простая лямбда-функция ❺, проверяющая очередь на пустоту, но ей можно передать любую функцию или вызываемый объект. Если у вас уже есть функция для проверки условия (возможно, проверка сложнее простого тестирования, как здесь), то ее можно передать напрямую — нет необходимости заключать ее в саму лямбда-функцию. В ходе вызова `wait()` условная переменная может проверять предоставленное условие любое количество раз, но всегда делает это с заблокированным мьютексом и немедленно вернет управление, если и только если предоставленная для тестирования условия функция вернет значение `true`. Когда ожидающий поток повторно получает блокировку мьютекса и проверяет условие не в ответ на извещение от другого потока, это действие называется *ложным пробуждением* (spurious wake). Поскольку количество и частота любых ложных пробуждений не регламентируются по определению, задействовать для проверки условия функцию с побочными эффектами не рекомендуется. Если вы нарушите эту рекомендацию, будьте готовы к неоднократному возникновению побочных эффектов.

По сути, применение `std::condition_variable::wait`, по сравнению с использованием схемы «занят — ожидайте», является оптимизацией. И действительно,

соответствующий, но все еще далекий от идеала метод реализации представляет собой простой цикл:

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk,Predicate pred){
    while(!pred()){
        lk.unlock();
        lk.lock();
    }
}
```

Ваш код должен быть готов к работе с подобной минимальной реализацией функции `wait()` наравне с реализацией, пробуждающейся только с вызовом `notify_one()` или `notify_all()`.

Гибкость разблокировки `std::unique_lock` применяется не только для вызова `wait()`, она также используется при наличии данных для обработки, но еще до их обработки ❻. Обработка данных потенциально может быть операцией, затратной по времени, и, как было показано в главе 3, удерживание блокировки на мьютексе дольше необходимого имеет пагубные последствия.

Использование очереди для передачи данных между потоками, как в листинге 4.1, — широко распространенный сценарий. При качественной проработке проекта синхронизация может ограничиться самой очередью, что существенно сокращает возможное количество проблем синхронизации и состояний гонки. Исходя из этого, давайте поработаем над извлечением обобщенной потокобезопасной очереди из листинга 4.1.

4.1.2. Создание потокобезопасной очереди с условными переменными

Намереваясь разработать обобщенную очередь, стоит уделить несколько минут обдумыванию операций, которые могут понадобиться с наибольшей вероятностью, как уже делалось в подразделе 3.2.3 в отношении потокобезопасного стека. Поищем в стандартной библиотеке C++ источник вдохновения в виде адаптера контейнера `std::queue<>`, показанного в листинге 4.2.

Листинг 4.2. Интерфейс 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 проигнорировать операции конструирования, присваивания и обмена, останутся три группы операций: для запроса состояния всей очереди (`empty()` и `size()`), запроса элементов очереди (`front()` и `back()`) и внесения изменений в очередь (`push()`, `pop()` и `emplace()`). Они аналогичны тому, что в подразделе 3.2.3 использовалось для стека, поэтому возникают те же вопросы относительно состояний гонки, присущих интерфейсу. Следовательно, требуется объединить `front()` и `pop()` в единый вызов функции во многом аналогично тому, как для стека объединялись функции `top()` и `pop()`. Но в код листинга 4.1 добавлен один нюанс: при использовании очереди для передачи данных между потоками получающему потоку зачастую нужно ожидать данные. Предоставим два варианта функции `pop()`: `try_pop()`, которая будет пытаться извлечь значение из очереди, но всегда тут же возвращать управление (с признаком сбоя), даже если значения для извлечения не было, и `wait_and_pop()`, которая станет ждать, пока не появится значение для извлечения. Если придерживаться сигнатур из примера разработки стека, интерфейс приобретет следующий вид (листинг 4.3).

Листинг 4.3. Интерфейс потокобезопасной очереди threadsafe_queue

```
#include <memory>                          ← Для std::shared_ptr
template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(                    ← Запрещение присваивания
        const threadsafe_queue&) = delete;            в целях упрощения
    void push(T new_value);
    bool try_pop(T& value);          ← ❶
    std::shared_ptr<T> try_pop();
    void wait_and_pop(T& value);     ← ❷
    std::shared_ptr<T> wait_and_pop();
    bool empty() const;
};
```

Как и для стека, здесь в целях упрощения кода исключены конструкторы и удалено присваивание. Как и ранее, предоставлены две версии и `try_pop()`, и `wait_for_pop()`. Первое переопределение `try_pop()` ❶ сохраняет извлеченное значение в ссылочной переменной, что позволяет воспользоваться для состояния возвращаемым значением: функция возвращает `true`, если извлекает значение, и `false`, если его извлечь не удалось (см. раздел А.2). Второе переопределение ❷

этого сделать не может, поскольку возвращает извлеченное значение напрямую. Но если значения для извлечения нет, возвращаемый указатель может быть установлен в NULL.

Но как все это соотносится с листингом 4.1? Оттуда можно взять код для функций push() и wait_and_pop() (листинг 4.4).

Листинг 4.4. Извлечение push() и wait_and_pop() 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;    ←❶
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);   ←
    }                                ❷
}
void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data); ←❸
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Теперь мьютекс и условная переменная содержатся в экземпляре `threadsafe_queue`, поэтому отдельные переменные уже не требуются ❶, а для вызова `push()` не нужна внешняя синхронизация ❷. Кроме того, за ожидание условной переменной отвечает функция `wait_and_pop()` ❸.

Теперь не составляет труда создать еще одно переопределение функции `wait_and_pop()`, а остальные функции можно скопировать почти полностью из примера стека, показанного в листинге 3.5. Финальная реализация очереди показана в листинге 4.5.

Листинг 4.5. Полное определение класса потокобезопасной очереди, использующей условные переменные

```
#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;
    }
```

❶ Мьютекс должен быть изменяемым

```
    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>();
        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();
    }
};
```

Несмотря на то что `empty()` — компонентная функция, объявленная с модификатором, и другой параметр копирующего конструктора является `const`-ссылкой, у других потоков могут быть не-`const`-ссылки на объект и они способны вызывать изменяющие компонентные функции, поэтому потребность в блокировке мьютекса сохраняется. Поскольку блокировка мьютекса — изменяющая операция, объект мьютекса должен быть помечен как изменяемый, `mutable` ❶, и его можно заблокировать в функции `empty()` и в копирующем конструкторе.

Условные переменные пригодятся, когда одного и то же события дожидаются несколько потоков. Если потоки предназначались для разбиения рабочей нагрузки и, таким образом, только один из них должен реагировать на уведомление, можно воспользоваться точно такой же структурой, какая показана в листинге 4.1, просто нужно запустить несколько экземпляров потоков, обрабатывающих данные. Когда будут готовы новые данные (так как только что к очереди `data_queue` добавлен элемент), вызов функции `notify_one()` инициирует один из потоков, выполняющих на тот момент функцию `wait()` для проверки заложенного в нее условия и возвращения управления из `wait()`. Гарантий уведомления того или иного потока дать нельзя, нельзя даже гарантировать наличие потока, ожидающего уведомления, так как все обрабатывающие потоки могут быть все еще заняты обработкой данных.

Другой возможностью является ожидание одного и того же события сразу несколькими потоками, и всем им нужно на него отреагировать. Такая ситуация может сложиться, когда совместно используемые данные были проинициализированы и все обрабатывающие потоки могли воспользоваться одними и теми же данными, но вынуждены были дожидаться их инициализации (хотя для подобной ситуации есть более подходящие механизмы, например `std::call_once`, вариан-

ты можно найти в подразделе 3.3.1) или когда потокам необходимо дождаться обновления совместно используемых данных, например в ходе периодически выполняемой повторной инициализации. В таких случаях поток, подготавливающий данные, может вместо `notify_one()` вызвать для условной переменной компонентную функцию `notify_all()`. Судя по ее названию, она заставит все потоки, выполняющие на данный момент функцию `wait()`, проверить условие, выполнения которого дожидались.

Если ожидающий поток собирается ждать только один раз, то при получении условием значения `true` он не станет больше ждать, основываясь на значении этой условной переменной, и она уже не будет наилучшим вариантом механизма синхронизации. Это особенно справедливо, если условие, соблюдения которого дожидался поток, выражается в доступности конкретного фрагмента данных. В этом сценарии более подходящим вариантом может стать использование фьючерсов.

4.2. Ожидание единичных событий с помощью фьючерсов

Предположим, что в отпуске вы собираетесь полететь на самолете за границу. Прибыв в аэропорт и пройдя все регистрационные процедуры, вы все же вынуждены ожидать, возможно в течение нескольких часов, приглашения на посадку. Конечно, вы можете придумать, как убить время, например почитать книгу, побродить по Интернету или поесть в дорогущем кафе аэропорта, но, по сути, просто ждете сигнала о возможности посадки в самолет. Однако не только он, но и текущий полет случится только один раз, в следующем отпуске вы будете ждать посадки уже на другой рейс.

В стандартной библиотеке C++ эта разновидность единичного события моделируется с помощью средства под названием «фьючерс». Если поток нуждается в ожидании конкретного единичного события, он каким-то образом получает фьючерс, представляющий событие. Затем поток может периодически непродолжительное время ожидать фьючерс, чтобы увидеть, состоялось событие или нет (смотреть на табло вылета), выполняя между проверками какую-нибудь другую задачу (питаясь в дорогущем кафе). В качестве альтернативного варианта он может выполнять другую задачу, пока, если событие не наступило, не сможет продолжить работу, а затем просто ждать, пока фьючерс не будет *готов*. У фьючерса могут быть связанные с ним данные (например, через какой выход произойдет посадка на рейс), но их может и не быть. Как только событие состоится (и фьючерс будет *готов*), фьючерс нельзя будет сбросить к исходному значению.

В стандартной библиотеке C++ имеется два вида фьючерсов, которые реализованы в виде двух шаблонов классов, объявленных в библиотечном заголовке `<future>`: *уникальные фьючерсы* (`std::future<>`) и *совместно владеющие фьючерсы* (`std::shared_future<>`). Они созданы по образцу `std::unique_ptr` и `std::shared_ptr`. Экземпляр `std::future` является уникальным, ссылающимся на связанное с ним событие, а несколько экземпляров `std::shared_future` могут ссылаться на одно и то же

событие. В последнем случае все экземпляры достигнут *готовности* одновременно и все могут обращаться к любым данным, связанным с событием. Связанные данные и являются причиной того, что фьючерсы реализованы в виде шаблонов: точно так же, как и в случае с `std::unique_ptr` и `std::shared_ptr`, параметром шаблона является тип связанных данных. Когда связанных данных нет, нужно использовать шаблонные специализации `std:future<void>` и `std::shared_future<void>`. Хотя фьючерсы используются для обмена данными между потоками, сами объекты фьючерсов не обеспечивают синхронизированных обращений. Если к одному и тому же объекту фьючерса нужно обратиться сразу нескольким потокам, они должны защитить доступ с помощью мьютекса или другого механизма синхронизации в соответствии с приемами, рассмотренными в главе 3. Но в подразделе 4.2.5 будет показано, что каждый из нескольких потоков может обращаться к собственной копии `std::shared_future<>` без какой-либо синхронизации, даже если они ссылаются на один и тот же результат, полученный в асинхронном режиме.

В технической спецификации по конкурентности (Concurrency TS) в пространстве имен `std::experimental` предоставляются расширенные версии этих шаблонов классов: `std::experimental::future<>` и `std::experimental::shared_future<>`. Они ведут себя так же, как и их аналоги в пространстве имен `std`, но при этом имеют дополнительные компонентные функции для предоставления расширений. Важно отметить, что имя `std::experimental` не влияет на качество кода (смею надеяться, что реализация будет такой же по качеству, как и все то, что поставляет ваш разработчик библиотеки), но стоит подчеркнуть, что это нестандартные классы и функции и поэтому они могут не иметь абсолютно таких же синтаксиса и семантики, если когда-нибудь будут окончательно приняты во фьючерсы стандарта C++. Чтобы воспользоваться этими средствами, следует включить в программу заголовок `<experimental/future>`.

Элементарными единичными событиями являются результаты вычислений, запущенных в фоновом режиме. В главе 2 было показано, что `std::thread` не предоставляет простых средств для возвращения значения из подобной задачи, и я обещал, что этот вопрос будет рассмотрен с фьючерсами в главе 4. Теперь самое время выполнить обещание.

4.2.1. Возвращение значений из фоновых задач

Предположим, что имеются какие-то длительные вычисления, которые, как ожидается, дадут со временем полезный результат, значение которого вас пока не интересует. Возможно, вы нашли способ получения ответа на вопрос о сути Жизни, Вселенной и всего остального, позаимствовав пример из книги Дугласа Адамса (Douglas Adams)[1]. Для выполнения вычислений можно запустить новый поток, но это будет означать, что следует позаботиться о передаче результата в основную про-

[1] В книге «Автостопом по Галактике» описывалось создание компьютера Deep Thought для получения ответа на вопрос о сути Жизни, Вселенной и всего остального. Ответом стало число 42.

грамму, поскольку `std::thread` не предоставляет непосредственного механизма для выполнения этой задачи. И здесь весьма кстати воспользоваться функциональным шаблоном `std::async`, также объявленным в заголовке `<future>`.

Шаблон `std::async` применяется для запуска *асинхронной задачи*, результат решения которой пока не нужен. Вместо предоставления объекта ожидания `std::thread` шаблон `std::async` возвращает объект `std::future`, который в конечном итоге будет содержать значение, возвращаемое функцией. Когда это значение понадобится, для фьючерса просто вызывается функция `get()` и поток блокируется до *готовности* фьючерса, а затем возвращает значение. Простой пример показан в листинге 4.6.

Листинг 4.6. Использование std::future для получения значения, возвращаемого задачей, выполняемой в асинхронном режиме

```
#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<<"The answer is "<<the_answer.get()<<std::endl;
}
```

Шаблон `std::async` позволяет передавать функции дополнительные аргументы, добавляя их к вызову точно так же, как это делается при использовании `std::thread`. Если первым аргументом является указатель на компонентную функцию, второй аргумент предоставляет объект, к которому применяется компонентная функция (либо напрямую, либо через указатель, либо заключенным в оболочку `std::ref`), а остальные аргументы передаются компонентной функции в качестве ее простых аргументов. Или же второй и последующие аргументы передаются в качестве аргументов функции или вызываемого объекта, указанного в качестве первого аргумента. Как и в случае с `std::thread`, если аргументы являются *r*-значениями, то *перемещением* оригиналов создаются копии. Это позволяет воспользоваться типами, допускающими только перемещения, как в качестве функционального объекта, так и в качестве аргументов.

Рассмотрим листинг 4.7.

Листинг 4.7. Передача аргументов функции с помощью 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");
```

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

```
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());
```

Вызов p->foo(42,"hello"), где p — это &x

Вызов tmpy(3.141), где tmpy — это перемещение, созданное из Y()

Вызов y(2.718)

Вызов baz(x)

Вызов tmp(), где tmp создается из std::move(move_only())

Когда ожидается фьючерс, изначально именно от реализации зависит, станет ли `std::async` запускать новый поток, или же задача будет запущена в синхронном режиме. В большинстве случаев именно это и нужно, но можно все же указать, чем следует воспользоваться, для чего перед функцией для вызова применить дополнительный параметр для `std::async`. Этот параметр относится к типу `std::launch` и может быть либо `td::launch::deferred`, чтобы указать, что вызов функции будет отложен до тех пор, пока во фьючерсе не будет вызвана функция `wait()` или `get()`, либо `std::launch::async`, чтобы обозначить, что функция должна быть запущена в своем потоке, либо `std::launch::deferred | std::launch::async`, чтобы указать, что реализация может сделать собственный выбор. Последний вариант является выбором по умолчанию. Если вызов функции отложен, она может никогда не запуститься, 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));
f7.wait();
```

Запуск в новом потоке

Запуск в wait()или get()

По усмотрению реализации

Вызов отложенной функции

Чуть позже здесь, а также в главе 8 будет показано, что использование `std::async` упрощает разбиение алгоритмов на задачи, которые могут запускаться в режиме конкурентности. Но это не единственный способ связать `std::future` с задачей, сделать это можно также заключением задачи в экземпляр шаблона класса `std::packaged_task<>` или созданием кода для явной установки значений с помощью шаблона класса `std::promise<>`. Шаблон класса `std::packaged_task` является абстракцией более высокого уровня, чем `std::promise`, поэтому начнем с него.

4.2.2. Связывание задачи с фьючерсом

Шаблон класса `std::packaged_task<>` привязывает фьючерс к функции или вызываемому объекту. Когда вызывается объект `std::packaged_task<>`, он вызывает связанную функцию или объект и приводит фьючерс в состояние *готовности* с возвращаемым значением, сохраненным в качестве связанных данных. Этим можно воспользоваться как строительным блоком для пула потоков (см. главу 9) или других схем управления задачами, например для запуска каждой задачи в собственном потоке или последовательного запуска всех задач в специально выделенном потоке, работающем в фоновом режиме. Если крупную операцию возможно разбить на автономные подзадачи, каждую из подзадач можно заключить в экземпляр `std::packaged_task<>`, который затем будет передан диспетчеру задач или пулу потоков. Тем самым удается абстрагироваться от подробностей задач — диспетчер имеет дело только с экземплярами `std::packaged_task<>`, а не с отдельно взятыми функциями.

Параметром шаблона для шаблона класса `std::packaged_task<>` является сигнатура функции, например `void()` для функции, не получающей параметры и не имеющей возвращаемых значений, или `int(std::string&,double*)` для функции, получающей не-`const`-ссылку на `std::string` и указатель на `double` и возвращающей значение типа `int`. При создании экземпляра `std::packaged_task` ему следует передать функцию или вызываемый объект, принимающий указанные параметры, а затем возвращающий тип, который можно преобразовать в указанный тип возвращаемого значения. Точного совпадения типов не требуется, можно сконструировать объект `std::packaged_task<double(double)>` из функции, принимающей значение типа `int` и возвращающей значение типа `float`, поскольку типы способны проходить подразумеваемое преобразование.

Тип возвращаемого значения, указанный в сигнатуре функции, определяет тип объекта `std::future<>`, возвращаемого из компонентной функции `get_future()`, а заданный в сигнатуре список аргументов используется для определения сигнатуры оператора вызова в классе упакованной задачи. Например, в листинге 4.8 приведено частичное определение класса для `std::packaged _task <std::string(std::vector<char>*,int)>`.

Листинг 4.8. Частичное определение класса для 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);
};
```

Объект `std::packaged_task` является вызываемым, значит, его можно обернуть объектом `std::function`, передать `std::thread` в качестве функции потока, передать другой функции, требующей вызываемого объекта, или даже вызвать напрямую.

Когда `std::packaged_task` вызывается в качестве функционального объекта, аргументы, предоставленные оператору вызова функции, передаются содержащейся в этом объекте функции, а возвращаемое значение сохраняется в качестве асинхронного результата в объекте `std::future`, полученном от `get_future()`.

Таким образом, задачу можно заключить в объект `std::packaged_task` и извлечь фьючерс перед передачей объекта `std::packaged_task` в то место, где он надлежащим образом будет вызван. Когда понадобится результат, можно будет дождаться готовности фьючерса. Практическое применение рассмотренного алгоритма показано в следующем примере.

Передача задач между потоками

Многие среды графических пользовательских интерфейсов (GUI) требуют, чтобы обновления GUI выполнялись из строго определенных потоков, поэтому, если какому-то потоку требуется обновить GUI, он должен отправить сообщение потоку, определенному для выполнения таких обновлений. В листинге 4.9 показано, что один из способов решения данной задачи, не требующий специальных сообщений для каких-либо действий, связанных с GUI-интерфейсом, обеспечивается шаблоном `std:packaged_task`.

Листинг 4.9. Запуск кода в GUI-потоке с помощью 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()                                    ❶
{
    while(!gui_shutdown_message_received())          ❷
    {
        get_and_process_gui_message();               ❸
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())                        ❹
                continue;
            task=std::move(tasks.front());           ❺
            tasks.pop_front();
        }
        task();                                      ❻
    }
}
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);  ❼
    std::future<void> res=task.get_future();  ❽
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));  ❾
    return res;  ❿
}
```

Код не отличается особой сложностью: поток GUI ❶ выполняет цикл, пока не будет получено сообщение, предписывающее GUI прекратить работу ❷, многократно проводя опрос на наличие GUI-сообщений на обработку ❸, например, щелчков кнопкой мыши, и на наличие задач в очереди. Если в очереди нет задач ❹, цикл повторяется, в противном случае задача извлекается из очереди ❺, с очереди снимается блокировка, после чего запускается задача ❻. Когда задача будет выполнена, фьючерс, связанный с ней, перейдет в состояние готовности.

Так же просто происходит и помещение задачи в очередь: предоставленная функция создает новую упакованную задачу ❼, вызовом компонентной функции `get_future()` из этой задачи получается фьючерс ❽ и задача помещается в список ❾ перед тем, как фьючерс возвращается вызывающему коду ❿. Затем код, отправивший сообщение потоку GUI, может ожидать фьючерс, если ему требуется узнать о завершении задачи, или же проигнорировать фьючерс, если эта информация не нужна.

В данном примере для задач используется экземпляр класса `std::packaged_task<void()>`, в который заключается функция или другой вызываемый объект, не принимающий параметров и возвращающий значение `void` (если он возвращает что-то иное, это значение игнорируется). Это самая простая из возможных задач, но из ранее увиденного легко сделать вывод, что `std::packaged_task` применяется и в более сложных ситуациях — указав в качестве параметра шаблона другую сигнатуру функции, можно изменить тип возвращаемого значения (и, стало быть, тип данных, хранящихся в состоянии, связанном с фьючерсом), а также типы аргументов в операторе вызова функции. Этот пример легко можно распространить на задачи, предназначенные для запуска в потоке GUI, чтобы получить аргументы и возвратить значение в `std::future`, а не просто воспользоваться им в качестве индикатора завершения задачи.

А как быть с такими задачами, которые нельзя выразить в виде простого вызова функции, или с теми, где результат может приходить из более чем одного места? В таких случаях приходится обращаться к третьему способу создания фьючерса — к использованию `std::promise` для явного задания значения.

4.2.3. Создание промисов std::promise

Когда есть приложение, нуждающееся в обработке большого количества сетевых подключений, оно зачастую пытается обработать каждое подключение в отдельном потоке, так как это позволяет легче воспринимать сетевое подключение и упрощает программирование. При небольшом количестве подключений и, соответственно,

небольшом количестве потоков проблем не возникает. К сожалению, по мере роста числа подключений такой подход утрачивает привлекательность: большое количество потоков потребляет немалый объем ресурсов операционной системы и потенциально становится причиной множества переключений контекста (когда количество потоков превышает доступный объем ресурсов оборудования для конкурентности), что ухудшает производительность. В экстремальных случаях операционная система может исчерпать свои ресурсы по запуску новых потоков до того, как будут растрачены ее объемы для сетевых подключений. Поэтому в приложениях с большим количеством сетевых подключений для их обработки принято обходиться небольшим количеством потоков (возможно, только одним потоком), каждый из которых имеет дело с несколькими подключениями.

Рассмотрим один из таких потоков, обрабатывающих подключения. Пакеты данных будут поступать из разных подключений и обрабатываться, по сути, в произвольном порядке, и точно так же в произвольном порядке пакеты данных будут помещаться в очередь на отправку. Во многих случаях другим частям приложения придется ждать либо успешной отправки данных, либо успешного получения новых пакетов данных через конкретные сетевые подключения.

Шаблон `std::promise<T>` позволяет устанавливать значение (типа `T`), которое позже можно прочитать через связанный объект `std::future<T>`. Один из возможных механизмов такого рода будет реализован парой `std::promise` — `std::future`, ожидающий поток может заблокироваться на фьючерсе, а поток, предоставляющий данные, — воспользоваться другой половиной пары, промисом (promise, иногда называют обещанием), для установки связанного значения и приведения фьючерса в состояние готовности.

Получить объект фьючерса `std::future`, связанный с заданным промисом `std::promise`, можно вызовом компонентной функции `get_future()`, как это было с `std::packaged_task`. Когда значение промиса установлено (с помощью компонентной функции `set_value()`), фьючерс приводится в состояние готовности и может использоваться для извлечения сохраненного значения. Если объект `std::promise` уничтожить без установки значения, вместо него будет сохранено исключение. Передача исключений между потоками рассматривается в подразделе 4.2.4.

В листинге 4.10 показан пример кода для потока, обрабатывающего подключения в соответствии в приведенным ранее описанием. В данном примере пара `std::promise<bool>` — `std::future<bool>` используется для идентификации успешной передачи блока исходящих данных: значение, связанное с фьючерсом, представляет собой простой флаг успеха/сбоя. Для входящих пакетов данными, связанными с фьючерсом, становится полезная нагрузка пакета.

Листинг 4.10. Обработка с помощью промисов сразу нескольких подключений в одном потоке

```
#include <future>
void process_connections(connection_set& connections)
{
    while(!done(connections))  ←❶
    {                                  ❷
        for(connection_iterator        ←
```

```
            connection=connections.begin(),end=connections.end();
        connection!=end;
        ++connection)
    {
        if(connection->has_incoming_data()) ◄──❸
        {
            data_packet data=connection->incoming();
            std::promise<payload_type>& p=
                connection->get_promise(data.id); ◄──❹
            p.set_value(data.payload);
        }
        if(connection->has_outgoing_data()) ◄──❺
        {
            outgoing_packet data=
            connection->top_of_outgoing_queue();
            connection->send(data.payload);
            data.promise.set_value(true);     ◄──❻
        }
    }
}
```

Функция `process_connections()` выполняет цикл, пока функция `done()` не возвратит значение `true` ❶. При каждом проходе цикла поочередно проверяется каждое подключения ❷, извлекаются входящие данные, если они есть ❸, или отправляются любые стоящие в очереди исходящие данные ❺. Предполагается, что у пакета входящих данных есть идентификатор, а у него — полезная нагрузка. Идентификатор отображается на объект `std::promise` (возможно, поиском в ассоциативном контейнере) ❹, а в качестве значения устанавливается информационная нагрузка пакета. Исходящий пакет извлекается из очереди исходящих данных и отправляется по подключению в сеть. Когда отправка завершится, промис, связанный с исходящими данными, устанавливается равным `true`, что говорит об успешной передаче ❻. Насколько хорошо все это будет перенесено на сетевой протокол, зависит от самого протокола: такой стиль структуры «промис — фьючерс» для конкретного сценария может и не сработать, хотя данная структура похожа на структуру поддержки асинхронного ввода-вывода некоторых операционных систем.

В приводимом до сих пор коде проигнорированы возможности выдачи исключений. Приятно, конечно, представлять себе мир, в котором все всегда работает идеально, но так не бывает. Иногда возникает дефицит дискового пространства, или еще нет искомых данных, или сбоит сеть, или пропадает доступ к базе данных. Если бы операция выполнялась в том потоке, которому нужен результат, код мог бы просто отрапортовать об ошибке путем выдачи исключения, но было бы неразумно ограничиваться требованием, чтобы все шло как надо, только потому, что возникла потребность использовать `std::packaged_task` или `std::promise`. Поэтому стандартная библиотека C++ задает четкий путь решения проблем с выдачей исключений при таком сценарии развития событий, позволяя сохранять эти исключения в качестве части связанного результата.

4.2.4. Сохранение исключения на будущее

Рассмотрим следующий короткий фрагмент кода. Если в функцию `square_root()` передается значение `-1`, она выдает исключение, которое становится видимым вызывающему коду:

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

Теперь предположим, что вместо вызова `square_root()` из текущего потока:

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

функция вызывается в асинхронном режиме:

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

В идеале хотелось бы получить точно такое же поведение: было бы неплохо, чтобы, подобно тому как в любом случае `y` получает результат вызова функции, код, вызвавший `f.get()`, мог бы видеть исключение, как при однопоточном варианте выполнения.

Именно так все и происходит: если вызов функции, инициированный как часть `std::async`, выдает исключение, оно сохраняется во фьючерсе на месте сохраненного значения, фьючерс приводится в состояние готовности и вызов `get()` повторно выдает сохраненное исключение. (Примечание: в стандарте не указано, является ли повторно выдаваемое исключение исходным объектом исключения или его копией, разные компиляторы и библиотеки делают выбор по своему усмотрению.) То же самое происходит, если функция заключена в `std::packaged_task`: когда при вызове задачи этой функцией выдается исключение, которое сохраняется во фьючерсе на месте результата, готового к выдаче при вызове функции `get()`.

Вполне естественно, что `std::promise` предоставляет те же возможности и в случае явного вызова функции. Если вместо значения требуется сохранить исключение, то вместо `set_value()` вызывается компонентная функция `set_exception()`. Обычно для исключения, выдаваемого в качестве части алгоритма, это делается в блоке `catch`, чтобы заполнить промис этим исключением:

```
extern std::promise<double> some_promise;
try
{
    some_promise.set_value(calculate_value());
}
catch(…)
{
    some_promise.set_exception(std::current_exception());
}
```

Для извлечения выданного исключения здесь используется функция `std::current_exception()`, чтобы сохранить новое исключение напрямую, не выдавая его, можно было бы в качестве альтернативы воспользоваться функцией `std::make_exception_ptr()`:

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

Если тип исключения известен и следует предпочесть именно его, то такой код выглядит намного понятнее, чем код с применением блока `try-catch`, — это не только упрощает код, но и расширяет возможности компилятора в области оптимизации кода.

Еще один способ сохранения исключения во фьючерсе заключается в уничтожении связанного с фьючерсом объекта `std::promise` или объекта `std::packaged_task` без вызова каких-либо `set`-функций в отношении промиса или без обращения к упакованной задаче. В любом случае деструктор `std::promise` или `std::packaged_task` сохранит исключение `std::future_error` с кодом ошибки `std::future_errc::broken_promise` в связанном состоянии, если фьючерс еще не перешел в состояние готовности: созданием фьючерса дается обещание предоставить значение или исключение, а уничтожением источника этого значения или исключения без их предоставления это обещание нарушается. Если бы компилятор в таком случае ничего не сохранял во фьючерсе, ожидающие потоки могли бы ожидать бесконечно.

До сих пор во всех примерах использовался объект `std::future`. Но у него есть ограничения, и не в последнюю очередь то, что результата может дожидаться только один поток. Если наступления одного и того же события нужно дожидаться сразу из нескольких потоков, следует вместо этого объекта воспользоваться объектом `std::shared_future`.

4.2.5. Ожидание сразу в нескольких потоках

Хотя `std::future` вполне справляется со всей синхронизацией, необходимой для переноса данных из одного потока в другой, вызовы компонентных функций из конкретного экземпляра `std::future` не синхронизированы друг с другом. Если обращаться к одному и тому же объекту `std::future` из нескольких потоков без дополнительной синхронизации, возникнут *состояние гонки за данными* и неопределенное поведение. Все предопределено конструкцией: `std::future` моделирует исключительное владение результатом асинхронных вычислений, а одноразовая природа функции `get()` лишает конкурентный доступ всякого смысла — значение можно извлечь только одним потоком, поскольку после первого же вызова `get()` значения для извлечения уже не останется.

Если же ваш великолепный проект конкуретного кода требует, чтобы ожидать наступления одного и того же события могли сразу несколько потоков, не стоит отчаиваться: именно это позволяет допустить `std::shared_future`. Если `std::future` *допускает только перемещение* (чтобы право владения передавалось между экземплярами, но чтобы в конкретный момент только один экземпляр ссылался на конкретный результат асинхронного вычисления), экземпляры `std::shared_future` *допускают копирование*, поэтому могут существовать сразу несколько объектов, ссылающихся на одно и то же связанное состояние.

Теперь при наличии `std::shared_future` компонентные функции отдельно взятого объекта по-прежнему не синхронизированы, и во избежание состояний гонок данных при обращении к одному и тому же объекту из нескольких потоков нужно защитить доступ с помощью блокировки. Предпочтительнее всего будет передать копию объекта `shared_future` каждому потоку, чтобы у всех них была возможность безопасно обращаться к собственному локальному объекту `shared_future`, поскольку теперь все внутренние компоненты корректно синхронизируются средствами библиотеки. Безопасность доступа к асинхронному состоянию из нескольких потоков обеспечивается, если каждый поток обращается к этому состоянию посредством собственного объекта `std::shared_future` (рис. 4.1).

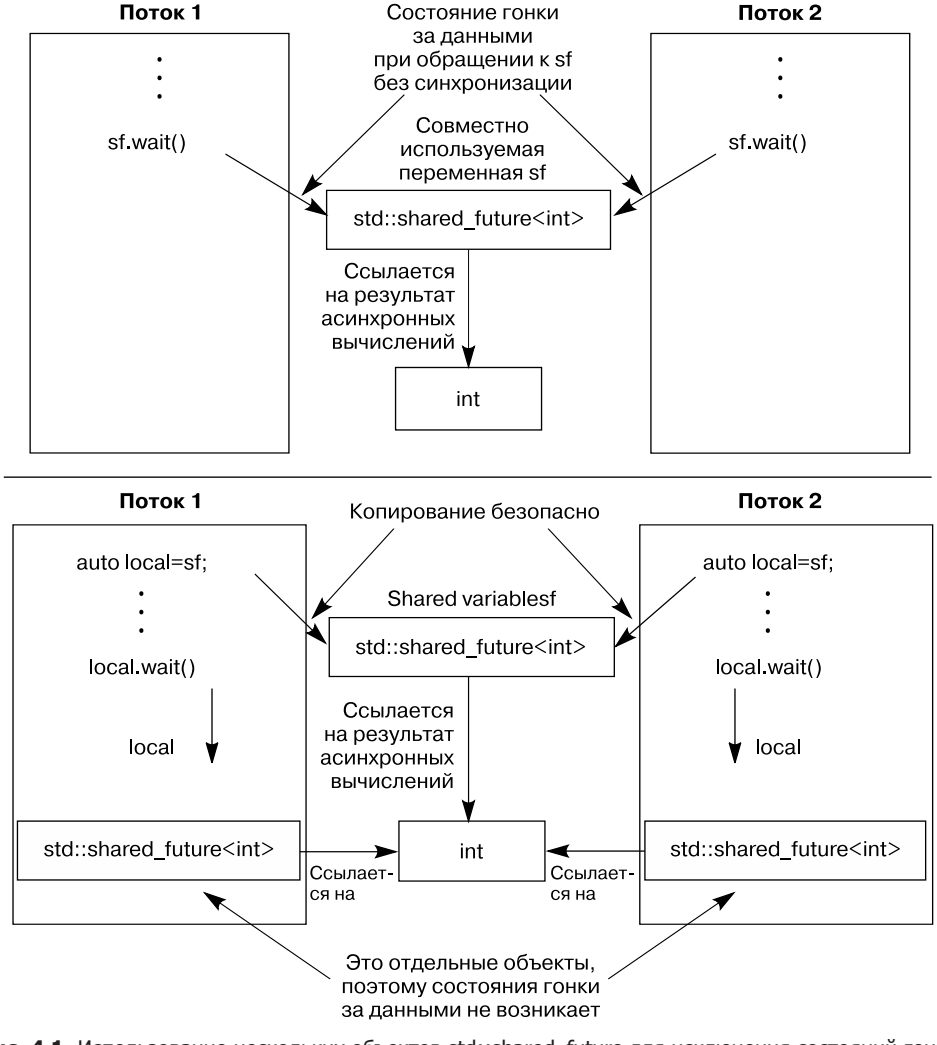


Рис. 4.1. Использование нескольких объектов std::shared_future для исключения состояний гонки за данными

Одним из возможных применений `std::shared_future` является реализация параллельного выполнения для чего-то похожего на сложную электронную таблицу: у каждой ячейки есть одно конечное значение, которое может использоваться формулами в нескольких других ячейках. Затем формулы для вычисления значений в зависящих ячейках могут использовать `std::shared_future`, чтобы ссылаться на первую ячейку. Если после этого все формулы для отдельных ячеек будут вычисляться в параллельном режиме, то те задачи, которые могут дойти до конца, будут выполнены, а те, что зависят от результатов вычислений других ячеек, окажутся заблокированы до разрешения зависимостей. Это позволит системе максимально воспользоваться возможностями оборудования для реализации конкурентности.

Экземпляры `std::shared_future`, ссылающиеся на некое асинхронное состояние, составлены из экземпляров `std::future`, ссылающихся на это состояние. Поскольку объекты `std::future` не владеют асинхронным состоянием совместно с любым другим объектом, право собственности должно быть передано в `std::shared_future` с помощью `std::move`, оставляя `std::future` в пустом состоянии, как будто этот объект является конструктором по умолчанию:

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());
std::shared_future<int> sf(std::move(f));
assert(!f.valid());
assert(sf.valid());
```

❶ Фьючерс f является допустимым

❷ f больше не является допустимым

❸ sf теперь является допустимым

Здесь фьючерс `f` изначально является допустимым ❶, поскольку ссылается на асинхронное состояние промиса `p`, но после переноса состояния в `sf` фьючерс `f` оказывается недопустимым ❷, а `sf` — допустимым ❸.

Как и для других перемещаемых объектов, передача владения для r-значений происходит подразумеваемо, поэтому `std::shared_future` можно создавать непосредственно из возвращаемого значения компонентной функции `get_future()` объекта `std::promise`, например:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future());
```

❶ Подразумеваемый перенос владения

Здесь перенос владения подразумевается и не требует явного указания: `std::shared_future<>` создается из r-значения типа `std::future<std::string>` ❶.

У `std::future` также имеется дополнительное средство, облегчающее использование `std::shared_future`, с новой функцией для автоматического вывода типа переменной из инициализатора (см. раздел А.6). В `std::future` есть компонентная функция `share()`, создающая новый экземпляр `std::shared_future` и непосредственно ему передающая владение асинхронным состоянием. Это обстоятельство может сэкономить массу времени при наборе кода и упростить внесение в него изменений:

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

В данном случае тип `sf` выводится как `std::shared_future< std::map< SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, что

и произнести-то трудно. Если компаратор или распределитель изменяются, вам нужно только изменить тип промиса, а тип фьючерса будет, соответственно, обновлен автоматически.

Иногда требуется лимитировать время ожидания наступления события либо из-за строгих ограничений времени на выполнение конкретного раздела кода, либо потому, что поток сможет сделать другую полезную работу, если событие не наступит в скором времени. Обеспечить эту возможность помогает то, что у многих функций ожидания имеются варианты, допускающие указание срока ожидания.

4.3. Ожидание с ограничением по времени

Все рассмотренные до сих пор блокирующие вызовы выполняли блокировку на неопределенный период времени, приостанавливая поток в ожидании наступления события. Во многих случаях этого вполне достаточно, но иногда требуется установить лимит ожидания. Это может позволить вам отправить что-нибудь вроде сообщения «Я еще жив» либо интерактивному пользователю, либо другому процессу или действительно дать возможность прервать ожидание, если пользователь не хочет ждать и нажал кнопку отмены.

Есть две разновидности указываемого срока ожидания: *продолжительность* (ожидание длится указанный период времени, например 30 мс) или *абсолютный срок* (ожидание длится до конкретного времени, например 17:30:15.045987023 UTC 30 ноября 2011 года). Большинство функций ожидания предоставляют варианты, соответствующие обеим этим разновидностям. У вариантов с указанием продолжительности имеется суффикс `_for`, а у вариантов с абсолютным сроком ожидания — суффикс `_until`.

Поэтому, к примеру, у `std::condition_variable` есть два переопределения компонентной функции `wait_for()` и два переопределения компонентной функции `wait_until()`, соответствующих двум переопределениям `wait()`. Первое переопределение ожидает поступления сигнала, или истечения срока ожидания, или наступления фиктивного пробуждения, а второе станет при пробуждении проверять предоставленный предикат и возвратит управление, только когда он равен `true` (и условной переменной поступил сигнал) или истек срок ожидания.

Прежде чем рассматривать подробности функций, использующих сроки ожидания, изучим способ указания времени в C++, начиная с часов.

4.3.1. Часы

Что касается стандартной библиотеки C++, часы являются источником информации о времени. Если конкретнее, то часы являются классом, предоставляющим четыре различных элемента информации:

- *текущее* время;
- тип значения, используемого для представления показаний времени, получаемых из часов;

- продолжительность такта часов;
- одинаковая ли тактовая частота у часов и считаются ли они в силу этого *стабильными*.

Текущее время часов можно получить, вызвав для данного класса часов статическую компонентную функцию `now()`. Например, в результате вызова `std::chrono::system_clock::now()` будет возвращено текущее время системных часов. Тип момента времени для конкретных часов указывается с помощью `typedef`-компонента `time_point`, поэтому возвращаемым типом `some_clock::now()` является `some_clock::time_point`.

Продолжительность такта часов указывается в виде дробного числа секунд, которое задается `typedef`-компонентом часов `period`: часы, тикающие 25 раз в секунду, имеют значение `period`, равное `std::ratio<1,25>`, а часы, тикающие каждые 2,5 с, имеют значение `period`, равное `std::ratio<5,2>`. Если определить продолжительность такта часов до выполнения программы невозможно или она может изменяться в ходе выполнения приложения, значение переменной `period` можно указать как среднюю продолжительность такта часов, наименьшую возможную продолжительность такта часов или какое-то иное значение, которое создатели библиотеки посчитали подходящим. Нет никакой гарантии, что наблюдаемая в ходе текущего выполнения программы продолжительность такта часов совпадет с продолжительностью, указанной для них в компоненте `period`.

Если часы ходят *с постоянной частотой* (независимо от того, соответствует эта частота значению `period` или нет) и не допускают *подведения*, значит, можно считать их *стабильными*. Статический компонент данных класса часов `is_steady` имеет значение `true`, если часы стабильны, и `false` — в противном случае. Обычно часы `std::chrono::system_clock` нестабильны, поскольку допускают подстройку, даже если она выполняется автоматически, чтобы учесть уход вперед локальных часов. Такая подстройка может стать причиной того, что при вызове `now()` будет возвращено время, предшествующее возвращенному предыдущим вызовом `now()`, что является нарушением требования к равномерности хода часов. Вскоре будет показано, что стабильные часы играют в вычислениях срока ожидания весьма важную роль, поэтому стандартная библиотека C++ предоставляет такие часы в виде `std::chrono::steady_clock`. Другими часами, имеющимися в стандартной библиотеке C++, являются упомянутые ранее `std::chrono::system_clock`, представляющие часы реального времени системы и предлагающие функции для преобразования их моментов времени в значения `time_t` и обратно, а также `std::chrono::high_resolution_clock`, которые дают наименьшую возможную продолжительность такта часов (а следовательно, и самое высокое возможное разрешение) среди всех часов, предоставляемых библиотекой. Возможно, будет еще и определитель типа `typedef` для каких-либо других часов. Эти часы наряду с другими средствами работы со временем определены в библиотечном заголовке `<chrono>`.

Вскоре дойдет очередь и до представлений моментов времени, но сначала рассмотрим представление продолжительности.

4.3.2. Продолжительность

Продолжительность — это самая простая часть поддержки времени, она регламентируется шаблоном класса `std::chrono::duration<>` (все средства работы со временем, имеющиеся в C++ и использующиеся Thread Library, находятся в пространстве имен `std::chrono`). Первым параметром шаблона является тип представления, например `int`, `long` или `double`, а вторым — дробное значение количества секунд, представляющее каждую единицу продолжительности. Например, количество минут, хранящееся в значении типа `short`, кратно `std::chrono::duration<short,std::ratio<60,1>>`, поскольку в минуте 60 с. В то же время число миллисекунд, хранящееся в значении типа `double`, кратно `std::chrono::duration<double,std::ratio <1,1000>>`, так как миллисекунда представляет собой одну тысячную (1/1000) секунды.

Стандартная библиотека предоставляет в пространстве имен `std::chrono` целый набор предопределенных `typedef`-компонентов для различных продолжительностей: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes` и `hours`. Все они используют довольно большой целочисленный тип для представления выбранной продолжительности, благодаря чему можно указывать в нужных единицах измерения продолжительность, превышающую 500 лет. Есть также `typedef`-компоненты для всех определенных в системе СИ степеней 10 от `std::atto` (10–18) до `std::exa` (1018) (и более, если ваша платформа поддерживает 128-разрядные целочисленные типы), используемые при указании пользовательской продолжительности, например `std::duration<double,std::centi>` для одной сотой секунды (1/100), представленной в значении типа `double`.

Для удобства представления продолжительности в пространстве имен `std::chrono_literals` есть несколько предопределенных операторов литеральных суффиксов (появились в C++14). Они позволяют упростить код, применяющий жестко запрограммированные значения продолжительности, например:

```
using namespace std::chrono_literals;
auto one_day=24h;
auto half_an_hour=30min;
auto max_time_between_messages=30ms;
```

При использовании с целочисленными литералами эти суффиксы эквивалентны применению предопределенных `typedef`-компонентов продолжительности, благодаря чему `15ns` и `std::chrono::nanoseconds(15)` будут идентичными значениями. Но при использовании с литералами чисел с плавающей точкой суффиксы создают соответствующим образом масштабируемую продолжительность, выраженную числом с плавающей точкой и неуказанным типом представления. Таким образом, `2.5min` будет равно `std::chrono::duration<`*`some-floating-point-type`*`,std::ratio<60,1>>`. Если нужно указать диапазон или точность реализации выбранного типа с плавающей точкой, следует сконструировать объект с подходящим представлением самостоятельно, а не пользоваться удобствами, предоставляемыми литеральными суффиксами.

Если значения не нужно усекать, то преобразования между продолжительностями выполняются предопределяемым образом (то есть часы преобразуются в секунды

именно так, а вот секунды в часы — нет). Явно заданное преобразование может выполняться с помощью `std::chrono::duration_cast<>`:

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

Результат получается усеченным, а не округленным, следовательно, значение `s` в данном примере — `54`.

Продолжительность поддерживает арифметику, поэтому ее можно складывать и вычитать для получения новой продолжительности или умножать либо делить на постоянное число базового типа представления, отраженного в первом параметре шаблона. Таким образом, `5*seconds(1)` — то же самое, что и `seconds(5)` или `minutes(1) - seconds(55)`. Количество единиц продолжительности можно получить с помощью компонентной функции `count()`. Так, например, значение, возвращаемое вызовом функции `std::chrono::milliseconds(1234).count()`, равно `1234`.

Ожидания, основанные на продолжительности, задаются с помощью экземпляра класса `std::chrono::duration<>`. Например, можно ждать готовности фьючерса до истечения 35 мс:

```
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 состояние, показывающее либо истечение срока ожидания, либо наступление ожидаемого события. В данном случае ожидается готовность фьючерса, следовательно, функция возвращает `std::future_status::timeout`, если срок ожидания истек, либо `std::future_status::ready`, если фьючерс готов, либо `std::future_status::deferred`, если связанная с фьючерсом задача отложена. Время ожидания, основанного на продолжительности, измеряется с использованием внутренних стабильных часов библиотеки, следовательно, 35 мс означает 35 мс фактической продолжительности, даже если системные часы в ходе ожидания были подстроены — переведены вперед или назад. Конечно, непредсказуемость системной диспетчеризации и варьирующаяся точность часов операционной системы означают, что между выдачей вызова потоком и возвращением управления может пройти значительно больше 35 мс.

Разобравшись с продолжительностью, нужно переходить к моментам времени.

4.3.3. Моменты времени

Момент времени для часов представляется с помощью экземпляра шаблона класса `std::chrono::time_point<>`, в первом параметре которого определяется, на какие часы делается ссылка, а во втором — единица измерения (специализация шаблона `std::chrono::duration<>`). Значение момента времени представляет собой промежуток времени (кратный заданной продолжительности), прошедший с указанного момента времени, называемого *эпохой отсчета часов*. Эта эпоха является базовым свойством, которое невозможно запросить или указать напрямую стандартом C++.

К числу типовых относятся эпохи с 00:00 1 января 1970 года и с момента начальной загрузки компьютера, на котором запущено приложение. Часы могут использовать эпохи совместно или же иметь независимые эпохи. Если двое часов совместно используют одну и ту же эпоху, `typedef`-компонент `time_point` в одном классе может определять другой компонент как тип часов, связанный с `time_point`. Хотя определить начало эпохи невозможно, для заданного компонента `time_point` можно получить время с начала эпохи, вызвав функцию `time_since_epoch()`. Эта компонентная функция возвращает значение продолжительности, указывающее количество времени с начала эпохи отсчета часов до конкретного момента времени.

Например, момент времени можно указать как `std::chrono::time_point<std::chrono::system_clock,std::chrono::minutes>`. В нем будет представлено время относительно системных часов, но выраженное в минутах, а не в единицах естественной точности этих часов, которая обычно выражается в секундах или еще меньших единицах времени.

Чтобы получать новые моменты времени, продолжительность можно складывать со значением `std::chrono::time_point<>` или вычитать из него, то есть вычисление вида `std::chrono::high_resolution_clock::now()` + `std::chrono::nanoseconds(500)` даст в результате время, наступающее через 500 нс. Это пригодится для вычисления абсолютного времени ожидания, когда известна максимальная продолжительность выполнения блока кода, но в нем имеются многочисленные вызовы функций ожидания или функций, не связанных с ожиданием и предшествующих функциям ожидания, но отнимающих часть отведенного времени.

Один момент времени можно вычесть из другого, использующего те же самые часы. В результате получится промежуток времени между двумя моментами. Этим, к примеру, можно воспользоваться для измерения времени выполнения блоков кода:

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

Параметр `clock` экземпляра класса `std::chrono::time_point<>` определяет не только эпоху отсчета. Когда функции `wait` передается момент времени, показывающий абсолютный срок ожидания, для измерения времени используется параметр `clock`. Это важно, когда часы изменены, поскольку `wait` отслеживает изменение часов и не вернет управление до тех пор, пока функция часов `now()` не возвратит значение времени, наступающее позже указанного срока ожидания. Если часы переведены вперед, это может сократить продолжительность ожидания, измеренную стабильными часами, а если переведены назад — увеличить общую продолжительность ожидания.

Нетрудно предположить, что моменты времени используются с вариантами функции `wait`, имеющими суффикс `_until` (до). Типичный пример применения —

смещение от момента *some-clock*::now(), вычисленного в определенном месте программы, хотя для диспетчеризации операций в видимом пользователю пространстве времени моменты времени, связанные с системными часами, можно получить преобразованием значения time_t, для чего используется статическая компонентная функция std::chrono::system_clock::to_time_point(). Например, если имеется максимум 500 мс ожидания наступления события, связанного с условной переменной, можно предпринять действия, схожие с показанными в листинге 4.11.

Листинг 4.11. Ожидание условной переменной с указанием определенного срока

```
#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 для условных переменных с ограничением по времени, если функции ожидания wait не передается никакого предиката. При этом ограничивается общая продолжительность цикла. Как показано в подразделе 4.1.1, применение цикла тогда, когда используются условные переменные и функции не передается предикат, обусловливается необходимостью обработки ложных пробуждений. Если в цикле воспользоваться функцией wait_for(), ожидание может продлиться почти все отведенное время, прежде чем произойдет ложное пробуждение, после чего при следующем прохождении цикла отсчет времени начнется заново. Это может повторяться произвольное количество раз, аннулируя все границы времени ожидания.

Усвоив основы задания сроков ожидания, можно переходить к функциям, с которыми используются эти сроки.

4.3.4. Функции, принимающие сроки ожидания

Самым простым примером использования срока ожидания является добавление задержки к обработке конкретного потока, чтобы он не отнимал время обработки у других потоков, когда ему нечего делать. Соответствующий пример был показан в разделе 4.1, где в цикле выполнялся опрос состояния флага done (сделано).

Этим занимались две функции: `std::this_thread::sleep_for()` и `std::this_thread::sleep_until()`. Они работали наподобие обычного будильника: поток впадал в спячку либо на указанный срок (с помощью `sleep_for()`), либо до тех пор, пока на наступал указанный момент времени (с помощью `sleep_until()`). Применение функции `sleep_for()` подходит для примеров, подобных приведенному в разделе 4.1, где что-то должно делаться периодически и значение имеет только лишь истекшее время. В то же время функция `sleep_until()` позволяет спланировать пробуждение потока в конкретный момент времени. Этот прием может пригодиться для инициирования создания резервных копий в полночь, или распечатки платежной ведомости в 06:00, или приостановки потока до тех пор, пока не будет обновлен следующий кадр при воспроизведении видео.

В сроках ожидания нуждаются не только средства организации спячки. Как уже было показано, сроки ожидания можно применять с условными переменными и фьючерсами. Их можно задействовать даже при попытке заблокировать мьютекс, если, конечно, он их поддерживает. Обычные `std::mutex` и `std::recursive_mutex` не поддерживают сроки ожидания установки блокировки, а вот `std::timed_mutex` и `std::recursive_timed_mutex` — поддерживают. Оба этих типа поддерживают компонентные функции `try_lock_for()` и `try_lock_until()`, которые пытаются получить блокировку в течение указанного времени или до наступления заданного момента времени. В табл. 4.1 показаны функции из стандартной библиотеки C++, способные принимать сроки ожидания, их параметры и возвращаемые значения. Параметры, перечисленные как `duration` (продолжительность), должны быть экземпляром класса `std::duration<>`, а те, что перечислены как `time_point` (момент времени), — экземпляром класса `std::time_point<>`.

Таблица 4.1

Класс/пространство имен	Функции	Возвращаемые значения
std::this_thread пространство имен	sleep_for(*duration*) sleep_until(*time_point*)	Неприменимо
std::condition_variable или std::condition_variable_ anywait_for(*lock*,*duration*) wait_until(*lock*,*time_point*)	std::cv_status::timeout или std::cv_status::no_timeout wait_for(*lock*,*duration*,*predicate*) wait_until(*lock*,*time_point*,*predicate*)	 bool — значение, возвращенное предикатом при пробуждении
std::timed_mutex, std::recursive_timed_mutex или std::shared_timed_ mutextry_lock_for(*duration*) try_lock_until(*time_point*)		bool — true, если блокировка была получена, в противном случае false
std::shared_timed_mutex	try_lock_shared_for(*duration*) try_lock_shared_until(*time_point*)	bool — true, если блокировка получена, false, если нет

Класс/пространство имен	Функции	Возвращаемые значения
std::unique_lock<TimedLockable>unique_lock(*lockable*, *duration*) unique_lock(*lockable*,*time_point*)	Неприменимо — owns_lock() на вновь созданном объекте возвращает true, если блокировка была получена, и false, если нет	
	try_lock_for(*duration*) try_lock_until(*time_point*)	bool — true, если блокировка получена, false, если нет
std::shared_lock<SharedTimedLockable>shared_lock(*lockable*,*duration*) shared_lock(lockable,time_point)	Неприменимо — owns_lock() на вновь созданном объекте возвращает true, если блокировка была получена, и false, если нет	
	try_lock_for(*duration*) try_lock_until(*time_point*)	bool — true, если блокировка получена, false, если нет
std::future<ValueType> или std::shared_future<ValueType>wait_for(*duration*) wait_until(*time_point*)	std::future_status::timeout если срок ожидания истек, std::future_status::ready, если фьючерс готов, или std::future_status::deferred, если фьючерс содержит отложенную функцию, которая еще не запущена	

Теперь, когда рассмотрены механизмы условных переменных, фьючерсов, промисов и упакованных задач, настало время расширить горизонты и посмотреть, как их можно использовать для упрощения синхронизации операций между потоками.

4.4. Применение синхронизации операций для упрощения кода

До сих пор в этой главе средства синхронизации рассматривались в качестве строительных блоков, позволяющих сконцентрироваться на операциях, требующих синхронизации, а не на механизмах ее реализации. Например, код можно упростить, используя более *функциональный* (в смысле *функционального программирования*) подход к конкурентному программированию. Вместо того чтобы позволять нескольким потокам непосредственно совместно использовать данные, можно снабжать каждую задачу нужными ей данными, а результат с помощью фьючерсов распространять среди любых других потоков, которым они необходимы.

4.4.1. Функциональное программирование с применением фьючерсов

Понятие *функционального программирования (FP)* относится к стилю программирования, при котором результат вызова функции зависит только от ее параметров и не зависит от любого внешнего состояния. Это перекликается с математическим

понятием функции и означает, что при неоднократном вызове функции с одними и теми же параметрами будут получены абсолютно одинаковые результаты. Это свойственно многим математическим функциям, имеющимся в стандартной библиотеке C++, например `sin`, `cos` и `sqrt`, и простым операциям над базовыми типами, например `3+3`, `6*9` или `1.3/4.7`. *Чистая* функция *не изменяет* никакое внешнее состояние, производимый ею эффект строго ограничивается возвращаемым значением.

Это упрощает осмысление порядка вещей, особенно когда дело касается конкурентности, поскольку исчезают многие проблемы, связанные с совместным использованием памяти, которые рассматривались в главе 3. При отсутствии изменений в совместно используемой памяти не будет никаких состояний гонок, следовательно, не потребуется защищать совместно используемые данные с помощью мьютексов. Достигаемое в результате упрощение при программировании конкурентных систем стало причиной роста популярности таких языков, как Haskell (http://www.haskell.org/), где все функции по умолчанию чистые. Поскольку в основном все средства относятся к категории чистых, *нечистые* функции, фактически *вносящие изменения* в совместно используемые данные, еще сильнее выделяются из общей массы и становится проще решать, насколько органично они вписываются в общую структуру приложения.

Но преимущества функционального программирования не ограничиваются лишь языками, в которых оно играет роль парадигмы по умолчанию. C++ является языком множества парадигм, предоставляющим все возможности написания программ в стиле функционального программирования. Еще проще это стало делать с выходом стандарта C++11, и по сравнению с C++98 с появлением лямбда-функций (см. раздел A.6), включением `std::bind` из Boost и TR1 и внедрением автоматического вывода типа переменных (см. раздел A.7) ситуация существенно улучшилась. Завершающим элементом пазла стали фьючерсы, сделавшие конкурентность в FP-стиле реальной в C++: фьючерс может передаваться между потоками, позволяя результату одного вычисления зависеть от результата другого, *не имея явного доступа к совместно используемым данным*.

Быстрая сортировка в стиле функционального программирования

Чтобы показать применение фьючерсов при написании конкурентных программ в FP-стиле, рассмотрим простую реализацию алгоритма быстрой сортировки Quicksort. Основной его замысел довольно прост: в списке значений выбирается некий опорный элемент, а затем список разбивается на два набора значений — тех, что меньше значения опорного элемента, и тех, что больше или равны ему. Отсортированная копия списка получается сортировкой двух наборов значений и возвращением отсортированного списка значений меньше значения опорного элемента, за которым следуют опорный элемент и отсортированный список значений, больших или равных ему. На рис. 4.2 показано, как список из десяти целых чисел проходит сортировку по этой схеме. В листинге 4.12 приведена последовательная реализация в FP-стиле, где список получается и возвращается по значению, а не сортируется на месте, как это делает функция `std::sort()`.

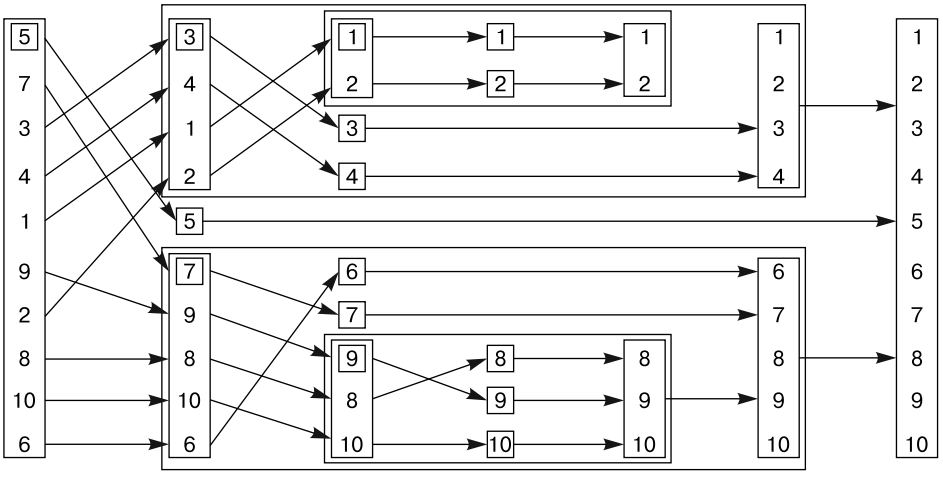


Рис. 4.2. Рекурсивная сортировка в FP-стиле

Листинг 4.12. Последовательная реализация Quicksort

```
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());
    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);
    auto new_lower(
        sequential_quick_sort(std::move(lower_part)));
    auto new_higher(
        sequential_quick_sort(std::move(input)));
    result.splice(result.end(),new_higher);
    result.splice(result.begin(),new_lower);
    return result;
}
```

❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽

Хотя интерфейс реализован в FP-стиле, если использовать этот стиль повсеместно, придется выполнять множество копирований, поэтому внутри применен обычный императивный стиль. В качестве опорного берется первый элемент, отрезаемый от начала списка с помощью функции `splice()` ❶. Несмотря на то что сортировка при этом может оказаться неоптимальной (в понятиях количества

сравнений и перестановок), при каких-либо других действиях с `std::list` может быть потрачено дополнительное время на обход элементов списка. Известно, что этот элемент планируется поместить в результат, поэтому его можно непосредственно вмонтировать в список, предназначенный для получения этого результата. Его же планируется задействовать для сравнений, поэтому, чтобы избежать копирования, возьмем ссылку на него ❷. Затем для разбиения последовательности на элементы, значения которых меньше значения опорного элемента и не меньше него, воспользуемся функцией `std::partition` ❸. Наипростейший способ задания критерия — использование лямбда-функции, а чтобы избежать копирования значения опорного элемента `pivot`, ссылка берется в замыкание (дополнительные сведения о лямбда-функциях изложены в разделе А.5).

Функция `std::partition()` переставляет элементы списка на месте и возвращает итератор, ссылающийся на первый элемент, значение которого не меньше значения опорного элемента. Полный тип итератора может быть очень длинным, поэтому используется лишь спецификатор типа `auto`, заставляющий компилятор выполнить работу за вас (см. раздел А.7).

Теперь, поскольку выбран интерфейс в FP-стиле, намереваясь воспользоваться рекурсией для сортировки двух «половин», нужно создать два списка. Для этого снова можно применить функцию `splice()`, чтобы переместить из списка `input` значения, предшествующие границе раздела `divide_point`, в новый список `lower_part` ❹. Тем самым все остальные значения останутся в списке `input`. Затем оба списка можно отсортировать с помощью рекурсивных вызовов ❺ и ❻. За счет использования для передачи списков функции `std::move()` здесь также удается избежать копирования — результат в любом случае перемещается подразумеваемым образом. И наконец, функцией `splice()` можно воспользоваться еще раз для получения результата в правильном порядке. Значения `new_higher` помещаются в конец ❼, после значения опорного элемента, а значения `new_lower` — в начало, до значения опорного элемента ❽.

Параллельная быстрая сортировка в FP-стиле

Поскольку функциональный стиль уже используется, теперь будет проще преобразовать код в версию, показанную в листинге 4.13, где выполняются параллельные вычисления с применением фьючерсов. Набор операций аналогичен предыдущему, за исключением того, что теперь некоторые из них запускаются в режиме параллельных вычислений. В этой версии реализуется алгоритм Quicksort с применением фьючерсов и FP-стиля.

Листинг 4.13. Параллельная быстрая сортировка Quicksort с использованием фьючерсов

```
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(
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));
    auto new_higher(
        parallel_quick_sort(std::move(input)));
    result.splice(result.end(),new_higher);
    result.splice(result.begin(),new_lower.get());
    return result;
}
```

❶ ❷ ❸ ❹

Заметное изменение здесь заключается в том, что вместо сортировки части списка с меньшими значениями в текущем потоке сортировка выполняется в другом потоке с использованием `std::async()` ❶. Часть списка со значениями, равными значению опорного элемента или превышающими его, сортируется, как и раньше, с помощью непосредственной рекурсии ❷. За счет рекурсивного вызова функции `parallel_quick_sort()` можно воспользоваться преимуществами оборудования, доступного для конкурентности. Если функция `std::async()` при каждом обращении к ней запускает новый поток, то при троекратной рекурсии вниз будут запущены восемь потоков, при десятикратной рекурсии вниз (для более чем 1000 элементов) — 1024 потока, если оборудование в состоянии справиться с таким количеством. Если библиотечные средства посчитают количество порожденных задач запредельным (возможно, потому, что оно превысило доступные оборудованию возможности конкурентности), они могут переключиться на синхронное порождение новых задач. Те станут запускаться в потоке, вызвавшем функцию `get()`, а не в новом потоке, позволяя тем самым избежать издержек, связанных с передачей задачи другому потоку в условиях, когда это не поможет повысить производительность. Стоит отметить, что реализация `std::async` идеально подходит для запуска нового потока для каждой задачи (даже в условиях существенного превышения лимита), если явно не установлен флаг `std::launch::deferred`, или для запуска всех задач в синхронном режиме, если явно не установлен флаг `std::launch::async`. Если для автоматического масштабирования полагаться на возможности библиотеки, нужно изучить документацию по конкретно используемой реализации, чтобы понять, какого поведения следует от нее ожидать.

Вместо использования функции `std::async()` можно, как показано в листинге 4.14, создать в качестве простой оболочки вокруг `std::packaged_task` и `std::thread` собственную функцию `spawn_task()`, при этом для результатов вызова функции создать объект `std::packaged_task`, получить из него фьючерс, запустить задачу в отдельном потоке и вернуть фьючерс. Больших преимуществ от этого ожидать не следует (и, скорее всего, окажется существенно превышен лимит возможностей оборудования), но тем самым будет проложен путь для перехода к более продуманной реализации, добавляющей задачи в очередь выполнения в пуле

рабочих потоков. Пулы потоков рассмотрим в главе 9. Скорее всего, смысл пойти по этому пути вместо того, чтобы воспользоваться `std::async`, появится, только если вы отдаете себе отчет в том, что делаете, и хотите получить полный контроль над способом создания пула потоков и выполнением задач в нем.

Вернемся к функции `parallel_quick_sort`. Поскольку для получения `new_higher` применялась прямая рекурсия, то составить его на месте можно, как и прежде ❸. А вот `new_lower` теперь представлен не списком, а объектом `std::future<std::list<T>>`, поэтому для получения значения перед тем, как появится возможность вызова функции `splice()`, нужно вызвать функцию `get()` ❹. Затем последуют ожидание завершения выполнения фоновой задачи и *перемещение* результата в вызов функции `splice()`, а функция `get()` возвратит ссылку на *r*-значение, содержащее результат, позволяя переместить его (дополнительные сведения о ссылках на *r*-значения и семантике перемещений можно найти в подразделе A.1.1).

Даже если предположить, что `std::async()` оптимально использует возможности оборудования по обеспечению конкурентности, это все же нельзя признать идеальной параллельной реализацией алгоритма Quicksort. Прежде всего на функцию `std::partition` возлагается слишком большой объем работы, и ее вызов все еще выполняется в последовательном режиме, но пока нас все устраивает. Если хочется получить самую быструю из возможных параллельных реализаций, обратитесь к научной литературе. В качестве альтернативы можно воспользоваться параллельным переопределением из стандартной библиотеки C++17 (см. главу 10).

Листинг 4.14. Примерная 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 — это не единственная парадигма конкурентности с отказом от совместно используемых изменяемых данных, есть еще парадигма взаимодействующих последовательных процессов (Communicating Sequential Processes, CSP[1]), где концептуально потоки полностью отделены друг от друга и не имеют совместно используемых данных, но задействуют каналы обмена данными, позволяющие сообщениям передаваться между собой. Эта парадигма принята языком программирования Erlang (http://www.erlang.org/) и средой интерфейса передачи сообщений MPI (Message Passing Interface; http://www.mpi-forum.org/), обычно

[1] *Hoare C. A. R.* Communicating Sequential Processes. — Prentice Hall, 1985; находится в свободном доступе в Сети по адресу http://www.usingcsp.com/cspbook.pdf.

используемой для высокопроизводительных вычислений в C и C++. Я уверен, что вас удивит сообщение о том, что при условии соблюдения определенных правил все это поддерживается и в C++. Один из способов такой поддержки рассматривается в следующем разделе.

4.4.2. Синхронизация операций путем передачи сообщений

Замысел, на котором базируется технология CSP, довольно прост: если нет совместно используемых данных, каждый поток может считаться абсолютно независимым исключительно на основе того, как он ведет себя в ответ на получаемые сообщения. В силу этого каждый поток, по сути, является механизмом состояния: при получении сообщения он каким-то образом обновляет свое состояние и, возможно, отправляет одно или несколько сообщений другим потокам в ходе обработки, выполняемой в зависимости от исходного состояния. Одним из способов (но не единственным) создания таких потоков будет формализация этих действий и реализация модели конечного автомата (finite state machine), также механизм состояния может подразумеваться в структуре приложения. Какой из методов будет лучше работать в любом заданном сценарии, зависит от конкретных требований к поведению в той или иной ситуации и опыта команды программистов. Но, что бы ни было выбрано для реализации каждого из потоков, разделение на независимые процессы может избавить от сложностей конкурентности с совместно используемыми данными, а поэтому упрощает программирование, снижая вероятность ошибочных решений.

У настоящих взаимодействующих последовательных процессов вообще нет совместно используемых данных, а весь обмен данными проходит через очередь сообщений, но поскольку потоки в C++ совместно работают с адресным пространством, выполнить это требование невозможно. Здесь вступает в силу соблюдение определенных правил: ответственность за то, что потоки гарантируют отказ от совместного использования данных, возлагается на создателей приложения или библиотеки. Разумеется, очередь сообщений должна применяться совместно, чтобы потоки могли обмениваться данными, но подробности реализации могут быть заключены в библиотечные средства.

Представим на минуту, что создается код для банкомата. Он должен обеспечивать взаимодействие с человеком, пытающимся снять денежную сумму, и с соответствующим банком, а также управлять физическим оборудованием, принимающим пластиковую карту клиента, выводящим на экран сообщения, обрабатывающим нажатие клавиш, выдающим деньги и возвращающим карту клиенту.

Одним из способов справиться со всем этим будет разбиение кода на три независимых потока: первый будет управлять физическим оборудованием, второй — отрабатывать логику поведения банкомата, а третий — обмениваться данными с банком. Эти потоки могут обмениваться данными, только передавая сообщения, а не совместно используя какие-либо данные. Например, поток, управляющий оборудованием, будет отправлять сообщение потоку логики, когда клиент вставит карту или нажмет кнопку, а поток логики станет отправлять потоку управления оборудованием сообщение, показывающее, сколько денег выдать, и т. д.

Логику банкомата можно смоделировать в том числе и в виде конечного автомата. Во всех состояниях поток ожидает приемлемого сообщения, которое затем обрабатывает. В результате может произойти переход в новое состояние и цикл продолжится. Состояния, участвующие в простой реализации, показаны на рис. 4.3. Система ждет, что клиент вставит карту. Как только это будет сделано, она станет ждать ввода пользователем цифр ПИН-кода. Пользователь может удалить последнюю цифру. Как только будет введено достаточное количество цифр, начнется проверка ПИН-кода. Если он неправильный, работа завершится, карта будет возвращена клиенту. Возобновится ожидание вставки карты каким-нибудь клиентом. Если введен правильный ПИН-код, от клиента ожидается либо отмена трансакции, либо выбор снимаемой суммы. Если выбрана отмена, работа завершится и карта будет возвращена. Если выбрана сумма снятия, то перед ее выдачей и возвращением карты или же до отображения на дисплее сообщения о нехватке средств на карт-счете и возвращения карты будет ожидаться подтверждение из банка. Конечно же, настоящий банкомат устроен гораздо сложнее, но для иллюстрации замысла вполне достаточно и этого.

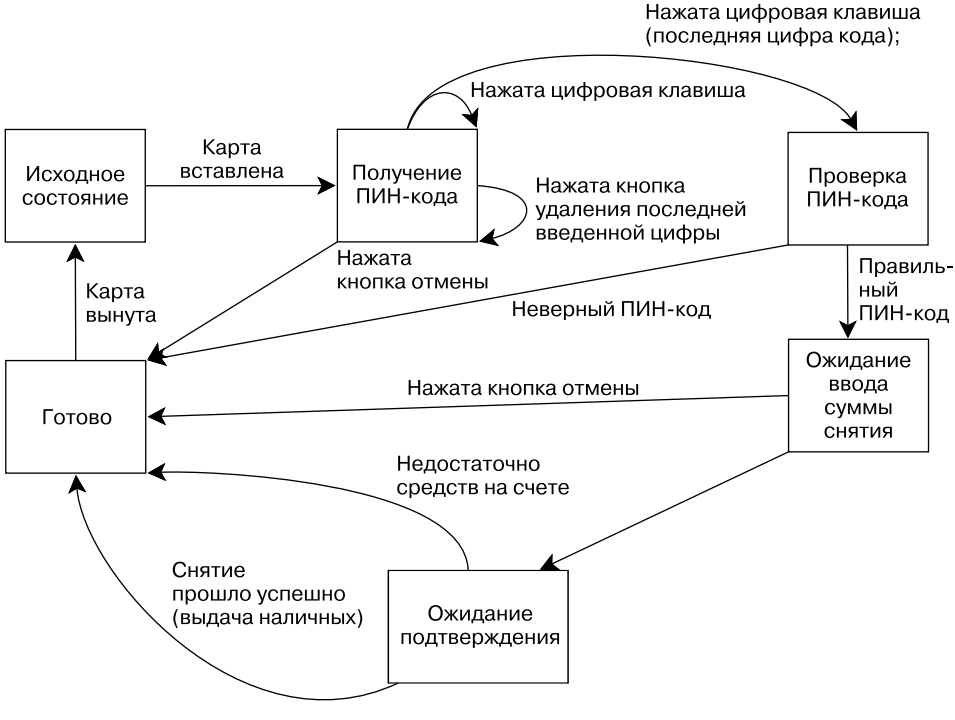


Рис. 4.3. Модель простого конечного автомата для банкомата

После разработки конструкции конечного автомата для логики банкомата можно приступить к ее реализации с помощью класса, имеющего компонентную функцию для представления всех состояний. Затем каждая компонентная функция может ждать конкретного набора входящих сообщений и обрабатывать их по мере поступления,

возможно инициируя при этом переключение в другое состояние. Каждый отдельно взятый тип сообщения представлен отдельной структурой `struct`. Часть простой реализации логики банкомата в такой системе показана в листинге 4.15, где имеются основной цикл и реализация первого состояния — ожидание вставки карты.

Можно заметить, что вся необходимая синхронизация для передачи сообщений скрыта в библиотеке передачи сообщений, базовая реализация которой приводится в приложении B вместе с полным кодом данного примера.

Листинг 4.15. Простая реализация класса логики 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()   <--❶
    {
        interface_hardware.send(display_enter_card());  <--❷
        incoming.wait()                                 <--❸
            .handle<card_inserted>(
                [&](card_inserted const& msg)  <--❹
                {
                    account=msg.account;
                    pin="";
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
                );
    }
    void getting_pin();
public:
    void run()    <--❺
    {
        state=&atm::waiting_for_card;  <--❻
        try
        {
            for(;;)
            {
                (this->*state)();  <--❼
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }
};
```

Как уже упоминалось, рассматриваемая здесь реализация является весьма упрощенной версией реальной логики, требующейся банкомату, но она дает представление о стиле программирования, основанном на передаче сообщений. Здесь не нужно задумываться о синхронизации и конкурентности, следует просто разобраться, какие сообщения можно получить в заданном месте, а какие — отправить из него. Конечный автомат для этой логики банкомата запускается в одном потоке, а остальные части системы, такие как интерфейс с банком и интерфейс терминала, — в отдельных потоках. Такой стиль проектирования программ называется *моделью акторов* — в системе есть несколько *акторов* (каждый запускается в отдельном потоке), отправляющих друг другу сообщения о необходимости выполнения конкретной задачи, и нет совместно используемого состояния, за исключением того, которое передается непосредственно через сообщения.

Выполнение начинается с запуска компонентной функции `run()` ❺, которая устанавливает исходное состояние ожидания карты `waiting_for_card` ❻, а затем периодически выполняет компонентную функцию, представляющую текущее состояние (что бы это ни было) ❼. Функции состояния являются простыми компонентными функциями класса `atm`. Функция состояния `waiting_for_card` ❶ также не отличается особой сложностью: она отправляет интерфейсу сообщение, предписывающее вывести на экран слова «Вставьте карту» ❷, а затем ожидает сообщения для последующей обработки ❸. Она может обработать только один тип сообщения, свидетельствующего о том, что карта вставлена, — `card_inserted` и делает это с помощью лямбда-функции ❹. Функции обработки `handle` можно передать любую функцию или функциональный объект, но для простого случая, такого как этот, проще всего воспользоваться лямбда-функцией. Заметьте, что вызов функции `handle()` встроен в цепочку вызова функции `wait()`, и если полученное сообщение не соответствует указанному типу, оно игнорируется и поток продолжает ждать получения соответствующего сообщения.

Сама лямбда-функция сохраняет номер счета с карты в компонентной переменной, сбрасывает текущий ПИН-код, отправляет оборудованию интерфейса сообщение, предписывающее вывести на дисплей сообщение с предложением клиенту ввести ПИН-код, и изменяет состояние на «получение ПИН-кода». Как только обработчик сообщения завершит свою работу, функция состояния вернет управление, после чего основной цикл вызовет новую функцию состояния ❼.

Функция состояния `getting_pin` немного сложнее: как показано на рис. 4.3, она может обрабатывать три разных типа сообщений. Ее код показан в листинге 4.16.

Листинг 4.16. Функция состояния getting_pin для простой реализации логики банкомата

```
void atm::getting_pin()
{
    incoming.wait()                                    ❶
        .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>(                    ❷
            [&](clear_last_pressed const& msg)
            {
                if(!pin.empty())
                {
                    pin.resize(pin.length()-1);
                }
            }
            )                                   ❸
        .handle<cancel_pressed>(
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}
```

На этот раз можно обработать три типа сообщений, поэтому у функции `wait()` имеется три вызова функции `handle()`, выстроенные в цепочку в конце ее вызова, ❶, ❷ и ❸. В каждом вызове функции `handle()` в качестве параметра шаблона указан тип сообщения, а затем передана лямбда-функция, принимающая в качестве параметра этот конкретный тип сообщения. Поскольку вызовы выстроены в цепочку таким образом, реализация функции `wait()` осведомлена, что следует ждать сообщения `digit_pressed`, `clear_last_pressed` или `cancel_pressed`. Сообщения любого другого типа вновь будут игнорироваться.

На этот раз получение сообщения не становится обязательным поводом для смены состояния. Например, при получении сообщения `digit_pressed` цифра, если она не последняя, добавляется к набираемому ПИН-коду. Затем основной цикл ❼ из листинга 4.15 снова вызовет функцию `getting_pin()`, чтобы ждать ввода следующей цифры (или удаления текущей цифры, или отмены операции).

Это соответствует поведению, показанному на рис. 4.3. Каждое состояние, вписанное в прямоугольник, реализовано в виде отдельной компонентной функции, ожидающей допустимые сообщения и соответствующим образом обновляющей состояние.

Как видите, данный стиль программирования способен существенно упростить задачу проектирования систем, работающих в конкурентном режиме, поскольку каждый поток может рассматриваться как абсолютно независимый. Это пример использования нескольких потоков для разделения решаемых вопросов, что требует от вас конкретных решений о распределении задач между потоками.

В разделе 4.2 уже упоминалось, что в технической спецификации по конкурентности (Concurrency TS) предоставляются расширенные версии фьючерсов. Основная суть расширений заключается в возможности указания *продолжений* (continuations) — дополнительных функций, запускаемых автоматически по готовности фьючерса. Воспользуемся возможностью и узнаем, как с помощью этих расширений можно упростить код.

4.4.3. Конкурентность, организованная в стиле продолжений с применением Concurrency TS

Спецификация Concurrency TS предоставляет в пространстве имен `std::experimental` новые версии классов `std::promise` и `std::packaged_task`, которые отличаются от своих `std`-оригиналов одним и тем же: вместо `std::future` они возвращают экземпляры `std::experimental::future`. Это позволяет использовать преимущества ключевого нововведения `std::experimental::future` — *продолжения*.

Предположим, что имеются запущенная задача, выдающая какой-либо результат, и фьючерс, который будет содержать этот результат, когда он станет доступен. И есть некий код, который нужно запустить, чтобы обработать этот результат. При использовании `std::future` пришлось бы ожидать готовности фьючерса с помощью либо полностью блокирующей компонентной функции `wait()`, либо одной из двух компонентных функций, `wait_for()` или `wait_until()`, позволяющих учитывать срок ожидания. Код, применяющий эти функции, может быть неудобен и слишком сложен. А вам нужно то, что можно выразить словами: «По готовности данных следует их обработать». Именно этим и занимаются продолжения, и неудивительно, что компонентная функция для добавления продолжения к фьючерсу называется `then()`. Если есть фьючерс `fut`, продолжение добавляется вызовом `fut.then(`*`продолжение`*`)`.

Класс `std::experimental::future`, как и класс `std::future`, допускает только однократное извлечение сохраняемого значения. Если это значение используется продолжением, значит, оно будет недоступно другому коду. Следовательно, когда с помощью `fut.then()` добавлено продолжение, исходный фьючерс `fut` становится *недействительным*. Вместо него в результате вызова функции `fut.then()` возвращается новый фьючерс для хранения результата вызова продолжения. Все это показано в следующем коде:

```
std::experimental::future<int> find_the_answer;
auto fut=find_the_answer();
auto fut2=fut.then(find_the_question);
assert(!fut.valid());
assert(fut2.valid());
```

Когда исходный фьючерс входит в состояние *готовности*, функция продолжения `find_the_question` планируется для запуска в неуказанном потоке. Это дает реализации свободу запуска в пуле потоков или другом потоке, управляемом библиотекой. На данном этапе реализация получает больше свободы, и это сделано намеренно в расчете на то, что при добавлении продолжений к будущему стандарту C++ разработчики смогут учесть свой опыт, чтобы лучше разобраться с выбором потоков и обеспечить пользователям подходящие механизмы для управления этим выбором.

В отличие от непосредственных вызовов `std::async` или `std::thread`, передавать аргументы функции продолжения нельзя, поскольку аргумент уже определен библиотекой — продолжению передается *готовый* фьючерс, содержащий результат, который инициирует продолжение. Если предположить, что функция `find_the_answer` возвращает значение типа `int`, функция `find_the_question`, на которую ссылались

в предыдущем примере, должна принять в качестве единственного параметра `std::experimental::future<int>`, например:

```
std::string find_the_question(std::experimental::future<int> the_answer);
```

Дело в том, что фьючерс, к которому было прикреплено продолжение, может в итоге содержать значение или исключение. Если бы фьючерс был подразумеваемо разыменован для передачи значения непосредственно продолжению, то библиотеке пришлось бы решать, как обрабатывать исключение, а при передаче фьючерса продолжению исключение можно было бы обработать этим продолжением. В простых случаях это можно сделать, вызвав функцию `fut.get()` и разрешив повторную выдачу исключения для его распространения из функции продолжения. Как и в случае с функциями, передаваемыми в `std::async`, исключения, покидающие продолжение, сохраняются во фьючерсе, содержащем результат выполнения продолжения.

Следует отметить, что в спецификации Concurrency TS не говорится о существовании эквивалента `std::async`, хотя реализацию можно предоставить в виде расширения. Создать соответствующую функцию нетрудно: как показано в листинге 4.17, нужно получить фьючерс с помощью `std::experimental::promise`, а затем породить новый поток, запускающий лямбда-функцию, которая устанавливает значение промиса равным возвращаемому значению предоставленной функции.

Листинг 4.17. Простой эквивалент std::async для фьючерсов Concurrency TS

```
template<typename Func>
std::experimental::future<decltype(std::declval<Func>()())>
spawn_async(Func&& func){
    std::experimental::promise<
        decltype(std::declval<Func>()())> p;
    auto res=p.get_future();
    std::thread t(
        [p=std::move(p),f=std::decay_t<Func>(func)]()
            mutable{
            try{
                p.set_value_at_thread_exit(f());
            } catch(…){
                p.set_exception_at_thread_exit(std::current_exception());
            }
    });
    t.detach();
    return res;
}
```

Результат функции сохраняется во фьючерсе, или же исключение, выданное из функции, перехватывается и сохраняется во фьючерсе точно так же, как происходит при вызове `std::async`. Здесь также используются функции `set_value_at_thread_exit` и `set_exception_at_thread_exit`, чтобы обеспечить должную очистку переменных `thread_local` до готовности фьючерса.

Значение, возвращенное в результате вызова функции `then()`, само по себе является полноценным фьючерсом. Это означает, что продолжения можно выстраивать в цепочку.

4.4.4. Выстраиваем продолжения в цепочку

Предположим, что нужно выполнить целый ряд задач, требующих много времени, и сделать это в асинхронном режиме, чтобы высвободить основной поток для решения других задач. Например, когда зарегистрированный пользователь входит в ваше приложение, может возникнуть потребность отправить его учетные данные на сервер для аутентификации, а затем, когда эти данные будут проверены на подлинность, выдать еще один запрос к серверу для получения информации об учетной записи пользователя и, наконец, когда информация будет извлечена, обновить отображение на дисплее с выводом соответствующей информации. Для этого можно создать последовательный код, похожий на приведенный в листинге 4.18.

Листинг 4.18. Простая последовательная функция обработки входа зарегистрированного пользователя

```
void process_login(std::string const& username,std::string const& password)
{
    try {
        user_id const id=backend.authenticate_user(username,password);
        user_data const info_to_display=backend.request_current_info(id);
        update_display(info_to_display);
    } catch(std::exception& e){
        display_error(e);
    }
}
```

Но последовательный код вас не устраивает — нужен асинхронный код, чтобы не блокировался поток пользовательского интерфейса. Обычная функция `std::async` позволяет поместить все в фоновый поток (листинг 4.19), но при этом он по-прежнему будет заблокирован, потребляя ресурсы в ходе ожидания завершения задач. При наличии множества задач может быть запущено большое количество потоков, не занятых ничем, кроме ожидания.

Листинг 4.19. Обработка входных данных пользователя с помощью одной асинхронной задачи

```
std::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return std::async(std::launch::async,[=](){
        try {
            user_id const id=backend.authenticate_user(username,password);
            user_data const info_to_display=
                backend.request_current_info(id);
            update_display(info_to_display);
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

Чтобы можно было обойтись без заблокированных потоков, нужен некий механизм для выстраивания цепочки задач, выполняемых по мере завершения предыду-

щих задач, то есть требуются продолжения. В листинге 4.20 показан тот же общий процесс, но на этот раз разбитый на серию задач, каждая из которых следует по цепочке за предыдущей как продолжение.

Листинг 4.20. Функция обработки входных данных пользователя с продолжениями

```
std::experimental::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return spawn_async([=](){
        return backend.authenticate_user(username,password);
    }).then([](std::experimental::future<user_id> id){
        return backend.request_current_info(id.get());
    }).then([](std::experimental::future<user_data> info_to_display){
        try{
            update_display(info_to_display.get());
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

Обратите внимание на то, как каждое продолжение получает в качестве единственного параметра экземпляр `std::experimental::future`, а затем использует `.get()` для извлечения упакованного значения. Это означает, что исключения распространяются по всей цепочке и вызов функции `info_to_display.get()` в заключительном продолжении выдаст исключение, если таковое было ранее выдано любой из предыдущих функций цепочки. Имеющийся здесь блок `catch` сможет обработать все исключения точно так же, как этот делал блок `catch`, показанный в листинге 4.18.

Если вызовы серверного блока происходят внутри функций по причине ожидания ими передаваемых по сети сообщений или ожидания завершения операции, выполняемой в базе данных, значит, вы еще не все сделали. Вы могли разбить задачу на части, но они по-прежнему блокируют вызовы, следовательно, вы все еще не избавились от заблокированных потоков. Нужно, чтобы вызовы сервера возвращали фьючерсы, приходящие в состояние готовности, когда будут готовы данные, без блокировки каких-либо потоков. В данном случае вызов функции `backend.async_authenticate_user(username,password)` должен возвратить не обычный идентификатор пользователя `user_id`, а фьючерс `std::experimental::future<user_id>`.

Можно подумать, что это усложнит код, поскольку возвращение фьючерса из продолжения даст `future<future<some_value>>`, иначе придется ставить вызовы `.then` внутри продолжений. К счастью, подобные мысли ошибочны: поддержка продолжений обладает великолепной особенностью под названием «будущее развертывание» (future-unwrapping). Если функция продолжения, передаваемая в вызов функции `.then()`, возвращает `future<some_type>`, то и вызов `.then()` возвратит `future<some_type>`. Это означает, что в окончательном виде код будет похож на тот, что показан в листинге 4.21, и в вашей цепочке асинхронных функций не будет блокировки.

Листинг 4.21. Функция обработки входных данных пользователя с полностью асинхронными операциями

```
std::experimental::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return backend.async_authenticate_user(username,password).then(
        [](std::experimental::future<user_id> id){
            return backend.async_request_current_info(id.get());
        }).then([](std::experimental::future<user_data> info_to_display){
            try{
                update_display(info_to_display.get());
            } catch(std::exception& e){
                display_error(e);
            }
        });
}
```

Этот код выглядит почти так же просто, как и последовательный код из листинга 4.18, но в нем чуть больше шаблонов вокруг вызовов `.then` и объявлений лямбда-функций. Если ваш компилятор поддерживает обобщенные лямбда-функции C++14, то вместо типов фьючерсов в параметрах лямбда-функций можно поставить указатель `auto`, что еще больше упростит код:

```
return backend.async_authenticate_user(username,password).then(
    [](auto id){
        return backend.async_request_current_info(id.get());
    });
```

Если нужно что-то более сложное, чем простой линейный поток управления, это можно реализовать помещением логики в одну из лямбда-функций, а для действительно сложного потока управления, вероятнее всего, понадобится создать отдельную функцию.

До сих пор основное внимание уделялось поддержке продолжений в `std::experimental::future`. Вполне ожидаемо продолжения поддерживаются и в `std::experimental::shared_future`. Разница здесь в том, что объекты `std::experimental::shared_future` могут иметь более одного продолжения и параметром продолжения является `std::experimental::shared_future`, а не `std::experimental::future`. Естественно, это идет вразрез с общей природой `std::experimental::shared_future`, поскольку на одно и то же совместно используемое состояние могут ссылаться сразу несколько объектов. И если разрешено только одно продолжение, возникнет состояние гонки между двумя потоками, каждый из которых будет пытаться добавить продолжения к собственным объектам `std::experimental::shared_future`. Конечно же, такое развитие событий нежелательно, поэтому разрешены несколько продолжений. При наличии нескольких разрешенных продолжений можно позволить им добавляться через один и тот же экземпляр `std::experimental::shared_future` вместо того, чтобы разрешать только одно продолжение на каждый объект. К тому же нельзя запаковывать совместно используемое состояние в одноразовый объект `std::experimental::future`, передаваемый первому продолжению, если есть намерение передать его и второму продолжению. Таким образом, параметр, переданный функции продолжения, должен быть также экземпляром `std::experimental::shared_future`:

```
auto fut=spawn_async(some_function).share();
auto fut2=fut.then([](std::experimental::shared_future<some_data> data){
    do_stuff(data);
    });
auto fut3=fut.then([](std::experimental::shared_future<some_data> data){
    return do_other_stuff(data);
    });
```

Поскольку `fut` вызывает функцию `share()`, он относится к типу `std::experimental::shared_future`, поэтому функция продолжения должна в качестве параметра принимать `std::experimental::shared_future`. Но значение, возвращаемое из продолжения, является простым объектом типа `std::experimental::future` и не используется совместно, пока для этого не будет что-нибудь сделано, поэтому и `fut2`, и `fut3` относятся к типу `std::experimental::future`.

Продолжения — не единственное усовершенствование фьючерсов в Concurrency TS, хотя, наверное, могут считаться самым важным. Предоставляются также две переопределяемые функции, позволяющие ждать, пока в состояние готовности не перейдут *одна из* связок фьючерсов или *все* связки.

4.4.5. Ожидание готовности более чем одного фьючерса

Предположим, имеется большой объем данных, предназначенных для обработки, и каждый элемент можно обработать независимо от других. Это прекрасная возможность воспользоваться доступным оборудованием, породив набор асинхронно выполняемых задач для обработки элементов данных, каждая из которых возвращает обработанные данные через фьючерс. Но если требуется ждать завершения всех задач, а затем собирать все результаты для какой-нибудь финальной обработки, могут возникнуть неудобства, поскольку придется по очереди дожидаться готовности каждого фьючерса, а затем собирать результаты. Если требуется собрать их с помощью еще одной асинхронной задачи, то придется либо породить ее с самого начала, чтобы она заняла ожидающий поток, либо организовать периодический опрос фьючерсов и породить новую задачу, когда в состояние готовности придут все фьючерсы. Пример кода показан в листинге 4.22.

Листинг 4.22. Сбор результатов из фьючерсов с помощью std::async

```
std::future<FinalResult> process_data(std::vector<MyData>& vec)
{
    size_t const chunk_size=whatever;
    std::vector<std::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            std::async(process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::async([all_results=std::move(results)](){
        std::vector<ChunkResult> v;
        v.reserve(all_results.size());
        for(auto& f: all_results)
```

```
        {
            v.push_back(f.get());  ◄──❶
        }
        return gather_results(v);
    });
}
```

Этот код порождает новую асинхронную задачу для ожидания результатов, а затем, когда все они становятся доступными, обрабатывает их. Но, поскольку поток ждет каждую задачу по отдельности, диспетчер ❶ будет многократно пробуждать его, как только окажется готов каждый результат, а затем он снова станет впадать в спячку, обнаружив, что другой результат еще не готов. Здесь не только задействуется ожидающий поток, но и добавляются дополнительные переключения контекста, как только каждый фьючерс придет в состояние готовности, что потребует дополнительных издержек.

Использование `std::experimental::when_all` позволяет избежать всех этих ожиданий и переключений. Набор фьючерсов, готовность которых будет ожидаться, передается функции `when_all`, а она возвращает новый фьючерс, который оказывается готовым, когда в таком состоянии будут все фьючерсы исходного набора. Затем этот фьючерс может применяться с продолжениями для планирования дальнейшей работы после того, как все фьючерсы будут готовы. Пример показан в листинге 4.23.

Листинг 4.23. Сбор результатов из фьючерсов с использованием std::experimental::when_all

```
std::experimental::future<FinalResult> process_data(
    std::vector<MyData>& vec)
{
    size_t const chunk_size=whatever;
    std::vector<std::experimental::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            spawn_async(
            process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::experimental::when_all(
        results.begin(),results.end()).then(    ◄──❶
        [](std::future<std::vector<
            std::experimental::future<ChunkResult>>> ready_results)
        {
            std::vector<std::experimental::future<ChunkResult>>
                all_results=ready_results .get();
            std::vector<ChunkResult> v;
            v.reserve(all_results.size());
            for(auto& f: all_results)
            {
                v.push_back(f.get());  ◄──❷
            }
            return gather_results(v);
        });
}
```

В данном случае функция `when_all` используется при ожидании готовности всех фьючерсов, а затем планирует работу функции, используя `.then`, а не `async` ❶. Хотя внешне лямбда-функция выглядит точно так же, в качестве параметра она получает вектор `results` (заключенный во фьючерс), а не замыкание, и вызов `get` в отношении фьючерса в строке ❷ не блокируется, поскольку в тот момент, когда выполнение доходит до этого места, все значения уже готовы. Таким образом, появляется возможность снизить нагрузку на систему, незначительно изменив код.

В дополнение к `when_all` есть функция `when_any`. Она создает фьючерс, переходящий в состояние готовности, когда будет готов *любой* из предоставленных фьючерсов. Это хорошо вписывается в сценарии, где для получения преимуществ от доступных средств конкурентности порождается сразу несколько задач, но при этом, как только в состояние готовности переходит первый фьючерс, должны будут совершаться какие-то действия.

4.4.6. Ожидание первого фьючерса в наборе с when_any

Предположим, что в большом наборе данных выполняется поиск значения, которое соответствует определенному критерию, но если таких значений несколько, то подойдет любое. Это самая подходящая цель для применения параллелизма — можно породить несколько потоков, каждый из которых проверяет поднабор данных и, если найдет подходящее значение, установит флаг, показывающий, что остальные потоки должны прекратить поиск, а затем установить итоговое возвращаемое значение. В данном случае продолжать обработку нужно, когда поиск завершится в какой-либо из задач, даже если другие задачи еще не завершены.

Здесь можно воспользоваться `std::experimental::when_any` для сбора фьючерсов в один набор и предоставить новый фьючерс, готовность которого наступает по готовности как минимум одного фьючерса из исходного набора. Функция `when_all` дала вам фьючерс, в который заключена вся коллекция переданных ей фьючерсов, в то же время функция `when_any` добавила еще один уровень, сочетающий коллекцию с индексным значением, показывающим, какой из фьючерсов инициировал готовность объединенного фьючерса в экземпляре шаблона класса `std::experimental::when_any_result`.

Пример такого применения `when_any` показан в листинге 4.24.

Листинг 4.24. Использование std::experimental::when_any для обработки первого же найденного значения

```
std::experimental::future<FinalResult>
find_and_process_value(std::vector<MyData> &data)
{
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_tasks = (concurrency > 0) ? concurrency : 2;
    std::vector<std::experimental::future<MyData *>> results;
    auto const chunk_size = (data.size() + num_tasks - 1) / num_tasks;
    auto chunk_begin = data.begin();
    std::shared_ptr<std::atomic<bool>> done_flag =
        std::make_shared<std::atomic<bool>>(false);
    for (unsigned i = 0; i < num_tasks; ++i) {        ←❶
```

```
        auto chunk_end =
            (i < (num_tasks - 1)) ? chunk_begin + chunk_size : data.end();
        results.push_back(spawn_async([=] {                          ❷
            for (auto entry = chunk_begin;
                !*done_flag && (entry != chunk_end);
                ++entry) {
                if (matches_find_criteria(*entry)) {
                    *done_flag = true;
                    return &*entry;
                }
        }
        return (MyData *)nullptr;
    }));
    chunk_begin = chunk_end;
}
std::shared_ptr<std::experimental::promise<FinalResult>> final_result =
    std::make_shared<std::experimental::promise<FinalResult>>();
struct DoneCheck {
    std::shared_ptr<std::experimental::promise<FinalResult>>
        final_result;

    DoneCheck(
        std::shared_ptr<std::experimental::promise<FinalResult>>
            final_result_)
        : final_result(std::move(final_result_)) {}

                                                                     ❹
    void operator()(
        std::experimental::future<std::experimental::when_any_result<
            std::vector<std::experimental::future<MyData *>>>>
        results_param) {
            auto results = results_param.get();
            MyData *const ready_result =
                results.futures[results.index].get();  ❺
            if (ready_result)
                final_result->set_value(               ❻
                    process_found_value(*ready_result));
            else {
                results.futures.erase(
                    results.futures.begin() + results.index);  ❼
                if (!results.futures.empty()) {
                    std::experimental::when_any(                    ❽
                        results.futures.begin(), results.futures.end())
                        .then(std::move(*this));
                } else {
                    final_result->set_exception(
                        std::make_exception_ptr(           ❾
                            std::runtime_error("Not found")));
                }
        }
    };

    std::experimental::when_any(results.begin(), results.end())     ❸
        .then(DoneCheck(final_result));
    return final_result->get_future();        ❿
}
```

Исходный цикл ❶ порождает асинхронные задачи, количество которых указано в переменной num_tasks, все они запускают лямбда-функцию из строки ❷. Эта лямбда-функция захватывается путем копирования, поэтому у каждой задачи будут собственные значения для начала и конца ее порции данных — chunk_begin и chunk_end, а также копия совместно используемого указателя — флага готовности done_flag. Это позволяет снять все вопросы на весь период выполнения кода.

После порождения всех задач требуется обработать то, что будет возвращено задачей. Это делается выстраиванием цепочки продолжений в вызове when_any ❸. На этот раз продолжения записаны в виде класса, так как предполагается их повторно рекурсивно использовать. Когда одна из исходных задач будет готова, инициируется оператор вызова функции DoneCheck ❹. Сначала эта функция извлекает значение из готового фьючерса ❺, а затем, если искомое значение было найдено, обрабатывает его и устанавливает итоговый результат ❻. В противном случае готовый фьючерс выбрасывается из коллекции ❼ и, если еще остались проверяемые фьючерсы, выдается новый вызов функции when_any ❽, инициирующей свое продолжение по готовности нового фьючерса. Если фьючерсов не осталось, значит, ни одна из задач нужное значение не нашла и вместо искомого значения сохраняется исключение ❾. Возвращаемым значением функции является фьючерс для итогового результата ❿. Есть и альтернативные способы решения данной задачи, но я надеюсь, что именно этот способ позволил продемонстрировать возможность применения функции when_any.

В обоих приведенных ранее примерах использования when_all и when_any применялись переопределения этих функций с указанием диапазона итераторов, которые принимали пару итераторов, обозначающих начало и конец набора фьючерсов, готовности которых нужно ожидать. У обеих функций имеются также вариативные формы, где они принимают ряд фьючерсов непосредственно в виде параметров функции. В таком случае результатом является фьючерс, содержащий кортеж (или кортеж содержится в when_any_result), а не вектор:

```
std::experimental::future<int> f1=spawn_async(func1);
std::experimental::future<std::string> f2=spawn_async(func2);
std::experimental::future<double> f3=spawn_async(func3);
std::experimental::future<
    std::tuple<
        std::experimental::future<int>,
        std::experimental::future<std::string>,
        std::experimental::future<double>>> result=
    std::experimental::when_all(std::move(f1),std::move(f2),std::move(f3));
```

Этим примером подчеркиваются важные обстоятельства, относящиеся ко всем случаям применения функций when_any и when_all, — они всегда реализуются из любых экземпляров std::experimental::future, переданных через контейнер, и получают свои параметры по значению, поэтому фьючерсы должны перемещаться в них явным образом или передаваться им на время.

Иногда событие, наступление которого ожидается, заключается в достижении набором потоков определенного места в коде или в завершении обработки этими потоками конкретного количества элементов данных. В таких случаях, наверное, больше подойдет использование защелки или барьера. Рассмотрим защелки и барьеры, предоставляемые в рамках спецификации Concurrency TS.

4.4.7. Защелки и барьеры в Concurrency TS

Сначала узнаем, что подразумевается под защелкой или барьером. *Защелка* представляет собой объект синхронизации, состояние готовности которого наступает, когда показание его счетчика уменьшается до нуля. Название появилось благодаря *«защелкиванию»* выходной информации — как только защелка готова, она находится в состоянии готовности вплоть до ее уничтожения. Получается, что защелка выступает в роли необременительного средства ожидания наступления серии событий.

А *барьер* — это повторно используемый компонент синхронизации, применяемый для внутренней синхронизации в наборе потоков. Поскольку защелке все равно, какие потоки уменьшают значение счетчика — то ли оно многократно уменьшается одним и тем же потоком, то ли каждый из нескольких потоков уменьшает это значение на единицу, или же имеет место сочетание этих двух вариантов, — достигать барьера каждый поток может только один раз за цикл. Когда потоки достигают барьера, они блокируются до тех пор, пока его не достигнут все задействованные потоки, после чего все они освобождаются от блокировки. Затем барьер можно использовать повторно, то есть потоки могут достичь барьера еще раз, чтобы дожидаться всех потоков для прохождения следующего цикла.

Устройство защелки проще устройства барьера, поэтому начнем с типа защелки `std::experimental::latch` из Concurrency TS.

4.4.8. Базовый тип защелки std::experimental::latch

Тип `std::experimental::latch` берется из заголовка `<experimental/latch>`. При создании защелки на основе `std::experimental::latch` в качестве единственного аргумента конструктора указывается исходное значение счетчика. Затем в качестве события, наступление которого ожидается, для защелки вызывается функция `count_down`, и защелка переходит в состояние готовности, когда счетчик доходит до нуля. Если нужно дожидаться готовности защелки, для нее вызывается функция `wait`, а если всего лишь проверить ее готовность — функция `is_ready`. И наконец, если нужно вести обратный отсчет, а затем ожидать, пока счетчик дойдет до нуля, можно вызвать функцию `count_down_and_wait`. Простой пример применения защелки показан в листинге 4.25.

Листинг 4.25. Ожидание наступления событий с использованием std::experimental::latch

```
void foo(){
    unsigned const thread_count=…;
    latch done(thread_count);      ❶
    my_data data[thread_count];
    std::vector<std::future<void> > threads;
    for(unsigned i=0;i<thread_count;++i)
        threads.push_back(std::async(std::launch::async,[&,i]{ ❷
            data[i]=make_data(i);
            done.count_down(); ❸
            do_more_stuff(); ❹
        }));
```

```
    done.wait();                    ❺
    process_data(data,thread_count); ❻
}   ❼
```

Эта конструкция с помощью функции `done` применяется с количеством событий, наступления которых следует дождаться ❶, а затем порождает соответствующее количество потоков, используя для этого `std::async` ❷. Затем каждый поток, прежде чем продолжит дальнейшую обработку ❹, инициирует обратный отсчет в защелке, когда создает надлежащую часть данных ❸. Прежде чем обработать созданные данные ❻, основной поток может дожидаться на защелке готовности всех данных ❺. Потенциально обработка данных в строке кода ❻ может выполняться одновременно с последними этапами обработки в каждом потоке ❹ — нет никакой гарантии, что все потоки завершатся еще до запуска деструкторов `std::future` в конце функции ❼.

Следует заметить, что лямбда-функция, переданная `std::async` в строке кода ❷, захватывает по ссылке все, за исключением параметра `i`, который захватывается по значению. Дело в том, что `i` является счетчиком цикла и захват этого параметра по ссылке приведет к состоянию гонки за данными и неопределенному поведению, поскольку к `data` и `done` требуется совместный доступ. Кроме того, в этом сценарии требуется лишь защелка, поскольку после достижения данными готовности потоки заняты дополнительной обработкой, в противном случае можно было бы дожидаться готовности всех фьючерсов, чтобы убедиться в завершении задач перед обработкой данных.

Доступ к данным в вызове `process_data` ❻ безопасен, даже если они сохранены задачами, запущенными в других потоках, поскольку защелка является объектом синхронизации. Изменения, видимые потоку, вызывающему `count_down`, гарантированно будут видимы потоку, возвращающемуся из вызова функции ожидания `wait` на том же самом объекте защелки. Формально вызов `count_down` *синхронизируется* с вызовом `wait` — что именно это означает, будет показано при рассмотрении низкоуровневых ограничений распределения памяти и синхронизации в главе 5.

Помимо защелок, спецификация Concurrency TS предоставляет барьеры — повторно используемые объекты синхронизации, работающие с группой потоков. Рассмотрим их.

4.4.9. Основной барьер std::experimental::barrier

Спецификацией Concurrency TS предусмотрены два типа барьеров, объявляемых в заголовке `<experimental/barrier>`: `std::experimental::barrier` и `std::experimental::flex_barrier`. По конструкции первый из них проще, и издержки при его использовании меньше, а второй — гибче, но уровень издержек выше.

Предположим, имеется группа потоков, применяемая к неким данным. Каждый поток может выполнять свою обработку данных независимо от других, поэтому синхронизация во время обработки не требуется, но прежде, чем можно будет обработать следующий элемент данных или выполнить последующую обработку, свою обработку должны завершить все потоки. Класс `std::experimental::barrier` нацелен именно на такой сценарий. Создается барьер со счетчиком, указывающим количество потоков, задействованных в группе синхронизации. Завершив обработку,

каждый поток доходит до барьера и ждет остальную группу, вызывая в отношении барьера функцию `arrive_and_wait`. Когда барьера достигает последний поток группы, со всех потоков снимается блокировка и барьер перезапускается. Затем потоки в группе могут возобновить обработку и в зависимости от того, какие действия возможны, либо обработать следующий элемент данных, либо продолжить работу, выполняя следующую стадию обработки.

Защелки при срабатывании находятся в состоянии готовности. Перейдя в это состояние, они в нем и остаются, а вот барьеры — нет, они снимают блокировку с ожидающих потоков, а затем перезапускаются, из-за чего их можно использовать повторно. Они синхронизируются только внутри группы потоков — один поток не может ожидать готовности барьера, если только он не будет единственным потоком в группе синхронизации. Потоки могут покинуть группу, явно вызвав в отношении барьера функцию `arrive_and_drop`. Тогда такой поток больше не сможет ждать готовности барьера и количество потоков, которым нужно дойти до барьера в следующем цикле, будет на единицу меньше количества таких потоков в текущем цикле (листинг 4.26).

Листинг 4.26. Использование std::experimental::barrier

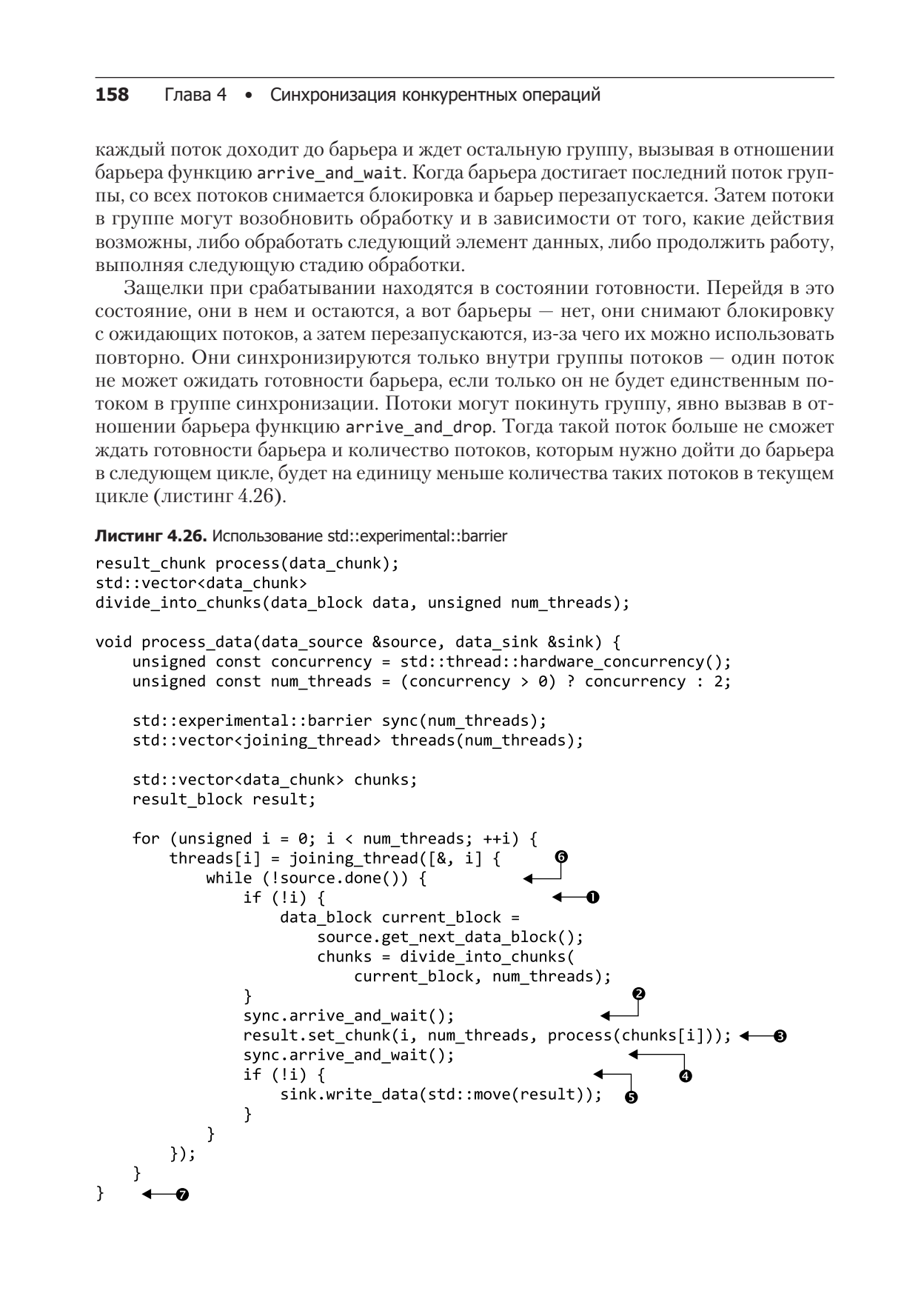
```
result_chunk process(data_chunk);
std::vector<data_chunk>
divide_into_chunks(data_block data, unsigned num_threads);

void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

    std::experimental::barrier sync(num_threads);
    std::vector<joining_thread> threads(num_threads);

    std::vector<data_chunk> chunks;
    result_block result;

    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = joining_thread([&, i] {
            while (!source.done()) {
                if (!i) {
                    data_block current_block =
                        source.get_next_data_block();
                    chunks = divide_into_chunks(
                        current_block, num_threads);
                }
                sync.arrive_and_wait();
                result.set_chunk(i, num_threads, process(chunks[i]));
                sync.arrive_and_wait();
                if (!i) {
                    sink.write_data(std::move(result));
                }
            }
        });
    }
}
```



В листинге 4.26 демонстрируется пример использования барьера для синхронизации группы потоков. Здесь имеются данные, поступающие из источника `sourse`, а данные, получаемые на выходе, записываются в их потребитель `sink`, но, чтобы воспользоваться доступной в системе конкурентностью, каждый блок данных разбивается на части по количеству потоков `num_threads`. Это должно выполняться последовательно, поэтому имеется исходный блок ❶, который запускается только в том потоке, для которого `i==0`. Затем, прежде чем достичь области параллельных вычислений, где каждый поток обрабатывает отдельную часть и обновляет результат ❸ перед повторной синхронизацией ❹, все потоки ожидают готовности барьера, наступающей по завершении выполнения последовательного кода ❷. Далее имеется вторая последовательная область, где только поток с нулевым номером записывает результат в `sink` ❺. После этого все потоки продолжают работу в таком же цикле до тех пор, пока источник `source` не сообщит, что все сделано ❻. Обратите внимание на то, что при завершении цикла каждым потоком последовательная область кода на дне цикла объединяется с областью на его вершине. Дело в том, что ко всему, что происходит в этих областях, имеет отношение только поток с нулевым номером. Это нормально, и все потоки будут синхронизированы при первом же использовании барьера ❷. Когда обработка будет выполнена, все потоки выйдут из цикла и деструкторы для объектов `joining_thread` станут ждать, пока все они не придут к финишу в конце внешней функции ❼ (объекты `joining_thread` впервые фигурировали в листинге 2.7).

Здесь нужно отметить, что вызовы функции `arrive_and_wait` находятся в тех местах кода, где важно, чтобы потоки приостановили свое выполнение, пока все они не будут готовы продолжать. В первой точке синхронизации все потоки ожидают достижения барьера потоком с нулевым номером, барьер — это четкая ограничительная линия, за которую ни один из них не переступает. Во второй точке синхронизации возникает обратная ситуация: именно поток с нулевым номером, прежде чем записать в `sink` результат завершения общей работы, дожидается достижения барьера всеми остальными потоками.

Спецификация Concurrency TS не ограничивается предоставлением только одного типа барьера, она предусматривает наряду с `std::experimental::barrier` более гибкий барьер `std::experimental::flex_barrier`. Одно из проявлений его гибкости — возможность запуска финальной последовательной области, когда все потоки достигли барьера, но прежде, чем они будут снова освобождены от блокировки.

4.4.10. Барьер std::experimental::flex_barrier — более гибкий соратник барьера std::experimental::barrier

Интерфейс барьера `std::experimental::flex_barrier` отличается от интерфейса `std::experimental::barrier` только одной особенностью — наличием дополнительного конструктора, получающего наряду с количеством потоков еще и функцию завершения. Она запускается только в одном потоке, достигшем барьера, как только до этого же барьера дойдут все потоки. С помощью барьера можно не только указать фрагмент кода, который должен выполняться в последовательном режиме, но и изменить количество потоков, которые должны дойти до барьера, чтобы начался

следующий цикл. Количество потоков может изменяться любым образом и становиться больше или меньше предыдущего количества. Проследить за тем, чтобы в следующий раз до барьера дошло нужное количество потоков, должен программист, использующий это средство.

В листинге 4.27 показано, как код листинга 4.26 можно переписать для использования `std::experimental::flex_barrier` с целью управления областью последовательно выполняемого кода.

Листинг 4.27. Применение std::flex_barrier для выполнения области последовательного кода

```
void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

    std::vector<data_chunk> chunks;

    auto split_source = [&] {          ←❶
        if (!source.done()) {
            data_block current_block = source.get_next_data_block();
            chunks = divide_into_chunks(current_block, num_threads);
        }
    };

    split_source();     ←❷

    result_block result;

    std::experimental::flex_barrier sync(num_threads, [&] { ←❸
        sink.write_data(std::move(result));
        split_source();                    ←❹
        return -1;      ←❺
    });
    std::vector<joining_thread> threads(num_threads);

    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = joining_thread([&, i] {     ❻
            while (!source.done()) {          ←┘
                result.set_chunk(i, num_threads, process(chunks[i]));
                sync.arrive_and_wait();  ←┐
            }                            ❼
        });
    }
}
```

Первое отличие данного кода от представленного в листинге 4.26 заключается в выделении лямбда-функции, разбивающей следующий блок данных на части ❶. Этот код вызывается до начала итерации ❷ и представляет собой код, который запускался в потоке с нулевым номером в начале каждой итерации.

Второе отличие состоит в том, что теперь sync-объектом является `std::experimental::flex_barrier` и наряду с количеством потоков ему передается функция завершения ❸. Эта функция запускается в одном потоке после того, как все потоки дойдут до барьера, и поэтому может представлять собой код, который нужно было

запускать в потоке с нулевым номером в конце каждой итерации. А затем выполняется вызов только что выделенной лямбда-функции разбиения данных из источника, которая должна вызываться в начале следующей итерации ❹. Возвращаемое значение `-1` ❺ показывает, что количество задействованных потоков должно оставаться неизменным, а возвращаемое значение 0 и более будет указывать количество потоков, задействованных в следующем цикле.

Основной поток ❻ теперь упрощен: в нем содержится только часть кода, выполняемого в параллельном режиме, поэтому нужна только одна точка синхронизации ❼. Таким образом, барьер `std::experimental::flex_barrier` позволяет упростить код.

Использование функции продолжения для выполнения последовательной части кода, как и изменение количества задействованных потоков, — весьма эффективное средство. Этой возможностью можно воспользоваться, например, для создания кода в стиле конвейера, где при начальном заполнении конвейера и его конечном опустошении количество потоков меньше, чем в ходе основной обработки, когда активны все стадии конвейера.

Резюме

Синхронизация операций между потоками — важная часть создания приложения, использующего конкурентность: если ее нет, потоки, по сути, являются независимыми, и их можно создать как отдельные приложения, запускаемые в виде группы, так как они выполняют связанные действия. В данной главе рассмотрены различные способы синхронизации операций: использование простых условных переменных, фьючерсов, промисов, упакованных задач, защелок и барьеров. Речь шла также о подходах к решению проблем синхронизации: о программировании в FP-стиле, где каждая задача дает результат, полностью зависящий от ее входных данных, а не от внешней среды, о передаче сообщений, при которой потоки связываются между собой с помощью асинхронных сообщений, отправленных через подсистему обмена сообщениями, выступающую в качестве посредника, а также о стиле продолжения, где для каждой операции задаются последующие задачи, а диспетчеризация возлагается на систему.

После обсуждения многих высокоуровневых средств, доступных в C++, настало время перейти к рассмотрению низкоуровневых механизмов, которые заставляют все это работать, — модели памяти C++ и атомарных операций.

5 Модель памяти C++ и операции над атомарными типами

В этой главе

- Особенности модели памяти C++.
- Атомарные типы, предоставляемые языком C++.
- Стандартная библиотека.
- Операции над атомарными типами.
- Как эти операции можно использовать для обеспечения синхронизации между потоками.

Одну из важнейших особенностей стандарта C++ многие программисты даже не замечают. Это не новые особенности синтаксиса и не новые библиотечные средства, а новая модель памяти, совместимая с многопоточными вычислениями. Без модели памяти с точным определением порядка работы основных строительных блоков нельзя было бы полагаться ни на одно их рассмотренных ранее средств. Причина, по которой большинство программистов не замечают этого, вполне понятна: если для защиты данных используются мьютексы, а для сигнальных событий — условные переменные, фьючерсы, защелки или барьеры, подробности того, что позволяет им работать, неважны. Тонкости модели памяти начинают интересовать только тогда, когда программист пытается подобраться ближе к «железу».

Помимо всего прочего, C++ является языком системного программирования. Одной из целей Комитета по стандартизации было стремление добиться того,

чтобы, кроме C++, не понадобился никакой другой язык более низкого уровня. C++ должен был обеспечить программистам достаточную гибкость для удовлетворения любых потребностей без каких-либо помех со стороны самого языка, позволив по мере необходимости приблизиться к «железу». Именно это позволяют сделать атомарные типы и операции, предоставляя средства для низкоуровневой синхронизации операций, которые обычно сводятся к одной-двум инструкциям центрального процессора.

Сначала в этой главе будут рассмотрены основные особенности модели памяти, затем внимание переместится на атомарные типы и операции, и, наконец, поговорим о различных типах синхронизации, доступных при выполнении операций над атомарными типами. Тема эта непростая, и, если вы не планируете создавать код, использующий для синхронизации атомарные операции (например, структуры данных без блокировок, рассматриваемые в главе 7), знать эти подробности вам ни к чему.

Для начала рассмотрим основы модели памяти.

5.1. Основы модели памяти

У модели памяти две составляющие: базовые *структурные* аспекты, относящиеся к тому, как и что располагается в памяти, и аспекты, относящиеся к *конкурентности*. Структурные аспекты важны для конкурентности, особенно если обратить внимание на низкоуровневые атомарные операции, поэтому начнем именно с них. Все в C++ касается объектов и их размещения в памяти.

5.1.1. Объекты и их размещение в памяти

Все данные в программах на C++ состоят из *объектов*. Это еще не значит, что новый класс можно создать в виде производного от класса `int`, или что у элементарных типов имеются компонентные функции, или что-то иное, наличие чего подразумевается, когда про такие языки, как Smalltalk или Ruby, говорят: «Объектами является абсолютно все». Это всего лишь утверждение, показывающее, что блоки данных в C++ создаются из объектов. В стандарте C++ объект определяется как область хранения, хотя далее в нем говорится о присваивании объектам таких свойств, как их типы и время жизни.

Некоторые объекты, например `int` или `float`, являются простыми значениями элементарного типа, а другие — экземплярами классов, определенных пользователями. У некоторых объектов, например массивов, экземпляров производных классов и экземпляров классов с нестатическими компонентными данными, имеются подобъекты, а у других их нет.

Объект любого типа хранится в одной или нескольких *областях памяти*. Каждая область памяти является либо объектом (или подобъектом) скалярного типа, например `unsigned short` или `my_class*`, либо последовательностью смежных битовых полей. Если используются битовые поля, то следует учесть одно важное обстоятельство: хотя смежные битовые поля — это отдельные объекты, они все же считаются

одной областью памяти. На рис. 5.1 показано, как структура `struct` делится на объекты и области памяти.

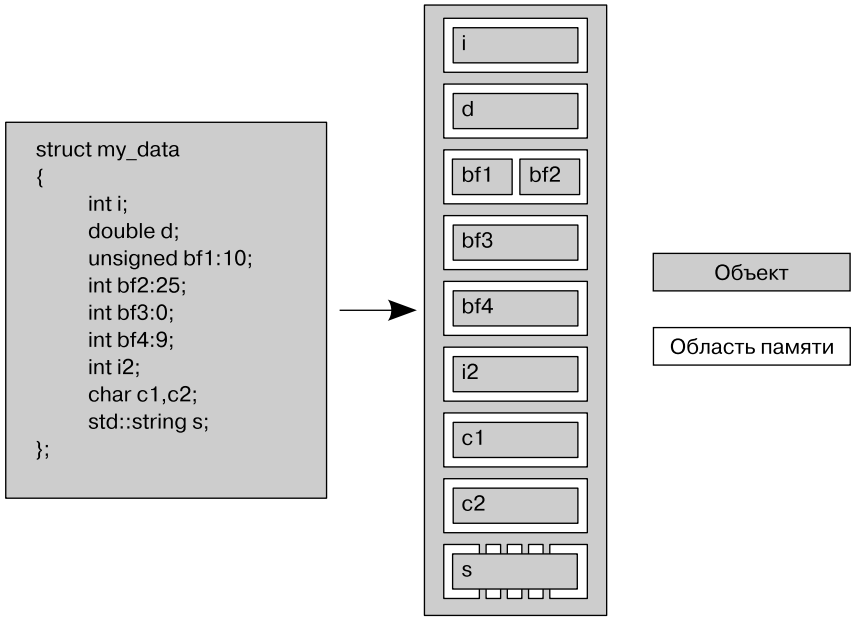


Рис. 5.1. Деление структуры на объекты и области памяти

Сначала вся структура представляет собой один объект, состоящий из нескольких подобъектов, по одному для каждого компонента данных. Битовые поля `bf1` и `bf2` используют общую область памяти, объект `std::string` по имени `s` состоит из нескольких областей памяти, а у всех остальных компонентов имеются собственные области памяти. Следует заметить, что битовое поле нулевой длины `bf3` (имя закомментировано, поскольку битовые поля нулевой длины должны быть безымянными) выделяет поле `bf4` в его собственную область памяти, но своей собственной области памяти не имеет.

На основании этого можно сделать четыре важных вывода.

- Каждая переменная — это объект, включая и те переменные, которые являются компонентами других объектов.
- Каждый объект занимает *как минимум одну* область памяти.
- Переменные элементарных типов, например `int` или `char`, всегда занимают только одну область памяти независимо от своего размера, даже если они расположены рядом или являются частью массива.
- Смежные битовые поля являются частью одной и той же области памяти.

Думаю, что вам интересно узнать, как все это связано с конкурентностью, поэтому перейдем к выяснению этого вопроса.

5.1.2. Объекты, области памяти и конкурентность

Теперь обратимся к тому, что имеет решающее значение для многопоточных приложений в C++: все зависит от областей памяти. Если два потока обращаются к *разным* областям памяти, проблем не возникает, все работает нормально. Но когда два потока обращаются к *одной и той же* области памяти, следует проявлять осторожность. Если область памяти не обновляется ни одним из потоков, то все в порядке, поскольку данные, предназначенные только для чтения, не нуждаются в защите или синхронизации доступа к ним. Но если какой-либо из потоков вносит в данные изменения, может сложиться состояние гонки, рассмотренное в главе 3.

Чтобы избежать состояния гонки, необходимо принудительно установить очередность доступа для двух потоков. Это может быть фиксированная очередность, при которой один поток всегда обращается к данным раньше другого, или же очередность, изменяемая от одного запуска приложения к другому, но с гарантией *определенного* порядка очередности. Один из способов обеспечения определенного порядка очередности, рассмотренный в главе 3, заключается в применении мьютексов: если один и тот же мьютекс блокируется до обоих обращений, то в конкретный момент времени к области памяти может обращаться только один поток, то есть одно обращение должно произойти прежде, чем произойдет другое (хотя заранее узнать, которое из них будет первым, как правило, невозможно). Другим способом является использование синхронизационных свойств *атомарных* операций (определение которых дается в разделе 5.2) в отношении или той же, или другой области, при этом в двух потоках происходит принудительная установка очередности доступа. Применение атомарных операций для принудительной установки очередности доступа рассматривается в разделе 5.3. Если к одной и той же области памяти обращается более двух потоков, определенная очередность должна быть у каждой пары обращений.

Если не определяется принудительная очередность между двумя обращениями к одной и той же области памяти от отдельных потоков, причем одно или оба обращения не являются атомарными, а также если одно или оба обращения выполняются для записи, возникает состояние гонки, вызывающее неопределенное поведение.

Важность этого утверждения трудно переоценить: неопределенное поведение — одно из самых темных пятен C++. Согласно стандарту языка, если в приложении наблюдается какое-то неопределенное поведение, то снимаются все гарантии, поведение приложения становится неопределенным, а его действия — непредсказуемыми. Мне известен пример неопределенного поведения, в результате которого загорелся монитор. Хотя с вами такое вряд ли произойдет, состояние гонки за данными, несомненно, является результатом серьезной ошибки и должно быть исключено любой ценой.

В этом утверждении есть еще один важный пункт: неопределенного поведения можно избежать, обратившись к областям памяти, вовлеченным в возможное состояние гонки, с помощью атомарных операций. Это не предотвратит само состояние гонки, поскольку по-прежнему не определено, какая из атомарных операций первой

доберется до области данных, но все-таки вернет программу в область определенного поведения.

Прежде чем рассматривать атомарные операции, нужно усвоить еще одну важную концепцию, касающуюся объектов и областей памяти, — очередность внесения изменений.

5.1.3. Очередность внесения изменений

У каждого объекта в программе на языке C++ имеется *очередность внесения изменений*, определенная на основе всех записей в этот объект из всех потоков выполнения программы, начиная с инициализации объекта. В большинстве случаев очередность будет варьироваться от запуска к запуску, но при любом отдельно взятом выполнении программы все имеющиеся в системе потоки должны согласовать очередность. Если рассматриваемый объект относится к одному из атомарных типов, описание которых дается в разделе 5.2, то вы должны убедиться в наличии достаточной синхронизации, гарантирующей согласованность очередности внесения изменений в каждую переменную. Если разным потокам видна разная последовательность значений одной и той же переменной, значит, возникло состояние гонки за данными, вызывающее неопределенное поведение (см. подраздел 5.1.2). Если используются атомарные операции, то за наличие необходимой синхронизации отвечает компилятор.

Это требование запрещает определенные виды спекулятивного выполнения, поскольку после того, как поток увидел конкретное значение объекта при данной очередности внесения изменений, последующие операции чтения в том же потоке должны возвращать более поздние значения, а последующие операции записи в тот же объект в потоке должны происходить позже при данной очередности внесения изменений. Кроме того, чтение объекта, которое следует за записью в него в том же потоке, должно возвращать либо записанное значение, либо другое значение, которое появилось позже в порядке очередности внесения изменений в этот объект. Несмотря на то что все потоки должны согласовывать очередность внесения изменений в каждый отдельно взятый объект в программе, им не обязательно согласовывать относительный порядок операций над отдельными объектами. Более подробно очередность выполнения потоками операций рассмотрена в подразделе 5.3.3.

Что же представляет собой атомарная операция и как такие операции можно использовать для принудительной установки очередности?

5.2. Атомарные операции и типы в C++

Атомарными называются неделимые операции. Ни один поток в системе не может наблюдать такую операцию на полпути к завершению: она либо выполнена, либо нет. Если операция загрузки, читающая значение объекта, является *атомарной* и все изменения этого объекта также являются атомарными, то в ходе загрузки будет извлечено либо исходное значение объекта, либо значение, сохраненное одним из изменений.

А вот при выполнении неатомарных операций другой поток может наблюдать их на полпути к завершению. Если неатомарная операция составлена из атомарных операций (таково, например, присваивание значения структуре с атомарными компонентами), то другие потоки могут видеть некоторое подмножество компонентов атомарной операции завершенным, а другие подмножества — еще не запущенными. Поэтому может наблюдаться или в конечном счете быть получено значение, представляющее собой смесь различных сохраненных значений. В любом случае несинхронизированный доступ к неатомарным переменным приводит к простому проблемному состоянию гонки, рассмотренному в главе 3, но на данном уровне это может быть состояние *гонки за данными* (см. раздел 5.1), вызывающее неопределенное поведение.

Для получения атомарных операций в C++ в большинстве случаев следует использовать атомарные типы, поэтому посмотрим, что это такое.

5.2.1. Стандартные атомарные типы

Определения стандартных *атомарных типов* можно найти в заголовке `<atomic>`. Все операции над этими типами являются атомарными, и в смысле принятого в языке определения атомарными являются только те операции, которые выполняются над этими типами, хотя сделать операции кажущимися атомарными можно с помощью мьютексов. Фактически такой эмуляцией могут воспользоваться и стандартные атомарные типы: все или почти все они имеют компонентную функцию `is_lock_free()`, которая позволяет пользователям определить, выполняются ли операции над данным типом непосредственно с использованием атомарных инструкций (`x.is_lock_free()` возвращает `true`) или же с помощью блокировки, внутренней по отношению к компилятору или библиотеке (`x.is_lock_free()` возвращает `false`).

Во многих случаях важно знать, что в основном атомарные операции используются в качестве замены операции, в которой в ином варианте для синхронизации применялся бы мьютекс. Если же сами атомарные операции задействуют внутренний мьютекс, то ожидаемый прирост производительности, вероятно, не проявится и, возможно, вместо этого стоит воспользоваться реализацией на основе мьютекса, в которой сложнее ошибиться. Именно так складываются обстоятельства вокруг структур данных без блокировки, например вокруг тех, что будут рассмотрены в главе 7.

Фактически важность данного вопроса настолько высока, что библиотека предоставляет целый набор макросов для того, чтобы в ходе компиляции определить, являются ли атомарные типы для различных целочисленных типов не имеющими блокировок.

В выходом C++17 у всех атомарных типов появились статические компонентные `constexpr`-переменные `X::is_always_lock_free`, получающие значение `true` только в том случае, если атомарный тип `X` не имеет блокировок для всего поддерживаемого оборудования, на котором может выполняться вывод текущей компиляции. Например, для заданной целевой платформы `std::atomic<int>` может никогда не иметь блокировки, следовательно, `std::atomic<int>::is_always_lock_free` получит

значение true. Но std::atomic<uintmax_t> может не иметь блокировки только в том случае, если оборудование, на котором выполняется программа, поддерживает необходимые инструкции. Следовательно, это свойство времени выполнения, и при компиляции на этой платформе std::atomic<uintmax_t>::is_always_lock_free получит значение false.

В число макросов входят ATOMIC_BOOL_LOCK_FREE, ATOMIC_CHAR_LOCK_FREE, ATOMIC_CHAR16_T_LOCK_FREE, ATOMIC_CHAR32_T_LOCK_FREE, ATOMIC_WCHAR_T_LOCK_FREE, ATOMIC_SHORT_LOCK_FREE, ATOMIC_INT_LOCK_FREE, ATOMIC_LONG_LOCK_FREE, ATOMIC_LLONG_LOCK_FREE и ATOMIC_POINTER_LOCK_FREE. С их помощью определяется статус отсутствия блокировки соответствующих атомарных типов для указанных встроенных типов и их беззнаковых аналогов (LLONG означает long, а POINTER относится ко всем типам указателей). Они вычисляются в 0, если атомарный тип *всегда* имеет блокировку, в значение 2, если атомарный тип *никогда* не имеет блокировки, и в 1, если статус не имеющего блокировки соответствующего атомарного типа упомянут ранее в качестве свойства времени выполнения.

Единственный тип, не предоставляющий компонентную функцию is_lock_free(), — это std::atomic_flag. Он является простым булевым флагом, и операции над ним должны быть без блокировок. Простым булевым флагом без применения блокировок можно воспользоваться как основой для реализации простой блокировки и всех остальных атомарных типов. Говоря о *простоте*, я подразумеваю следующее: объекты типа std::atomic_flag инициализируются в виде сброшенного флага, а затем могут быть либо запрошены и установлены (с помощью компонентной функции test_and_set()), либо сброшены (с помощью компонентной функции clear()). И больше ничего: никаких присваиваний, конструкций копирования, проверок со сбросом. И вообще больше никаких операций.

Все остальные атомарные типы доступны через специализации шаблона класса std::atomic<>, обладают более широкими функциональными возможностями, но могут, как объяснялось ранее, задействовать блокировки. Ожидается, что на наиболее популярных платформах атомарные варианты всех встроенных типов, например std::atomic<int> и std::atomic <void*>, действительно реализованы без применения блокировок. Однако этого не требуется. Как вскоре будет показано, в интерфейсе каждой специализации отражаются свойства типа. К примеру, такие поразрядные операции, как &=, не определены для обычных указателей, поэтому не определены и для атомарных указателей.

Для ссылки на определенные в конкретной реализации атомарные типы можно воспользоваться не только непосредственно шаблоном класса std::atomic<>, но и набором имен, приведенных в табл. 5.1. Из-за того, в каком порядке происходило добавление атомарных типов в стандарт C++, при использовании более старого компилятора эти альтернативные имена типов могут ссылаться либо на соответствующую специализацию std::atomic<>, либо на ее базовый класс. А в компиляторах, полностью поддерживающих стандарт C++17, они всегда будут псевдонимами соответствующих специализаций std::atomic<>. Смешивание альтернативных имен с непосредственным названием специализаций std::atomic<> может привести к тому, что код нельзя будет перенести на другую платформу.

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_uchar	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>

Наряду с основными атомарными типами стандартная библиотека C++ предоставляет набор `typedef`-синонимов для атомарных типов, соответствующих различным неатомарным `typedef`-синонимам стандартной библиотеки, например `std::size_t`. Этот набор показан в табл. 5.2.

Таблица 5.2. Стандартные атомарные typedef-синонимы и соответствующие им встроенные typedef-синонимы

Атомарные typedef-синонимы	**Соответствующие typedef-синонимы 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

Продолжение ⇩

Таблица 5.2 (продолжение)

Атомарные typedef-синонимы	Соответствующие typedef-синонимы стандартной библиотеки
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

Глаза разбегаются от такого количества типов! Но здесь можно применить довольно простую схему — для стандартного `typedef T` соответствующим именем атомарного типа будет то же имя, но уже с префиксом `atomic_`: `atomic_T`. Та же схема годится и для встроенных типов, за исключением того, что для типов со знаком (signed) используется сокращение `s`, для типов без знака — `u`, а для `long` — `llong`. В общем, вместо использования альтернативных имен проще будет для типа `T`, с которым нужно работать, указать `std::atomic<T>`.

Стандартные атомарные типы не допускают копирования или присваивания в общепринятом смысле, то есть не имеют конструкторов копирования или операторов копирующего присваивания. И все же в них поддерживаются присваивание из соответствующих встроенных типов и предполагаемое преобразование в эти типы, а также непосредственное использование компонентных функций `load()` и `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`. В них, где это применимо, также поддерживаются составные операторы присваивания: `+=`, `-=`, `*=`, `|=` и т. д. и, кроме этого, поддерживаются целочисленные типы и специализации `std::atomic<>` для указателей `++` и `--`. У операторов также имеются соответствующие поименованные компонентные функции с такими же возможностями: `fetch_add()`, `fetch_or()` и т. д. Возвращаемым значением из операторов присваивания и компонентных функций является либо сохраненное значение (применительно к операторам присваивания), либо значение, имевшееся до операции (применительно к именованным функциям). Это позволяет избежать возможных проблем, возникающих из-за свойства этих операторов присваивания возвращать ссылку на присваиваемый объект. Чтобы можно было получить из этих ссылок сохраненное значение, код должен выполнять отдельное чтение, позволяющее другому потоку изменять значение между присваиванием и чтением и открывающее возможность перехода в состояние гонки.

Но шаблон класса `std::atomic<>` — это не только набор специализаций. В нем имеется первичный шаблон, который может использоваться для создания атомарного варианта типа, определяемого пользователем. Поскольку это обобщенный шаблон класса, операции ограничены функциями `load()`, `store()` (а также присваиванием из типа, определенного пользователем, и преобразованием в этот тип), `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`.

У каждой операции над атомарными типами есть необязательный аргумент упорядочения доступа к памяти, являющийся одним из значений перечисления `std::memory_order`. Этот аргумент используется для указания требуемой семантики при упорядочении доступа к памяти. Перечисление `std::memory_order` содержит шесть возможных значений: `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_consume`, `std::memory_order_acq_rel`, `std::memory_order_release` и `std::memory_order_seq_cst`.

Значения, допустимые для упорядочения доступа к памяти, зависят от категории операции. Если значение упорядочения не указано, используется значение по умолчанию с наивысшей степенью строгости упорядочения `std::memory_order_seq_cst`. Точная семантика параметров упорядочения доступа к памяти рассматривается в разделе 5.3. А пока достаточно будет знать, что операции разбиты на три категории:

- *операции сохранения*, которые могут иметь значения `memory_order_relaxed`, `memory_order_release` или `memory_order_seq_cst`;
- *операции загрузки*, которые могут иметь значения `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire` или `memory_order_seq_cst`;
- *операции чтения — изменения — записи*, которые могут иметь значения `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel` или `memory_order_seq_cst`.

А теперь рассмотрим операции, применимые к каждому стандартному атомарному типу, и начнем с типа `std::atomic_flag`.

5.2.2. Операции над std::atomic_flag

Тип `std::atomic_flag` — это самый простой атомарный тип, представляющий собой булев флаг. Объекты этого типа могут находиться в одном из двух состояний: «установлен» или «сброшен». Этот тип намеренно сделан базовым в предположении, что он будет задействоваться только как строительный блок. Поэтому никогда не ожидалось, что он будет использоваться самостоятельно, за исключением каких-то крайних обстоятельств. И тем не менее он послужит отправной точкой для рассмотрения других атомарных типов, поскольку с его помощью можно продемонстрировать ряд общих принципов, применимых к этим типам.

Объекты типа `std::atomic_flag` *должны* быть проинициализированы значением `ATOMIC_FLAG_INIT`, в результате чего флаг при инициализации приобретает состояние *«сброшен»*. Здесь нет выбора, изначально флаг всегда сброшен:

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

Данное требование применяется независимо от того, где объект объявлен и в какой области видимости находится. Это единственный атомарный тип, требующий специального обхождения для инициализации, а также он является единственным типом, гарантированно свободным от блокировок. Если у объекта `std::atomic_flag` имеется статическая продолжительность хранения, то он гарантированно будет статически проинициализирован, а это означает, что никаких проблем с порядком инициализации не предвидится и его инициализация всегда будет выполняться ко времени первой операции с флагом.

После того как объект флага проинициализирован, с ним можно выполнить только три действия: уничтожить, сбросить или установить и запросить предыдущее значение. Этим действиям соответствуют деструктор, компонентная функция `clear()` и компонентная функция `test_and_set()` соответственно. Для обеих компонентных функций, `clear()` и `test_and_set()`, можно указать упорядочение доступа к памяти. Функция `clear()` выполняет операцию сохранения, поэтому не может иметь семантику `memory_order_acquire` или `memory_order_acq_rel`, а функция `test_and_set()` выполняет операцию чтения — изменения — записи, поэтому к функции можно применять любые теги упорядочения доступа к памяти. По умолчанию, как и для любой другой атомарной операции, для обеих функций задействуется аргумент упорядочения доступа к памяти `memory_order_seq_cst`, например:

```
f.clear(std::memory_order_release); ❶
bool x=f.test_and_set();            ❷
```

Здесь вызов функции `clear()` ❶ явно запрашивает сброс флага с применением семантики освобождения, а вызов функции `test_and_set()` ❷ использует упорядочение доступа к памяти по умолчанию для установки флага и извлечения старого значения.

Из первого объекта `std::atomic_flag` нельзя путем копирования создать еще один такой же объект, и один объект `std::atomic_flag` нельзя присвоить другому объекту. Это не конкретная особенность `std::atomic_flag`, а общее правило для всех атомарных типов. Все операции над атомарными типами определены как атомарные, а при присваивании и создании копии задействуются два объекта. Никакая операция над двумя различными объектами не может быть атомарной. В случае с созданием копии или копирующего присваивания значение сначала должно быть считано из одного объекта, а затем записано в другой объект. Это две разные операции над двумя разными объектами, и их комбинация не может быть атомарной. Поэтому такие операции запрещены.

Ввиду ограниченного набора функций шаблон класса `std::atomic_flag` идеально подходит для использования в качестве мьютекса со спин-блокировкой. Изначально флаг сброшен и мьютекс разблокирован. Чтобы его заблокировать, выполняется цикл с вызовом функции `test_and_set()`, пока старое значение не будет равно `false`, информируя, что этот поток установил значение флага в `true`. Снятие блокировки с мьютекса — это просто сброс флага. Рассмотренная реализация показана в листинге 5.1.

Листинг 5.1. Реализация мьютекса со спин-блокировкой 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);
    }
};
```

Это самый простой мьютекс, но и его вполне достаточно для использования с `std::lock_guard<>` (см. главу 3). По своей конструкции он находится в активном ожидании при выполнении функции `lock()`, поэтому, если предполагается хоть какая-то конкуренция, от него лучше отказаться, но взаимное исключение он все же обеспечивает. Если посмотреть на семантику упорядочения доступа к памяти, станет понятно, как здесь гарантируется необходимое принудительное упорядочение, подходящее к блокировке мьютекса. Этот пример рассматривается в подразделе 5.3.6.

Степень ограниченности `std::atomic_flag` не позволяет воспользоваться им даже для реализации обычного булева флага, поскольку в нем нет простой, не вносящей изменения операции запроса. Для создания флага лучше воспользоваться рассматриваемым далее шаблоном класса `std::atomic<bool>`.

5.2.3. Операции над std::atomic<bool>

Самым простым атомарным целочисленным типом является `std::atomic<bool>`. Это, как и ожидалось, булев флаг с более широкой функциональностью, чем у `std::atomic_flag`. Хотя по-прежнему создания или присваивания копий нет, его можно создать из неатомарного типа `bool`, поэтому изначально у него может быть значение `true` или `false` и значения экземплярам `std::atomic<bool>` можно присваивать также из неатомарного типа `bool`:

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

Относительно оператора присваивания от неатомарного типа `bool` следует отметить, что его поведение выпадает из общего соглашения о возврате ссылки на объект, которому выполняется присваивание: этот оператор возвращает тип `bool` с присвоенным ему значением. Это еще одна модель, общая для всех атомарных типов: поддерживаемые ими операторы присваивания возвращают значения (соответствующего неатомарного типа), а не ссылки. Если бы возвращалась ссылка на

атомарную переменную, то любой код, зависящий от результата присваивания, затем был бы вынужден загрузить значение явным образом, возможно получая при этом результат изменения этой переменной со стороны другого потока. А возвращение результата присваивания в виде неатомарного значения позволяет избежать такой дополнительной загрузки и быть уверенными, что полученное значение является тем же, что и сохраненное.

Вместо использования имеющей довольно ограниченные возможности функции `clear()` шаблона класса `std::atomic_flag` записи, как `true`, так и `false`, выполняются вызовом функции `store()`, хотя указание семантики упорядочения доступа к памяти допускается, как и прежде. Аналогично этому вместо функции `test_and_set()` работает более универсальная компонентная функция `exchange()`, позволяющая заменять сохраненное значение новым, выбранным вами значением и атомарно извлекать исходное значение. В `std::atomic<bool>` также поддерживается обычный, не вносящий изменений запрос значения с подразумеваемым преобразованием в обычный тип `bool` или явным вызовом функции `load()`. Как, наверное, и ожидалось, функция `store()` выполняет операцию сохранения, а функция `load()` — операцию загрузки. Функция `exchange()` выполняет операцию чтения — изменения — 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);
```

Не только функция `exchange()` поддерживает в `std::atomic<bool>` операцию чтения — изменения — записи, в этот шаблон класса введена также операция для сохранения нового значения, если текущее значение равно ожидаемому.

Сохранение (или несохранение) нового значения в зависимости от текущего значения

Новая операция называется сравнением — обменом, она представлена в виде комплексных функций `compare_exchange_weak()` и `compare_exchange_strong()`. Операция сравнения — обмена является краеугольным камнем программирования с применением атомарных типов, при ее выполнении значение атомарной переменной сравнивается с предоставленным ожидаемым значением. Если они равны, сохраняется предоставленное желаемое значение, если не равны, ожидаемое значение обновляется значением атомарной переменной. Возвращаемым типом функций сравнения — обмена является `bool`, если сохранение было выполнено, возвращается `true`, а если нет — `false`. Если выполнено сохранение (по причине равенства значений), операция считается *успешной*, если нет — *неудачной*, в случае успеха возвращается `true`, а в случае неудачи — `false`.

Что касается функции `compare_exchange_weak()`, сохранения может не произойти, даже если исходное значение было равно ожидаемому, в таком случае значение переменной остается прежним, а возвращаемым значением `compare_exchange_weak()` является `false`. Скорее всего, такое может случиться на машинах, где нет единой инструкции сравнения и обмена, если процессор не может гарантировать, что опера-

ция была выполнена атомарно — возможно, из-за того что в середине необходимой последовательность инструкций произошло переключение с потока, выполняющего операцию, на другой поток, запланированный на его место операционной системой, где потоков больше, чем процессоров. Это называется *ложным сбоем*, поскольку причина отказа связана не со значением переменной, а с фактором времени.

Поскольку выполнение функции `compare_exchange_weak()` может завершиться ложным сбоем, обычно ее следует применять в цикле:

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

В данном случае цикл продолжается, пока переменная `expected` имеет значение `false`, показывая, что вызов `compare_exchange_weak()` завершается ложным сбоем.

А вот функция `hand, compare_exchange_strong()` гарантированно возвращает `false`, только если значение было не равно ожидаемому значению `expected`. При этом, если нужно узнать, была ли переменная успешно изменена, или же какой-то из потоков добрался до нее первым, необходимость в показанном ранее цикле отпадает.

Если же переменную нужно изменить, каким бы ни было ее исходное значение (возможно, применив обновленное значение, зависящее от текущего значения), то для этого пригодится обновление переменной `expected`. При каждом прохождении цикла переменная `expected` перегружается, и если в промежутке никакой другой поток не изменяет ее значение, то вызов `compare_exchange_weak()` или `compare_exchange_strong()` при следующем прохождении цикла должен быть успешным. Если вычисление значения, предназначенного для сохранения, не представляет особой сложности, может пригодиться функция `compare_exchange_weak()`, позволяющая избежать двойного цикла на тех платформах, где она может дать ложный сбой (по причине которого в функции `compare_exchange_strong()` используется цикл). А если вычисление значения, предназначенного для сохранения, занимает много времени, возможно, стоит воспользоваться функцией `compare_exchange_strong()`, позволяющей избежать повторного вычисления значения для хранения, когда ожидаемое значение не изменилось. В случае применения `std:: atomic<bool>` это не столь важно, поскольку здесь речь идет только о двух возможных значениях, но для больших атомарных типов это может стать существенным.

Функции сравнения — обмена необычны еще и тем, что могут принимать два параметра упорядочения доступа к памяти. Это позволяет применять разную семантику упорядочения доступа к памяти в случаях успеха и неудачи: возможно, стоило бы в случае успешного вызова задействовать семантику `memory_order_acq_rel`, а в случае неудачи — `memory_order_relaxed`. При неудачном завершении операции сравнения — обмена сохранение не выполняется, поэтому семантика `memory_order_release` или `memory_order_acq_rel` применяться не может. Следовательно, указывать эти значения в качестве семантики упорядочения доступа к памяти на случай неудачного завершения запрещено. Кроме того, на случай неудачи нельзя задавать более строгое упорядочение, чем на случай успеха. Если есть желание задействовать на случай неудачи семантику `memory_order_acquire` или `memory_order_seq_cst`, ее же следует указать и на случай успеха.

Если упорядочение на случай неудачи не указано, то предполагается, что оно такое же, как и на случай удачи, за исключением того, что часть `release` из него удаляется: `memory_order_release` становится `memory_order_relaxed`, а `memory_order_acq_rel` превращается в `memory_order_acquire`. Если упорядочение вообще не указано, то, как обычно, по умолчанию устанавливается семантика `memory_order_seq_cstl`, предоставляющая полное последовательное упорядочение доступа к памяти как для успеха, так и для неудачи. Следующие два вызова функции `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);
```

Выяснение последствий выбора того или иного порядка доступа к памяти отложим до раздела 5.3.

Еще одно отличие `std::atomic<bool>` от `std::atomic_flag` заключается в том, что тип `std::atomic<bool>` может быть не свободен от блокировок, поскольку для обеспечения атомарности операций, возможно, внутри реализации понадобится заблокировать мьютекс. В тех редких случаях, когда для этого есть основания, можно воспользоваться компонентной функцией `is_lock_free()`, чтобы проверить, являются ли операции в отношении `std::atomic<bool>` свободными от блокировки. Это еще одна особенность всех атомарных типов, кроме `std::atomic_flag`.

Следующими самыми простыми атомарными типами являются специализации атомарных указателей `std::atomic<T*>`, которые сейчас и рассмотрим.

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

Атомарной формой указателя на некий тип `T` является `std::atomic<T*>`, что похоже на то, как атомарной формой `bool` является `std::atomic<bool>`. Интерфейс тот же, хотя работа ведется не со значениями типа `bool`, а с указателями на значения соответствующего типа. Аналогично типу `std::atomic<bool>` в нем нет конструкторов копирования и присваивания, хотя он допускает создание и присваивание из подходящих значений указателя. Наряду с обязательной компонентной функцией `is_lock_free()` в `std::atomic<T*>` определены компонентные функции `load()`, `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()` с такой же семантикой, что и у `std::atomic<bool>`, и здесь опять функции получают и возвращают не `bool`, а `T*`-значения.

Шаблон класса `std::atomic<T*>` предоставляет и новые арифметические операции над указателями. Основные операции предоставляются компонентными функциями `fetch_add()` и `fetch_sub()`, выполняющими атомарное сложение и вычитание над сохраненными адресами, операторами `+=` и `-=`, а также пред- и постинкрементными и декрементными операторами `++` и `--`, обеспечивающими удобные оболочки. Операторы работают так, как от них и ожидается на примере встроенных типов: если

x является указателем `std::atomic<Foo*>` на первую запись массива `Foo`-объектов, то выражение `x+=3` изменяет значение, чтобы оно указывало на четвертую запись, и возвращает значение обычного типа `Foo*`, которое также указывает на четвертую запись. Функции `fetch_add()` и `fetch_sub()` немного отличаются от операторов тем, что возвращают исходное значение. Следовательно, вызов `x.fetch_add(3)` приведет к обновлению x, где теперь будет указатель на четвертое значение, но возвращен будет указатель на первое значение массива. Эта операция известна также как операция изменения и прибавления, она относится к атомарным операциям *чтения — изменения — записи* наподобие `exchange()` и `compare_exchange_weak()`/`compare_exchange_strong()`. Как и в случае с другими операциями, возвращается простое `T*`-значение, а не ссылка на объект `std::atomic<T*> object`, поэтому вызывающий код должен выполнить действия на основе того, каким было предыдущее значение:

```
class Foo{};
Foo some_array[5];
std::atomic<Foo*> p(some_array);
Foo* x=p.fetch_add(2);
assert(x==some_array);
assert(p.load()==&some_array[2]);
x=(p-=1);
assert(x==&some_array[1]);
assert(p.load()==&some_array[1]);
```

Прибавление 2 к p и возвращение старого значения

Вычитание 1 из p и возвращение нового значения

Функция в качестве дополнительного аргумента своего вызова позволяет также указывать семантику упорядочения доступа к памяти:

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

Поскольку обе функции, `fetch_add()` и `fetch_sub()`, выполняют операции чтения — изменения — записи, у них могут быть любые теги упорядочения доступа к памяти и они могут участвовать в *последовательности освобождения*. Указать семантику упорядочения операторам невозможно, поскольку для этого нет способа предоставления информации, поэтому с операторами всегда применяется семантика `memory_order_seq_cst`.

Все остальные основные атомарные типы по сути одинаковы: относятся к атомарным целочисленным типам и имеют такой же интерфейс, как и у всех остальных, за исключением того, что они связаны с разными встроенными типами. Поэтому рассмотрим их в группе.

5.2.5. Операции над стандартными атомарными целочисленными типами

Наряду с обычным набором операций (`load()`, `store()`, `exchange()`, `compare_exchange_weak()` и `compare_exchange_strong()`) у таких атомарных целочисленных типов, как `std::atomic<int>` или `std::atomic<unsigned long>`, имеется довольно полный набор доступных операций: `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, `fetch_xor()`, соответствующие этим операциям формы комбинированного

присваивания (+=, -=, &=, |= и ^=), а также пред- и постинкременты и декременты (++x, x++, --x и x--). Это не исчерпывающий набор операций комбинированного присваивания из арсенала обычных целочисленных типов, но он довольно близок к таковому: нет только операторов деления, умножения и сдвига. Поскольку атомарные целочисленные значения обычно используются либо как счетчики, либо как битовые маски, этот недостаток практически незаметен, а дополнительные операции, если понадобится, можно легко выполнить в цикле с помощью функции compare_exchange_weak().

Семантика практически совпадает с той, что используется в функциях fetch_add() и fetch_sub() для std::atomic<T*>: поименованные функции атомарно выполняют свою операцию и возвращают *прежнее* значение, а операторы комбинированного присваивания возвращают *новое* значение. Пред- и постинкремент и декремент работают как обычно: ++x выполняет инкремент переменной и возвращает новое значение, а x++ выполняет инкремент переменной и возвращает прежнее значение. Как и ожидалось, результатом в обоих случаях будет значение связанного целочисленного типа.

После того как были рассмотрены все основные атомарные целочисленные типы, осталось поговорить не о каких-нибудь специализациях, а об обобщенном шаблоне первичного класса std::atomic<>, чем мы далее и займемся.

5.2.6. Шаблон первичного класса std::atomic<>

Наличие первичного шаблона позволяет пользователю создавать в дополнение к стандартным атомарным типам атомарные варианты типа, определенного им самим. Если, к примеру, взять определенный пользователем тип UDT, то std::atomic<UDT> предоставит точно такой же интерфейс, как и у шаблона класса std::atomic<bool>, рассмотренного в подразделе 5.2.3, за исключением того, что вместо bool-параметров и возвращаемых типов, имеющих отношение к сохраненному значению (в отличие от результата «успех — неудача» операций сравнения — обмена), будет использоваться тип UDT. Но просто применять с std::atomic<> любой определенный пользователем тип невозможно, поскольку он должен соответствовать вполне определенным критериям. Чтобы воспользоваться std::atomic<UDT> для некоего определенного пользователем типа UDT, этот тип должен иметь *тривиальный* оператор копирующего присваивания. Это означает, что у типа не должно быть виртуальных функций или виртуальных базовых классов, а копирующий оператор присваивания должен генерироваться компилятором. Но это еще не все: у каждого базового класса или нестатического компонента данных определенного пользователем типа должен быть также обычный оператор копирующего присваивания. Это разрешит компилятору воспользоваться для операций присваивания функцией memcpy() или эквивалентной операцией, поскольку не потребуется выполнять пользовательский код.

И наконец, следует отметить, что операции сравнения — обмена не используют какой-либо оператор сравнения, который может быть определен для UDT, а выполняют поразрядное сравнение, как при работе с функцией memcmp(). Если тип предоставляет операции сравнения, имеющие иную семантику, или у него имеются

биты заполнения, не участвующие в обычных сравнениях, это может привести к сбою операции сравнения — обмена, даже если сравниваемые значения будут равны друг другу.

Чтобы обосновать эти ограничения, нужно вернуться к одной из рекомендаций из главы 3: не передавать указатели и ссылки на защищенные данные за пределы области видимости блокировки путем их передачи в качестве аргументов функций, предоставляемых пользователем. Как правило, компилятор не в состоянии сгенерировать для `std::atomic<UDT>` код без блокировок, поэтому для всех операций ему придется воспользоваться внутренней блокировкой. Если бы предоставленные пользователем операторы копирующего присваивания или сравнения были разрешены, это в нарушение рекомендации потребовало бы передать функции, предоставляемой пользователем, ссылки на защищенные данные в качестве аргумента. Кроме того, библиотеке ничто не мешает использовать одну блокировку для всех нуждающихся в ней атомарных операций, а то, что при удержании этой блокировки разрешается вызывать функции, предоставленные пользователем, может привести к взаимной блокировке или к блокировке других потоков, поскольку операция сравнения занимает много времени. И наконец, эти ограничения увеличивают вероятность того, что компилятор сможет для `std::atomic<UDT>` напрямую воспользоваться атомарными инструкциями (и сделать конкретный экземпляр свободным от блокировки), поскольку он может рассматривать определенный пользователем тип в качестве набора обычных байтов.

Следует заметить: несмотря на возможность применения шаблонов `std::atomic<float>` или `std::atomic<double>`, определяемую тем, что встроенные типы с плавающей точкой удовлетворяют критериям использования с функциями `memcpy` и `memcmp`, их поведение в случае применения функции `compare_exchange_strong` может оказаться неожиданным (как уже было рассмотрено, выполнение `compare_exchange_weak` в силу произвольных внутренних причин всегда может завершиться неудачей). Операция может завершиться неудачей, даже когда прежнее сохраненное значение равно сравниваемому значению, если сохраненное значение имело другое внутреннее представление. Следует также обратить внимание на отсутствие атомарных арифметических операций над значениями с плавающей точкой. Аналогичное поведение `compare_exchange_strong` будет наблюдаться и при использовании `std::atomic<>` с типом, определенным пользователем, для которого определен оператор сравнения на равенство, отличный от сравнения с помощью `memcmp`. Операция может завершиться неудачей, потому что равные по сути значения имеют различное представление.

Если размер вашего типа `UDT` не превышает размера `int` или `void*`, то на большинстве платформ для типа `std::atomic<UDT>` можно сгенерировать код, содержащий только атомарные команды. На некоторых платформах также можно сгенерировать подобный код, когда размер пользовательского типа в два раза превышает размер `int` или `void*`. К таким платформам обычно относятся те, которые поддерживают так называемую *инструкцию сравнения и обмена двойных слов* (double-word-compare-and-swap, DWCAS), которая соответствует функциям `compare_exchange_xxx`. Такая поддержка, как будет показано в главе 7, может пригодиться при написании кода без блокировок.

Существующие ограничения означают, к примеру, невозможность создания `std::atomic<std::vector<int>>`, поскольку там есть необычный конструктор копирования и оператор копирующего присваивания. Но можно создать экземпляр `std::atomic<>` с классами, содержащими счетчики, или флаги, или указатели, или даже массивы простых элементов. Суть проблемы не в этом, поскольку чем сложнее структура данных, тем вероятнее, что над ними не потребуется выполнять операции сложнее простого присваивания и сравнения. Но в таком случае лучше воспользоваться классом `std::mutex`, гарантирующим, что в соответствии с положениями главы 3 данные для нужных операций должным образом защищены.

Как уже упоминалось, при создании экземпляра определенного пользователем типа `T` интерфейс `std::atomic<T>` ограничен набором операций, доступных для `std::atomic<bool>`: `load()`, `store()`, `exchange()`, `compare_exchange_weak()`, `compare_exchange_strong()`, а также присваиванием из экземпляра типа `T` и преобразованием в этот тип.

Операции, доступные для выполнения над каждым атомарным типом, сведены в табл. 5.3.

Таблица 5.3. Операции, доступные для выполнения над атомарными типами

Операция	atomic_flag	atomic <bool>	atomic <T*>	atomic <целочисленный тип>	atomic <другой тип>
test_and_set	Да	—	—	—	—
clear	Да	—	—	—	—
is_lock_free	—	Да	Да	Да	Да
load	—	Да	Да	Да	Да
store	—	Да	Да	Да	Да
exchange	—	Да	Да	Да	Да
compare_exchange_weak, compare_exchange_strong	—	Да	Да	Да	Да
fetch_add, +=	—	—	Да	Да	—
fetch_sub, –=	—	—	Да	Да	—
fetch_or, |=	—	—	—	Да	—
fetch_and, &=	—	—	—	Да	—
fetch_xor, ^=	—	—	—	Да	—
++, ––	—	—	Да	Да	—

5.2.7. Свободные функции для атомарных операций

До сих пор давались описания тех операций над атомарными типами, выполнение которых обеспечивалось компонентными функциями. Но для всех операций над различными атомарными типами есть еще и эквивалентные им некомпонентные функции. В основном эти функции называются так же, как и соответствующие им

компонентные, но с префиксом `atomic_` (например, `std::atomic_load()`). Затем эти функции переопределяются для каждого из атомарных типов. Там, где есть возможность задать тег упорядочения доступа к памяти, используется два вида: один без тега, другой — с суффиксом `_explicit` и дополнительным параметром или параметрами для тега или тегов упорядочения доступа к памяти (например, `std::atomic_store(&atomic_var,new_value)` в сравнении с `std::atomic_store_explicit(&atomic_var,new_value,std::memory_order_release)`. В то же время атомарный объект, на который ссылаются компонентные функции, подразумевается, все свободные функции получают в качестве первого параметра указатель на него.

Например, функция `std::atomic_is_lock_free()` бывает только одного вида (хотя и переопределяется для каждого типа), а функция `std::atomic_is_lock_free(&a)` возвращает такое же значение, как и вызов функции `a.is_lock_free()` для объекта атомарного типа `a`. Аналогично этому `std::atomic_load(&a)` — то же самое, что и `a.load()`, но эквивалентом `a.load(std::memory_order_acquire)` является `std::atomic_load_explicit(&a, std::memory_order_acquire`.

Свободные функции сконструированы с прицелом на совместимость с языком C, поэтому в них всегда используются не ссылки, а указатели. Например, первым параметром компонентных функций `compare_exchange_weak()` и `compare_exchange_strong()` (как и ожидалось) является ссылка, тогда как вторым параметром `std::atomic_compare_exchange_weak()` (первый — указатель на объект) служит указатель. Для функции `std::atomic_compare_exchange_weak_explicit()`, кроме этого, требуется указывать тег упорядочения доступа к памяти на случай как успеха, так и неудачи, тогда как у компонентных функций сравнения — обмена для обоих исходов имеются единая форма упорядочения доступа к памяти (а по умолчанию используется `std::memory_order_seq_cst`) и переопределение, получающее отдельные установки по упорядочению доступа к памяти на случай успеха и неудачи.

Операции над `std::atomic_flag` нарушают сложившуюся тенденцию, поскольку в их именах фигурирует слово `flag`: `std::atomic_flag_test_and_set()`, `std::atomic_flag_clear()`. А в именах дополнительных вариантов, где указывается упорядочение доступа к памяти, опять есть суффикс `_explicit`: `std::atomic_flag_test_and_set_explicit()` и `std::atomic_flag_clear_explicit()`.

Стандартная библиотека C++ также предоставляет свободные функции для доступа к экземплярам `std::shared_ptr<>` в атомарной манере. Тем самым нарушается принцип, согласно которому поддержка атомарных операций имеется только в атомарных типах, поскольку `std::shared_ptr<>` совершенно точно к атомарному типу не относится (обращение к одному и тому же объекту `std::shared_ptr<T>` из нескольких потоков без использования атомарных функций доступа со всех потоков или иной подходящей внешней синхронизации приводит к состоянию гонки за данными и неопределенному поведению). Но Комитет по стандартизации C++ посчитал, что предоставление этих дополнительных функций сыграет весьма существенную роль. Доступны следующие атомарные операции: загрузки, сохранения, обмена и сравнения — обмена, предоставляемые в виде переопределений аналогичных

операций над стандартными атомарными типами, получающими в виде первого аргумента `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 в случае атомарных операций над другими типами, предоставляются `_explicit`-варианты, позволяющие указывать нужное упорядочение доступа к памяти, а функция `std::atomic_is_lock_free()` может использоваться для проверки факта применения реализацией блокировок для обеспечения атомарности.

Спецификацией Concurrency TS также предусмотрено предоставление класса `std::experimental::atomic_shared_ptr<T>`, являющегося атомарным типом. Чтобы им воспользоваться, нужно включить в программу заголовок `<experimental/atomic>`. Этот класс обеспечивает такой же набор операций, что и `std::atomic<UDT>`: загрузки, сохранения, обмена и сравнения — обмена. Данный класс предоставляется в виде отдельного типа, поскольку это позволяет воспользоваться реализацией без блокировок, которая не требует дополнительных затрат на простые экземпляры `std::shared_ptr`. Но, как и в случае с шаблоном `std::atomic`, все равно требуется проверить, свободен ли этот класс от блокировок на вашей платформе, что можно сделать с помощью компонентной функции `is_lock_free`. Даже если он не свободен от блокировок, все же рекомендуется предпочесть `std::experimental::atomic_shared_ptr`, а не атомарные свободные функции, применяемые в отношении обычного класса `std::shared_ptr`, поскольку его использование в вашем коде делает его понятнее и гарантирует, что все обращения атомарны, исключая при этом вероятность гонки за данными из-за отказа от атомарных свободных функций. Как и в других случаях применения атомарных типов и операций, если все это служит для потенциального увеличения скорости выполнения программы, важно выполнить профилирование и сравнительный анализ с использованием альтернативных механизмов синхронизации.

Как отмечалось во введении, стандартные атомарные типы позволяют не только исключить неопределенность поведения, связанную с состоянием гонки за данными, но и позволить пользователю задать порядок выполнения операций в потоках. Принудительное упорядочение положено в основу таких средств защиты данных и синхронизации, как `std::mutex` и `std::future<>`. Помня об этом, перейдем к самой сути данной главы — подробностям конкурентных аспектов модели памяти и способам использования атомарных операций для синхронизации доступа к данным и задания принудительного упорядочения.

5.3. Синхронизация операций и принудительное упорядочение

Предположим, что имеются два потока, один из которых наполняет структуру данными, предназначенными для чтения вторым потоком. Чтобы избежать проблемного состояния гонки, первый поток устанавливает флаг, показывающий, что данные готовы. Пока флаг не установлен, второй поток не считывает данные. Этот сценарий показан в листинге 5.2.

Листинг 5.2. Чтение и запись переменных из разных потоков

```
#include <vector>
#include <atomic>
#include <iostream>
std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread()
{
    while(!data_ready.load())      ←❶
    {
        std::this_thread::sleep(std::chrono::milliseconds(1));
    }
    std::cout<<"The answer="<<data[0]<<"\n"; ←❷
}
void writer_thread()
{                                  ❸
    data.push_back(42);    ←┘
    data_ready=true;       ←❹
}
```

Абстрагируясь от неэффективности цикла, ожидающего готовности данных ❶, следует признать его необходимость для выполнения данной работы, поскольку в противном случае совместное использование данных этими потоками становится непрактичным — тогда каждый элемент данных должен стать атомарным. Как известно, доступ к одним и тем же данным с выполнением неатомарного чтения ❷ и записи ❸ без принудительного упорядочения приводит к неопределенному поведению, поэтому, чтобы код был работоспособным, нужно где-то задать принудительное упорядочение.

Принудительное упорядочение задается операциями `data_ready` над переменной `std::atomic<bool>`. Эти операции обеспечивают необходимое упорядочение благодаря отношениям модели памяти *«происходит до»* и *«синхронизируется с»*. Запись данных ❸ выполняется до записи во флаг `data_ready` ❹, а чтение флага ❶ — до чтения данных ❷. Если значение, считываемое из `data_ready` ❶, равно `true`, запись синхронизируется с чтением, создавая отношение «происходит до». Поскольку отношение «происходит до» является переходным, запись в данные ❸ происходит до записи во флаг ❹, а запись во флаг происходит до считывания из флага значения `true` ❶, которое, в свою очередь, выполняется до чтения данных ❷. Получается

принудительное упорядочение: данные записываются до их чтения, и все проходит нормально. Важные взаимоотношения двух потоков «происходит до» показаны на рис. 5.2. В читающий поток добавлены две итерации цикла `while`.

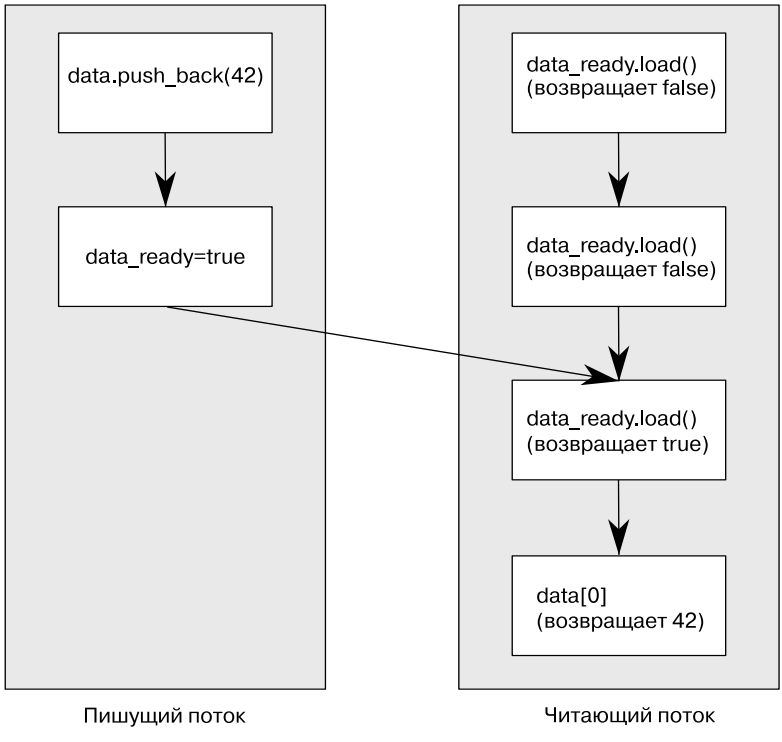


Рис. 5.2. Задание принудительного упорядочения между неатомарными операциями с помощью атомарных операций

Похоже, что все понятно на интуитивном уровне: операция, записывающая значение, выполняется до операции, считывающей его. Для атомарных операций по умолчанию все само собой разумеется (поэтому они и называются операциями по умолчанию), но все же требуется сказать: у атомарных операций имеются и другие варианты задания требований по упорядочению, вскоре мы их рассмотрим.

Теперь, когда отношения «происходит до» и «синхронизируется с» показаны на практике, настало время узнать, что же они означают. Начнем с отношения «синхронизируется с».

5.3.1. Отношение «синхронизируется с»

Отношение *«синхронизируется с»* можно получить только между операциями над атомарными типами. Операции над структурами данных, например блокирование мьютекса, могут реализовать это отношение, если структура данных содержит атомарные типы и внутри операции над этой структурой данных выполняются соот-

ветствующие атомарные операции. Но в основном оно исходит только от операций над атомарными типами.

Основной смысл таков: соответственно помеченная атомарная операция записи `W`, выполняемая над переменной `x`, синхронизируется соответственно помеченной атомарной операцией чтения, выполняемой над переменной `x` и считывающей значение, сохраненное либо записью `W`, либо последующей атомарной операцией записи, выполняемой над `x` тем же самым потоком, который выполнял исходную запись `W`, или последовательностью атомарных операций чтения — изменения — записи, выполняемых над `x` (например, `fetch_add()` или `compare_exchange_weak()`) в любом потоке, при условии, что значение, считываемое первым потоком последовательности, является значением, записанным операцией `W` (см. подраздел 5.3.4).

Отвлечемся пока от понятия «соответственно помеченная», поскольку все операции над атомарными типами соответственно помечены изначально. Смысл сказанного соответствует возможным ожиданиям: если поток А сохраняет значение, а поток Б считывает его, значит, между сохранением значения в потоке А и его загрузкой в потоке Б устанавливается отношение «синхронизируется с», как в листинге 5.2. Эта ситуация показана на рис. 5.2.

Надеюсь, вы уже догадались, что все тонкости данного обстоятельства заключены в понятии «соответственно помеченная». Модель памяти C++ допускает применение к операциям над атомарными типами различных ограничений порядка их выполнения (здесь имеется в виду установка меток). Разные варианты задания порядка доступа к памяти и их связь с отношением вида «синхронизируется с» рассматриваются в подразделе 5.3.3. Но сначала вернемся к отношению «происходит до».

5.3.2. Отношение «происходит до»

Отношения *«происходит до»* и *«происходит строго до»* являются основными строительными блоками задания порядка выполнения операций в программе. Они определяют, каким операциям будут видны результаты выполнения других операций. Когда поток всего один, все очень просто: если одна операция находится в последовательности выполнения перед другой, то она выполняется также до выполнения другой операции и «происходит строго до» него. Значит, если одна операция (А) появляется в инструкции исходного кода перед другой (Б), то А «происходит до» Б и А «происходит строго до» Б.

Именно это и было показано в листинге 5.2: запись в переменную `data` ❸ происходит до записи в переменную `data_ready` ❹. Если операции выполняются в одной и той же инструкции, то, как правило, между ними нет отношения «происходит до», потому что они не упорядочены. Иначе говоря, порядок не определен. Известно, что программа в листинге 5.3 выведет либо `1,2`, либо `2,1`, но что именно, не определено, потому что порядок двух вызовов `get_num()` не указан.

Порой операции в рамках одной инструкции выстроены в последовательность, например, когда используется встроенный оператор в виде запятой или результат одного выражения становится аргументом другого. Но чаще всего операции в рамках одной инструкции не выстроены в последовательность и между ними нет отношения «находится в последовательности до» (а стало быть, «происходит до»). Все операции в одной инструкции происходят до всех операций в следующей инструкции.

Листинг 5.3. Порядок вычисления аргументов вызова функции не определен

```
#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());  <-- Вызовы get_num() не упорядочены
}
```

Это всего лишь подтверждение привычных правил однопоточной последовательности, так что же тут нового? Новым является взаимодействие между потоками: если операция А в одном потоке выполняется до операции Б в другом потоке, значит, операция А происходит до операции Б. Пользы вроде бы немного: просто добавилось новое отношение («происходит до» среди потоков), но при написании многопоточного кода это отношение трудно переоценить.

На начальном уровне отношение «происходит до» среди потоков представляется довольно простым и полагается на отношение «синхронизируется с», представленное в подразделе 5.3.1: если операция А в одном потоке синхронизируется с операцией Б в другом потоке, то А между потоками происходит до Б. Это отношение транзитивно: если А между потоками происходит до Б, а Б между потоками происходит до В, то А между потоками происходит до В. Это также было показано в листинге 5.2.

Отношение «происходит до» среди потоков сочетается с отношением «находится в последовательности до»: если операция А находится в последовательности до операции Б, а операция Б между потоками происходит до операции В, то А между потоками происходит до В. Аналогично этому если А синхронизируется с Б, а Б находится в последовательности до В, то А между потоками происходит до В. В совокупности эти два утверждения означают, что если выполнить серию изменений в данные в одном потоке, то, чтобы данные были видны последующим операциям в потоке, выполняющим операцию В, понадобится всего одно отношение «синхронизируется с».

Отношение «происходит строго до» немного отличается от «происходит до», но в большинстве случаев дает тот же самый эффект. К нему применяются те же два ранее рассмотренных правила: если операция А «синхронизируется с» операцией Б или «находится в последовательности до» операции Б, то А «происходит строго до» Б. Применимо также транзитивное упорядочение: если А «происходит строго до» Б, а Б «происходит строго до» В, то А «происходит строго до» В. Разница состоит в том, что операции, помеченные как `memory_order_consume` (см. подраздел 5.3.3), участвуют в отношениях «происходит до среди потоков» (а следовательно, и в отношениях «происходит до»), но не участвуют в отношениях «происходит строго до». Поскольку в абсолютной массе создаваемого кода `memory_order_consume` не используется, это

отличие вряд ли существенно повлияет на вашу практическую деятельность. Далее в этой книге для краткости будет применяться только отношение «происходит до».

Именно эти весьма важные правила позволяют принудительно задать порядок операций среди потоков и добиться, чтобы все в листинге 5.2 работало нормально. Как вскоре будет показано, имеется также ряд дополнительных нюансов, связанных с зависимостью данных. Чтобы в них разобраться, придется рассмотреть признаки упорядочения доступа к памяти, используемые для атомарных операций, и их связь с отношением «синхронизируется с».

5.3.3. Упорядочение доступа к памяти для атомарных операций

К операциям над атомарными типами можно применять шесть вариантов упорядочения доступа к памяти: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel` и `memory_order_seq_cst`. Если для конкретной операции не указано что-либо иное, для всех операций над атомарными типами применяется вариант упорядочения доступа к памяти `memory_order_seq_cst` — самый жесткий из доступных. Хотя вариантов упорядочения шесть, представляют они всего три модели: *последовательно согласованное* упорядочение (`memory_order_seq_cst`), упорядочение *методом захвата — освобождения* (`memory_order_consume`, `memory_order_acquire`, `memory_order_release` и `memory_order_acq_rel`) и *нестрогое* упорядочение (`memory_order_relaxed`).

При выполнении на процессорах разных архитектур у различных моделей упорядочения доступа к памяти получаются разные издержки. Например, системам на основе архитектур с высокой степенью контроля над видимостью операций теми процессорами, которые не вносили изменения, для последовательно согласованного упорядочения может понадобиться больше инструкций синхронизации, чем для упорядочения на основе захвата — освобождения или нестрогого упорядочения, а для упорядочения на основе захвата — освобождения — больше инструкций, чем для нестрогого упорядочения. Если в таких системах много процессоров, эти дополнительные инструкции синхронизации могут отнять довольно много времени, снизив общую производительность системы. Что же касается центральных процессоров с архитектурами x86 или x86-64 (например, вполне обычных для настольных компьютеров процессоров Intel и AMD), то им для упорядочения на основе захвата — освобождения не требуется никаких дополнительных инструкций, кроме тех, что необходимы для обеспечения атомарности. И даже последовательно согласованному упорядочению никаких особых режимов для операций загрузки не требуется, хотя сохранение все же не обходится без небольших дополнительных издержек.

Наличие различных моделей упорядочения доступа к памяти позволяет специалистам воспользоваться преимуществами повышения производительности за счет более подробной детализации отношений упорядочения там, где это выгодно, наряду с возможностью использования в более простых ситуациях последовательно согласованного упорядочения по умолчанию, разобраться в котором гораздо проще, чем в остальных.

Чтобы выбрать конкретную модель упорядочения или разобраться в отношениях упорядочения в коде, использующем различные модели, важно знать, как выбор той или иной модели повлияет на поведение программы. Поэтому давайте посмотрим на последствия каждого варианта выбора для упорядочения операций и отношения «синхронизируется с».

Последовательно согласованное упорядочение

Упорядочение по умолчанию называется *последовательно согласованным*, так как оно означает, что поведение программы согласуется с простой последовательной картиной мира. Если все операции над экземплярами атомарного типа последовательно согласованы, поведение многопоточной программы будет таким же, как и при выполнении всех этих операций в конкретной последовательности в одном потоке. Это, конечно же, самый простой для понимания вариант упорядочения доступа к памяти, поэтому он используется по умолчанию: все потоки должны видеть один и тот же порядок выполнения операций. При этом рассуждать о поведении программы, созданной с применением атомарных переменных, становится намного легче. Можно выписать все возможные последовательности операций, выполняемых различными потоками, отбросить несогласованные и проверить, что в остальном код ведет себя вполне ожидаемо. Кроме того, это означает, что операции нельзя переупорядочить, то есть, если в одном потоке одна операция выполняется прежде другой, этот порядок должен быть виден остальным потокам.

С точки зрения синхронизации последовательно согласованное сохранение синхронизируется с последовательно согласованной загрузкой той же переменной, в которой читается сохраненное значение. Тем самым одно ограничение упорядочения предъявляется к работе сразу двух (и более) потоков, но последовательная согласованность способна на большее. Любые последовательно согласованные атомарные операции, выполненные после этой загрузки, должны появляться после сохранения в поле зрения других потоков, действующих в системе, использующей последовательно согласованные атомарные операции. Это ограничение упорядочения показано в действии в примере, приведенном в листинге 5.4. Оно не распространяется на потоки, в которых задействуются атомарные операции с нестрогим упорядочением доступа к памяти, которые по-прежнему могут видеть операции в другом порядке, поэтому, чтобы получить выгоду от последовательно согласованных операций, их нужно использовать во всех потоках.

Но за простоту понимания приходится платить. На машине с несколькими процессорами слабое упорядочение может привести к заметному снижению производительности, из-за того что должна поддерживаться согласованность последовательности операций между процессорами, зачастую требующая всесторонних (и весьма затратных!) синхронизационных процедур. И все же некоторые процессорные архитектуры, например широко распространенные x86 и x86-64, обеспечивают последовательную согласованность с относительно низкими издержками, поэтому, если вас беспокоит снижение производительности из-за применения последовательно согласованной упорядоченности, изучите документацию по процессорной архитектуре конкретной системы.

В листинге 5.4 показан пример использования последовательной согласованности. Для загрузки и сохранения x и y явно указан признак memory_order_seq_cst, хотя в данном случае он мог быть опущен, поскольку задействуется по умолчанию.

Листинг 5.4. Последовательная согласованность, подразумевающая полную 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);  ←❶
}
void write_y()
{
    y.store(true,std::memory_order_seq_cst);  ←❷
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))     ←❸
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))     ←❹
        ++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); ←❺
}
```

Утверждение assert ❺ никогда не сработает, поскольку первым должно произойти либо сохранение, относящееся к x ❶, либо сохранение, относящееся к y ❷, даже если не указано, какое именно. Если загрузка, относящаяся к y в функции read_x_then_y ❸, возвращает false, сохранение, относящееся к x, должно произойти до сохранения, относящегося к y. В таком случае загрузка, относящаяся к x в функции read_y_then_x

❹, должна возвратить `true`, поскольку цикл `while` гарантирует, что в этом месте кода `y` имеет значение `true`. Так как семантика `memory_order_seq_cst` требует единого полного упорядочения всех операций, помеченных признаком `memory_order_seq_cst`, между загрузкой, относящейся к `y`, которая возвращает `false` ❸, и сохранением, относящимся к `y` ❶, существует подразумеваемое отношение упорядоченности. Чтобы получилось единое полное упорядочение, если один поток сначала видит `x==true`, а затем `y==false`, предполагается, что в этом полном упорядочении сохранение, относящееся к `x`, происходит до сохранения, относящегося к `y`.

Поскольку все симметрично, события могут развиваться и в обратном порядке, при котором загрузка, относящаяся к `x` ❹, возвращает `false`, заставляя загрузку, относящуюся к `y` ❸, возвратить `true`. В обоих случаях `z` равно `1`. Обе загрузки могут возвратить `true`, и тогда `z` будет равно `2`, но ни при каких обстоятельствах `z` не равно `0`.

Операции и отношения «происходит до» для случая, когда `read_x_then_y` видит `x` как `true`, а `y` как `false`, показаны на рис. 5.3. Пунктирная линия, проведенная от загрузки, относящейся к `y` в функции `read_x_then_y`, к сохранению, относящемуся к `y` в функции `write_y`, показывает подразумеваемое отношение упорядочения, необходимое для обеспечения последовательной согласованности: чтобы получить показанные здесь результаты, загрузка в глобальном порядке операций с пометкой `memory_order_seq_cst` должна произойти до сохранения.

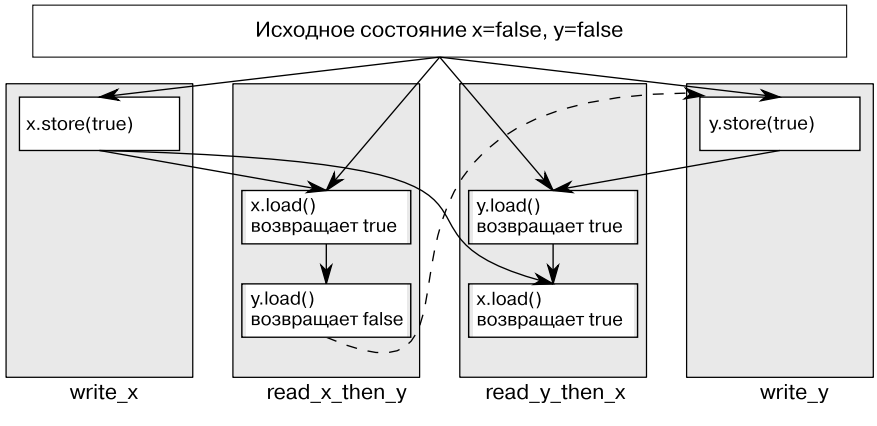


Рис. 5.3. Последовательная согласованность и отношение «происходит до»

Последовательная согласованность относится к наиболее простым и интуитивно понятным вариантам упорядочения доступа к памяти, но это и самый затратный вариант, поскольку требует глобальной синхронизации между всеми потоками. В многопроцессорной системе, чтобы обеспечить ее, может потребоваться всесторонний и весьма затратный по времени обмен данными между процессорами.

Чтобы избежать высоких затрат на синхронизацию, нужно покинуть мир последовательной согласованности и рассмотреть возможность использования других вариантов упорядочения доступа к памяти.

Непоследовательно согласованное упорядочение доступа к памяти

Стоит только покинуть уютный мир последовательной согласованности, как все начнет усложняться. Самой большой проблемой, с которой придется справляться, станет, наверное, *отсутствие прежнего единого глобального порядка событий*. Это означает, что разные потоки могут видеть разные представления одних и тех же операций и с любой воображаемой моделью аккуратно чередующихся операций из разных потоков следует распрощаться. Нужно учитывать не только то, что все происходит фактически одновременно, но и то, что *потоки не должны согласовывать порядок наступления событий*. Чтобы создать (или даже понять) любой код, использующий упорядочение доступа к памяти, отличное от применяемого по умолчанию варианта `memory_order_seq_cst`, чрезвычайно важно как следует во всем разобраться. Дело даже не в том, что компилятор может изменить порядок следования инструкций. Даже если в потоках запускается один и тот же фрагмент кода, у них могут быть разночтения в порядке наступления событий из-за операций в других потоках при отсутствии явно заданных ограничений упорядочения, из-за того, что в кэш-памяти и во внутренних буферах разных центральных процессоров могут содержаться разные значения для одной и той же области памяти. И еще раз подчеркну очень важное обстоятельство: *потоки не должны согласовывать порядок наступления событий*.

Нужно не только отбросить в сторону воображаемые модели, основанные на чередующихся операциях, но и распрощаться с воображаемыми моделями, основанными на идее изменения порядка следования инструкций компилятором или процессором. *В отсутствие других ограничений упорядочения единственным требованием остается согласие всех потоков на порядок изменения каждой отдельно взятой переменной*. Операции над различными переменными в разных потоках могут отображаться в разном порядке при условии, что наблюдаемые значения согласуются с любыми дополнительно наложенными ограничениями упорядочения.

Продемонстрировать это нагляднее всего можно, окончательно выйдя за рамки последовательной согласованности и воспользовавшись для всех операций вариантом упорядочения `memory_order_relaxed`. Разобравшись с ним, можно будет вернуться к упорядочению на основе захвата — освобождения, позволяющему избирательно вводить отношения упорядочения между операциями и снова стать на путь здравомыслия.

Нестрогое упорядочение

Операции над атомарными типами, выполняемые при нестрогом упорядочении, принимают участие в отношениях «синхронизируется с». Операции над одной и той же переменной в одном потоке по-прежнему связаны отношениями «происходит до», но на упорядочение доступа к памяти относительно других потоков не накладывается практически никаких ограничений. Единственное требование заключается в том, чтобы обращения к отдельно взятой атомарной переменной из одного и того же потока не могли быть переупорядочены: если данный поток

уже увидел конкретное значение атомарной переменной, то последующее чтение этим же потоком не может извлечь предыдущее значение переменной. В отсутствие дополнительной синхронизации порядок изменения каждой переменной — это единственное, за что несут совместную ответственность потоки, использующие вариант упорядочения `memory_order_relaxed`.

Чтобы продемонстрировать, до какой степени нестроги могут быть ваши нестрогие операции, достаточно будет, судя по листингу 5.5, всего двух потоков.

Листинг 5.5. К нестрогим операциям предъявляется минимум требований упорядочения

```
#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); ←❶
    y.store(true,std::memory_order_relaxed); ←❷
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed)); ←❸
    if(x.load(std::memory_order_relaxed))     ←❹
        ++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); ←❺
}
```

На этот раз утверждение `assert` ❺ *может* сработать, поскольку загрузка, относящаяся к `x` ❹, может прочитать `false`, даже если загрузка, относящаяся к `y` ❸, прочитает `true`, а сохранение, относящееся к `x` ❶, произойдет до сохранения, относящегося к `y` ❷. `x` и `y` — разные переменные, поэтому никаких гарантий упорядочения относительно видимости значений, получающихся в результате операций над каждой из них, просто нет.

Нестрогие операции над различными переменными можно свободно переупорядочивать при условии, что они подчиняются любым ограничивающим их отношениями «происходит до» (например, внутри того же самого потока). В них не вводятся отношения «синхронизируется с». Отношения «происходит до» из листинга 5.5, а также возможные последствия их применения показаны на рис. 5.4. Но даже при том, что между сохранениями и между загрузками есть отношения «происходит

до», между любым сохранением и любой загрузкой таких отношений нет, поэтому загрузки могут видеть сохранения в случайном порядке.

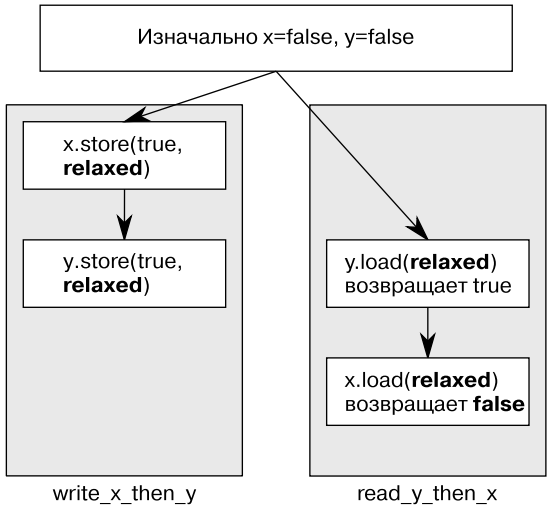


Рис. 5.4. Нестрогие атомарные операции и отношения «происходит до»

Рассмотрим пример посложнее (листинг 5.6), в котором используются три переменные и пять потоков.

Листинг 5.6. Нестрогие операции в нескольких потоках

```
#include <thread>
#include <atomic>
#include <iostream>
std::atomic<int> x(0),y(0),z(0);
std::atomic<bool> go(false);
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)
        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);
```

❶ ❷ ❸ Цикл в ожидании сигнала

```
        values[i].z=z.load(std::memory_order_relaxed);
        var_to_inc->store(i+1,std::memory_order_relaxed); ❹
        std::this_thread::yield();
    }
}
void read_vals(read_values* values)
{
    while(!go)                                    ❺ Цикл в ожидании 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 за запуск выполнения
    t5.join();                                    основного цикла ❻
    t4.join();
    t3.join();
    t2.join();
    t1.join();
    print(values1);                               ❼ Вывод на печать итоговых значений
    print(values2);
    print(values3);
    print(values4);
    print(values5);
}
```

Программа не отличается особой сложностью. В ней имеются три совместно используемые глобальные атомарные переменные ❶ и пять потоков. Каждый поток выполняет десять прохождений цикла, считывая значения трех атомарных переменных с использованием упорядочения `memory_order_relaxed`, и сохраняет их в массиве. Каждый из трех потоков обновляет одну из атомарных переменных при каждом прохождении цикла ❹, а другие два потока выполняют чтение. После

объединения всех потоков выводятся на печать значения из массивов, сохраненных каждым потоком ❼.

Атомарная переменная `go` ❷ используется для обеспечения как можно более одновременного запуска цикла во всех потоках. Запуск потока влечет за собой немалые издержки, и без явно указанной задержки первый поток мог бы завершить свою работу еще до запуска последнего потока. Каждый поток, прежде чем войти в основной цикл (❸ и ❺), ждет, пока переменная `go` не получит значение `true`, а для переменной `go` значение `true` устанавливается только после запуска всех потоков ❻.

Один из возможных вариантов вывода из этой программы выглядит так:

```
(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 строчки относятся к потокам, выполняющим обновления, а последние две — к потокам, выполняющим чтение. Каждая структура из трех элементов является набором значений переменных `x`, `y` и `z` в указанном порядке из одного прохода цикла. На основе выведенных данных можно сказать следующее.

- Первый набор значений показывает, что с каждой тройкой значение `x` увеличивается на единицу, во втором наборе на единицу увеличивается `y`, а в третьем — `z`.
- Элементы `x` каждой тройки увеличиваются только в пределах заданного набора, что справедливо также для элементов `y` и `z`, но приращения неравномерны и относительный порядок во всех потоках разный.
- Поток номер 3 не видит никаких обновлений `x` или `y`, он видит только обновления, происходящие с `z`. Но это не мешает другим потокам наряду с тем, что они видят обновления `x` и `y`, видеть также обновления `z`.

Это вполне допустимый, но далеко не единственно возможный результат выполнения нестрогих операций. Допустимым будет считаться любой набор, состоящий из трех переменных, в каждой из которых поочередно хранятся значения от 0 до 10, у которого также имеется поток, увеличивающий значение заданной переменной на единицу и выводящий для нее значения от 0 до 9.

Осмысление сути нестрогого упорядочения

Для того чтобы понять, как все это работает, представим себе каждую переменную в виде человека, сидящего в кабинке с блокнотом, в котором имеется список значений. Можно позвонить и попросить его назвать существующее значение или записать новое. Если попросить записать новое значение, он занесет его в конец списка. Если попросить назвать существующее значение, он прочитает число из списка.

При первом обращении с просьбой назвать значение человек может дать любое значение из списка, имеющееся в блокноте на данный момент. Если затем попросить назвать еще одно значение, он может дать вам или то же самое значение, или находящееся ниже по списку. Но никогда не назовет значение, находящееся выше по списку. Если попросить его записать значение, а затем назвать значение, он прочитает либо число, которое ему было названо для записи, либо число, находящееся ниже по списку.

Представим, что его список начинается со значений 5, 10, 23, 3, 1 и 2. В ответ на просьбу назвать значение можно получить любое из этих чисел. Если названо число 10, то в следующий раз после такой же просьбы может быть снова названо число 10 или любое из чисел, стоящих в списке позже него, но не 5. Если звонить этому человеку пять раз, он может назвать, к примеру, 10, 10, 1, 2, 2. Если попросить его записать число 42, он добавит его к концу списка. Если снова попросить его назвать число, он неизменно будет называть 42, пока в его списке не появится другое число и ему не захочется его назвать.

Теперь представим, что телефонный номер этого человека есть и у вашего друга Карла, который тоже может ему позвонить и попросить либо назвать число, либо записать новое число, и к общению с Карлом будут применяться те же правила, что и к общению с вами. У человека только один телефон, поэтому одновременно он может общаться только с одним из вас, так что список в его блокноте заполняется строго последовательным образом. Но то, что вы заставили его записать новое число, еще не значит, что он должен сообщить его Карлу, и наоборот. Если Карл попросил его назвать число и получил в ответ 23, то ваша просьба записать число 42 не означает, что оно будет названо Карлу в следующий раз. Человек может назвать Карлу любое из чисел последовательности 23, 3, 1, 2, 42 или даже число 67, которое Фред попросил записать уже после вашего звонка. Он вполне может назвать Карлу последовательность чисел 23, 3, 3, 1, 67, и это не будет противоречить тому, что было названо вам. Похоже, что этот человек с помощью небольших стикеров для каждого из своих собеседников отслеживает, какое из чисел и кому именно он назвал (рис. 5.5).

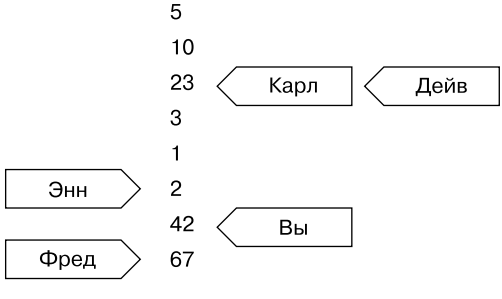


Рис. 5.5. Вид на блокнот человека, сидящего в кабинке

А теперь представьте, что это не один человек в кабинке, а целый зал кабинок с людьми, имеющими телефон и блокнот. Все они являются прообразами атомарных

переменных. У каждой переменной собственный порядок изменений (список значений в блокноте), но никаких отношений между этими переменными не существует. Если каждый звонящий (вы, Карл, Энн, Дейв и Фред) является прообразом потока, то получается именно то, что происходит при использовании каждой операцией упорядочения `memory_order_relaxed`. К человеку в кабинке можно обратиться с еще несколькими дополнительными просьбами, например: «Запиши это число и скажи мне, что было в конце списка» (`exchange`) и «Запиши *вот это* число, если число в конце списка равно *вот этому числу*, в противном случае назови число, о присутствии которого мне следовало бы догадаться» (`compare_exchange_strong`), но на общий принцип это не влияет.

Если обратиться к логике программы из листинга 5.5, то работа функции `write_x_then_y` похожа на чей-то звонок человеку в кабинке `x` с просьбой о записи значения `true` и последующий звонок человеку в кабинке `y` с просьбой о записи значения `true`. Поток, выполняющий функцию `read_y_then_x`, звонит человеку в кабинке `y` с просьбой назвать значение снова и снова, пока тот не скажет `true`, а затем звонит человеку в кабинке `x` с просьбой назвать значение. Человек в кабинке `x` не обязан называть вам какое-либо конкретное значение из своего списка и с полным правом может сказать `false`.

Это усложняет работу с нестрогими атомарными операциями. Чтобы улучшить синхронизацию между потоками, их следует использовать в сочетании с атомарными операциями, имеющими более строгую семантику упорядочения. Я настоятельно рекомендую избегать применения нестрогих атомарных операций без крайней необходимости и даже тогда использовать их с наибольшей осторожностью. Учитывая трудности с интуитивным предсказанием результатов, получаемых при наличии всего лишь двух потоков и двух переменных в листинге 5.5, нетрудно представить себе возможные сложности при задействовании большего количества потоков и переменных.

Одним из способов, позволяющих добиться дополнительной синхронизации без издержек, связанных с развернутым последовательно согласованным упорядочением, является использование *упорядочения на основе захвата — освобождения*.

Упорядочение на основе захвата — освобождения

Упорядочение на основе захвата — освобождения является расширением нестрогого упорядочения, при котором по-прежнему отсутствует тотальный порядок выполнения операций, но вводятся элементы синхронизации. Согласно данной модели упорядочения атомарные загрузки считаются операциями *захвата* (`memory_order_acquire`), атомарные сохранения — операциями *освобождения* (`memory_order_release`), а атомарные операции чтения — изменения — записи (такие как `fetch_add()` или `exchange()`) — либо операциями *захвата*, либо операциями *освобождения*, либо и тем и другим (`memory_order_acq_rel`). Синхронизируются два потока — тот, что выполняет освобождение, и тот, что выполняет захват. *Операция освобождения «синхронизируется с» операцией захвата, считывающей записанное значение*. Это означает, что разные потоки могут по-прежнему видеть разное упорядочение,

но это упорядочение ограничено. В листинге 5.7 показан переработанный код из листинга 5.4, в котором вместо семантики последовательной согласованности используется семантика захвата — освобождения.

Листинг 5.7. Семантика захвата — освобождения не подразумевает наличия полной упорядоченности

```
#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))   ←❶
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_acquire))    ←❷
        ++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);     ←❸
}
```

В данном случае утверждение `assert` ❸ может сработать (как и в случае применения нестрогого упорядочения), поскольку можно как для загрузки, относящейся к `x` ❷, так и для загрузки, относящейся к `y` ❶, считать значение `false`. Запись в `x` и `y` выполняется в разных потоках, поэтому упорядочение от захвата до освобождения в обоих случаях не влияет на операции в других потоках.

На рис. 5.6 показано отношение «происходит до» из листинга 5.7, а также возможный исход в том случае, когда каждый из двух считывающих потоков имеет собственное представление об окружающем мире. Как объяснялось ранее, такая возможность обусловливается отсутствием отношения «происходит до», принудительно задающего упорядочение.

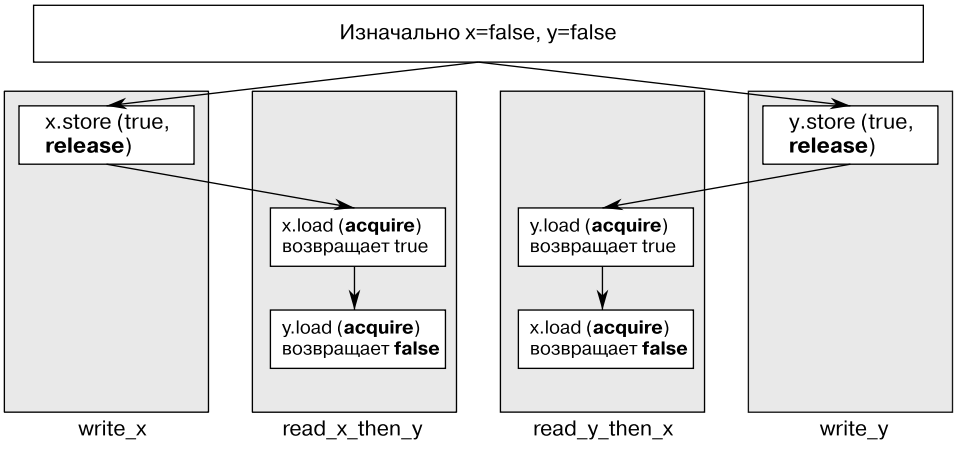


Рис. 5.6. Захват — освобождение и отношение «происходит до»

Чтобы понять, в чем заключается преимущество упорядочения на основе захвата — освобождения, следует рассмотреть два сохранения из одного и того же потока, как в листинге 5.5. Если для сохранения, относящегося к `y`, изменить семантику упорядочения на `memory_order_release`, а для загрузки, относящейся к `y`, воспользоваться семантикой `memory_order_acquire`, как в листинге 5.8, то операции, относящиеся к `x`, будут упорядочены.

Листинг 5.8. За счет операции с семантикой захвата — освобождения может накладываться упорядочение на нестрогие 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);
    y.store(true,std::memory_order_release);
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));

    if(x.load(std::memory_order_relaxed))
        ++z;
}
```

❶ ❷ ❸ Цикл в ожидании установки для y значения true ❹

```
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); ◀——❺
}
```

Со временем операция загрузки, относящаяся к `y` ❸, увидит значение `true`, записанное операцией сохранения ❷. Поскольку для сохранения используется семантика `memory_order_release`, а для загрузки — семантика `memory_order_acquire`, сохранение синхронизируется с загрузкой. Сохранение, относящееся к `x` ❶, происходит до сохранения, относящегося к `y` ❷, поскольку они выполняются в одном и том же потоке. Так как сохранение, относящееся к `y`, синхронизировано с загрузкой, относящейся к `y`, сохранение, относящееся к `x`, также происходит до загрузки, относящейся к `y`, следовательно, происходит до загрузки, относящейся к `x` ❹. Таким образом, загрузка, относящаяся к `x`, должна прочитать `true`, а утверждение `assert` ❺ сработать не может. Если загрузка, относящаяся к `y`, не была бы в цикле `while`, такого могло бы и не быть: загрузка, относящаяся к `y`, может считать `false`, тогда не будет предъявляться никаких требований к значению, считанному из `x`. Чтобы обеспечить синхронизацию, операции захвата и освобождения должны выполняться попарно. Значение, сохраненное операцией освобождения, должна видеть операция захвата, иначе они обе не дадут никакого эффекта. Если бы сохранение в строке ❷ либо загрузка в строке ❸ были нестрогими операциями, то никакого упорядочения доступа к `x` не было бы, следовательно, не было бы и гарантий того, что загрузка в строке ❹ считала бы `true` и утверждение `assert` могло бы сработать.

Упорядочение на основе захвата — освобождения можно по-прежнему представить в виде людей с блокнотами, сидящих в кабинках, но модель придется дополнить. Сначала представим, что каждое выполненное сохранение является частью пакета обновлений, поэтому в ходе звонка человеку с просьбой о записи числа ему также сообщается, к какому пакету относится это обновление: «Пожалуйста, запишите число 99 как часть пакета 423». Последнее сохранение в пакете оговаривается особо: «Пожалуйста, запишите число 147, и это будет последним сохранением в пакете 423». Человек в кабинке должным образом запишет эту информацию, как и тот, кто дал ему это значение. Тем самым будет смоделирована операция захвата — освобождения. При следующем обращении с просьбой о записи значения вы увеличите номер пакета на единицу: «Пожалуйста, запишите число 41 как часть пакета 424».

Теперь при запросе значения есть выбор: можно либо попросить назвать значение (это будет нестрогой загрузкой), и человек скажет вам только число, либо попросить назвать значение и сообщить, является ли оно последним в пакете (смоделировать загрузку с захватом). Если запрашивается информация о пакете и значение не явля-

ется в нем последним, человек скажет вам что-нибудь вроде: «Это число 987, являющееся обычным значением», а если значение было последним в пакете, сообщение будет примерно таким: «Это число 987, и оно является последним в пакете 956 от Энн». Тут вступает в действие семантика захвата — освобождения: если при запросе значения сообщить человеку обо всех известных вам пакетах, он посмотрит в свой список последних значений из каждого такого пакета и назовет либо это число, либо число, находящееся ниже по списку.

Как с помощью этого представления моделируется семантика захвата — освобождения? Давайте выясним на нашем примере. Сначала поток `a` выполняет функцию `write_x_then_y`, как бы говоря человеку в кабинке `x`: «Пожалуйста, запишите `true` как часть пакета 1 из потока `a`», что он должным образом и запишет. Затем поток `a` скажет человеку в кабинке `y`: «Пожалуйста, запишите `true` в качестве последнего значения пакета 1 из потока `a`», и он добросовестно это запишет. Тем временем поток `b` выполняет функцию `read_y_then_x`. Этот поток продолжает просить человека в кабинке `y` назвать значение с информацией о пакете, пока тот не скажет `true`. Возможно, придется обращаться с просьбой не один раз, но со временем этот человек скажет `true`. А человек в кабинке `y` скажет не только `true`, но и: «Это последняя запись в пакете 1 от потока `a`».

Теперь поток `b` просит человека в кабинке `x` назвать значение, но на этот раз говорит: «Не могу ли я получить значение? И кстати, я знаю, что есть пакет 1 от потока `a`». То есть человек из кабинки `x` должен найти в списке последнее упоминание о пакете 1 от потока `a`. Единственным найденным упоминанием будет значение `true`, которое также будет последним значением в его списке, поэтому он должен считать это значение, иначе нарушит правила игры.

Если вернуться к определению отношения «происходит до» среди потоков из подраздела 5.3.2, то одним из важных свойств станет его транзитивность: *если А среди потоков происходит до Б, а Б среди потоков происходит до В, то А среди потоков происходит до В*. Это означает, что для синхронизации данных среди нескольких потоков можно воспользоваться упорядочением на основе захвата — освобождения, даже если к данным не обращались никакие промежуточные потоки.

Транзитивная синхронизация с использованием упорядочения на основе захвата — освобождения

Чтобы разобраться в транзитивном упорядочении, нужны как минимум три потока. Первый поток вносит изменения в совместно используемые переменные, выполняя в отношении одной из них операцию сохранения с освобождением. Затем второй поток считывает значение переменной, к которой применялась операция сохранения с освобождением, для чего применяет к ней операцию загрузки с захватом и выполняет операцию сохранения с освобождением в отношении второй совместно используемой переменной. И наконец, третий поток реализует операцию загрузки с захватом в отношении второй совместно используемой переменной. При условии, что операции загрузки с захватом видят значения, записанные операциями сохранения с освобождением, поддерживая тем самым отношения «синхронизируется с»,

третий поток может прочитать значения других переменных, сохраненные первым потоком, даже если промежуточный поток не обращался ни к одной из них. Этот сценарий показан в листинге 5.9.

Листинг 5.9. Транзитивная синхронизация с помощью упорядочения на основе захвата — освобождения

```
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);
    sync1.store(true,std::memory_order_release);   ❶ Установка sync1
}
void thread_2()
{                                                  ❷ Цикл в ожидании установки
    while(!sync1.load(std::memory_order_acquire));
    sync2.store(true,std::memory_order_release);   ❸ Установка sync2
}
void thread_3()
{                                                  ❹ Цикл в ожидании установки
    while(!sync2.load(std::memory_order_acquire));
    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);
}
```

Несмотря на то что `thread_2` обращается только к переменным `sync1` ❷ и `sync2` ❸, этого вполне достаточно для синхронизации между `thread_1` и `thread_3`, чтобы гарантировать невозможность срабатывания утверждения `assert`. Начнем с того, что сохранения в `data` из потока `thread_1` происходят до сохранения в `sync1` ❶, поскольку в последовательности одного и того же потока стоят перед ним. Так как загрузка из `sync1` ❶ находится в цикле `while`, она со временем увидит значение, сохраненное из `thread_1`, и сформирует вторую половину пары освобождения — захвата. Следовательно, сохранение в `sync1` происходит до финальной загрузки из `sync1` в цикле `while`. Эта загрузка находится в последовательности до (и поэтому происходит до) сохранения в `sync2` ❸, которое формирует пару освобождения — захвата с финальной загрузкой из цикла `while` в `thread_3` ❹. Таким образом, сохранение в `sync2` ❸ происходит до загрузки ❹, которая выполняется до загрузок из `data`. Благодаря транзитивной сути отношений «происходит до» их можно выстраивать в цепочку: сохранения в `data` происходят до сохранения в `sync1` ❶, а оно выполняется до загрузки из `sync1` ❷, которая происходит до сохранения в `sync2` ❸, происходящего до загрузки из `sync2` ❹, которая выполняется до загрузок из `data`. Таким образом, сохранения в `data` в `thread_1` происходят до загрузок из `data` в `thread_3`, и утверждения `assert` не смогут сработать.

В данном случае можно объединить sync1 и sync2 в одну переменную, воспользовавшись в thread_2 операцией чтения — изменения — записи с сигнатурой упорядочения memory_order_acq_rel. Одним из вариантов может быть использование функции compare_exchange_strong(), чтобы гарантировать, что значение будет обновлено, только когда станет видимым сохранение из 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 — изменения — записи важно выбрать нужную семантику. В данном случае требуется как семантика захвата, так и семантика освобождения, следовательно, подходит вариант memory_order _acq_rel, но можно воспользоваться и другими семантиками упорядочения. Операция fetch_sub с семантикой memory_order_acquire ни с чем не синхронизируется, несмотря на то что сохраняет значение, поскольку она не является операцией освобождения. Аналогично сохранение не может синхронизироваться с операцией fetch_or, имеющей семантику memory_order_release, потому что та часть fetch_or, которая выполняет чтение, не является операцией захвата. Операции чтения — изменения — записи с семантикой memory_order_acq_rel ведут себя и как операции захвата, и как операции освобождения, поэтому предшествующее сохранение может быть синхронизировано с такой операцией, а она — быть синхронизирована с последующей загрузкой, что и происходит в данном примере.

Если смешивать операции захвата — освобождения с последовательно согласованными операциями, то последовательно согласованные загрузки ведут себя подобно загрузкам с семантикой захвата, а последовательно согласованные сохранения — подобно сохранениям с семантикой освобождения. Последовательно согласованные операции чтения — изменения — записи ведут себя и как операции захвата, и как операции освобождения. Нестрогие операции так и остаются нестрогими, но ограничиваются дополнительными отношениями «синхронизируется с» и последующими отношениями «происходит до», которые вводятся посредством использования семантики захвата — освобождения.

Несмотря на интуитивно неочевидные последствия, всем, кто использует блокировки, приходится иметь дело с аналогичными проблемами упорядочения:

блокировка мьютекса является операцией захвата, а его разблокировка — операцией освобождения. На примере мьютексов можно понять, что при чтении значения нужно обеспечить блокировку того же мьютекса, который был заблокирован при его записи, и такая же схема применятся здесь, чтобы обеспечить упорядочение, ваши операции захвата и освобождения должны выполняться в отношении одной и той же переменной. Если данные защищены мьютексом, то исключительный характер блокировки означает, что результат неотличим от того, что было бы, если бы блокировка и разблокировка были последовательно согласованными операциями. По аналогии с этим, если для создания простой блокировки по отношению к атомарным переменным используется упорядочение на основе захвата — освобождения, то с точки зрения кода, *использующего* эту блокировку, будет проявляться поведение последовательной согласованности, даже если природа внутренних операций иная.

Если ваши атомарные операции не нуждаются в строгости последовательно согласованного упорядочения, то попарная синхронизация упорядочения на основе захвата — освобождения может обеспечить синхронизацию со значительно меньшими издержками по сравнению с глобальным упорядочением, требующимся для последовательно согласованных операций. Издержки здесь выражаются в затратах времени на то, чтобы осмыслить возможность обеспечения правильной работы упорядочения, и на то, чтобы понять, что поведение приложения в многопоточной среде не вызывает проблем, а это сложно сделать на интуитивном уровне.

Зависимость от данных при упорядочении на основе захвата — освобождения и использовании семантики memory_order_consume

Во вступлении к этому разделу говорилось, что семантика `memory_order_consume` была частью модели упорядочения на основе захвата — освобождения, но в приведенном ранее описании она не упоминалась. Дело в том, что `memory_order_consume` является семантикой особого вида: она целиком завязана на зависимость от данных и вводит нюансы такой зависимости в отношение «происходит до» среди потоков, упомянутое в подразделе 5.3.2. Ее специфичность заключается еще и в том, что в стандарте C++17 настоятельно рекомендуется отказаться от ее использования. Поэтому здесь она рассматривается только для полноты изложения материала: применять `memory_order_consume` в своем коде не нужно!

В целом понятие зависимости от данных не отличается особой сложностью: зависимость от данных между двумя операциями наблюдается в том случае, если вторая операция является результатом выполнения первой. С зависимостью от данных связаны два новых отношения: *«выполняется по зависимости до»* (dependency-ordered-before) и *«переносит зависимость к»* (carries-a-dependency-to). Подобно отношению «находится в последовательности до», зависимость «переносит зависимость к» применяется исключительно в одном и том же потоке и моделирует зависимость от данных между операциями: если результат одной операции (А) используется в качестве операнда для другой операции (Б), то А переносит зависимость к Б. Если результат операции А является значением скалярного типа, например `int`, то

отношение по-прежнему применяется, если результат А сохраняется в переменной, а затем эта переменная используется в качестве операнда для операции Б. Эта операция также транзитивна, поэтому, если А переносит зависимость к Б, а Б переносит зависимость к В, то А переносит зависимость к В.

В отличие от этого, отношение «выполняется по зависимости до» может применяться среди потоков. Оно вводится путем использования операций атомарной загрузки, помеченных признаком `memory_order_consume`. Это особый случай `memory_order_acquire`, ограничивающий синхронизированный доступ к данным прямыми зависимостями: операция сохранения (А), помеченная признаком `memory_order_release`, `memory_order_acq_rel` или `memory_order_seq_cst`, находится в отношении «выполняется по зависимости до» с операцией загрузки (Б), помеченной признаком `memory_order_consume`, если та потребляет результаты считывания сохраненного значения. Это отношение противоположно отношению «синхронизировано с», получаемому, когда загрузка помечена признаком `memory_order_acquire`. Если затем данная операция (Б) переносит зависимость к какой-нибудь другой операции (В), то А также «выполняется по зависимости до» В.

Это не принесло бы никакой пользы в смысле синхронизации, если бы не влияло на отношение «происходит до» среди потоков, но такое влияние есть: если А «выполняется по зависимости до» Б, то А среди потоков также «происходит до» Б.

Задействовать эту разновидность упорядочения доступа к памяти можно, к примеру, когда атомарная операция загружает указатель на некие данные. За счет применения признака `memory_order_consume` в отношении загрузки и признака `memory_order_release` в отношении предшествующего ей сохранения обеспечивается корректная синхронизация доступа к указываемым данным даже без предъявления каких-либо требований синхронизации других данных, не имеющих зависимости. Пример реализации такого сценария показан в листинге 5.10.

Листинг 5.10. Использование std::memory_order_consume для синхронизации доступа к данным

```
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";                                          ❶
    a.store(99,std::memory_order_relaxed);  ◄──┘
    p.store(x,std::memory_order_release);       ◄──❷
}
void use_x()
{
    X* x;                                                  ❸
    while(!(x=p.load(std::memory_order_consume))) ◄──┘
```

```
    std::this_thread::sleep(std::chrono::microseconds(1));
    assert(x->i==42);                                          ❹
    assert(x->s=="hello");                                     ❺
    assert(a.load(std::memory_order_relaxed)==99);             ❻
}
int main()
{
    std::thread t1(create_x);
    std::thread t2(use_x);
    t1.join();
    t2.join();
}
```

Несмотря на то что сохранение, относящееся к `a` ❶, находится в последовательности до сохранения, относящегося к `p` ❷, а сохранение, относящееся к `p`, помечено признаком `memory_order_release`, загрузка, относящаяся к `p` ❸, помечена признаком `memory_order_consume`. Это означает, что сохранение, относящееся к `p`, происходит до выполнения только тех выражений, которые зависят от значения, загруженного из `p`. Следовательно, утверждения, относящиеся к компонентным данным структуры `X` (❹ и ❺), гарантированно не сработают, поскольку загрузка, относящаяся к `p`, переносит зависимость к этим выражениям посредством переменной `x`. В то же время утверждение в отношении значения `a` ❻ может либо сработать, либо нет: эта операция не зависит от значения, загруженного из `p`, следовательно, нет никаких гарантий относительно считываемого значения. Как видите, очевидность этого особо подчеркивается пометкой `memory_order_relaxed`.

Иногда издержки, связанные с переносом зависимости, крайне нежелательны. Нужно, чтобы компилятор мог кэшировать значения в регистрах и изменять порядок выполнения операций для оптимизации кода, а не возился с зависимостями. В таких случаях можно воспользоваться шаблоном функции `std::kill_dependency()`, чтобы явно разорвать цепочку зависимостей. Функция `std::kill_dependency()` просто копирует предоставленный аргумент в возвращаемое значение, но при этом разрывает цепочку зависимостей. Например, если имеется глобальный массив, предназначенный только для чтения, и при извлечении индекса из этого массива из другого потока используется признак `std::memory_order_consume`, то, чтобы сообщить компилятору, что ему не нужно перечитывать содержимое элемента массива, можно, как в следующем примере, воспользоваться функцией `std::kill_dependency()`:

```
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)]);
}
```

В реальном коде, как только появится желание воспользоваться `memory_order_consume`, следует применять `memory_order_acquire`, и тогда надобность в использовании функции `std::kill_dependency` отпадет сама собой.

Теперь, когда основы упорядочения доступа к памяти рассмотрены, настало время обратиться к более сложным частям отношения «синхронизируется с», проявляющимся в виде *последовательностей освобождений* (release sequences).

5.3.4. Последовательности освобождений и отношения «синхронизируется с»

В подразделе 5.3.1 уже упоминалось, что отношение «синхронизировано с» между сохранением в атомарную переменную и загрузкой значения этой атомарной переменной из другого потока можно получить, даже если между сохранением и загрузкой выполняется последовательность операций чтения — изменения — записи, при условии, что все операции помечены подходящими признаками упорядочения. Теперь, когда рассмотрены возможные признаки упорядочения доступа к памяти, о них можно поговорить подробнее. Если операция сохранения помечена признаком `memory_order_release`, `memory_order_acq_rel` или `memory_order_seq_cst`, а операция загрузки — признаком `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 представляет собой *последовательность освобождений*, а начальное сохранение «синхронизировано с» (при использовании признака `memory_order_acquire` или `memory_order_seq_cst`) или же «выполняется по зависимости до» (в случае использования признака `memory_order_consume`) финальной операции загрузки. Любые атомарные операции чтения — изменения — записи в цепочке могут иметь любые признаки упорядочения доступа к памяти (даже `memory_order_relaxed`).

Чтобы разобраться в значении и степени важности всего этого, рассмотрим значение типа `atomic<int>`, указанное в качестве счетчика `count` для подсчета количества записей в совместно используемой очереди (листинг 5.11).

Листинг 5.11. Чтение значений из очереди с помощью 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);
    }

    count.store(number_of_items,std::memory_order_release); ❶ Начальное сохранение
}
void consume_queue_items()
{
```

```
    while(true)
    {
        int item_index;
        if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)
        {
            wait_for_more_items();
            continue;
        }
        process(queue_data[item_index-1]);
    }
}
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();
}
```

❷ Операция чтения — изменения — записи

❸ Ожидание дополнительных элементов

❹ Чтение queue_data в безопасном режиме

Один из способов решения поставленной задачи заключается в применении потока, создающего данные, для хранения записей в совместно используемом буфере, а затем в выполнении операции `count.store(number_of_items, memory_order_release)` ❶, чтобы дать знать другим потокам о том, что данные доступны. Затем потоки, потребляющие элементы очереди, смогут выполнить операцию `count.fetch_sub(1,memory_order_acquire)` ❷, чтобы запросить элемент из очереди до чтения совместно используемого буфера ❹. Как только значение счетчика `count` станет равным нулю, очередь опустеет и поток вынужден будет находиться в состоянии ожидания ❸.

Если элементы очереди потребляет всего один поток, то все в порядке: `fetch_sub()` является операцией чтения с семантикой `memory_order_acquire`, а для сохранения используется семантика `memory_order_release`, поэтому сохранение синхронизируется с загрузкой и поток может прочитать элемент из буфера. Если же чтение выполняется из двух потоков, то при втором вызове функции `fetch_sub()` будет видно значение, записанное для ее первого вызова, а не значение, записанное операцией сохранения `store`. Без правила о последовательности освобождений у этого второго потока не будет отношения «происходит до» с первым потоком, и чтение совместно используемого буфера не окажется безопасным до тех пор, пока у первой операции `fetch_sub()` также не будет семантики `memory_order_release`, которая введет ненужную синхронизацию между двумя потребляющими потоками. Без правила, задающего последовательность освобождений, или без указания семантики `memory_order_release` в отношении операций `fetch_sub` не требовалось бы, чтобы второй потребитель видел то, что сохраняется в `queue_data`, и возникло бы состояние гонки за данными. К счастью, первый вызов `fetch_sub()` участвует в последовательности освобождений и операция `store()` синхронизирована со вторым вызовом `fetch_sub()`. Но отношение «синхронизируется с» между двумя потребляющими потоками по-прежнему отсутствует. Все это показано на рис. 5.7, где пунктирными

линиями обозначена последовательность освобождений, а сплошными линиями — отношения «происходит до».

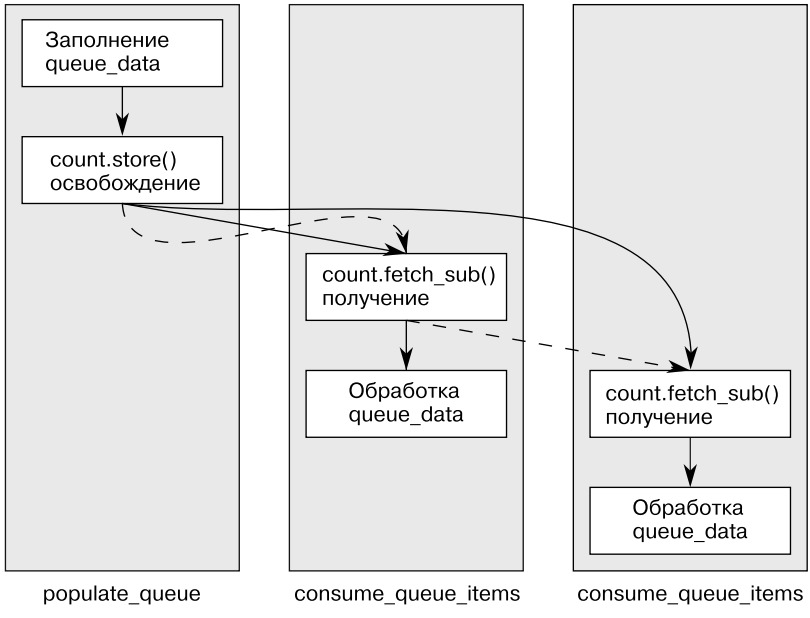


Рис. 5.7. Последовательность освобождений для операций с очередью из листинга 5.11

Цепочка может состоять из любого количества звеньев, но при условии, что все они являются операциями чтения — изменения — записи, такими как `fetch_sub()`. Операция `store()` будет по-прежнему синхронизирована с каждой операцией, помеченной признаком `memory_order_acquire`. В данном примере все звенья одинаковы и являются операциями захвата, но допускается также перемешивание различных операций с иной семантикой упорядочения доступа к памяти.

Основная часть отношений синхронизации создается благодаря применению к операциям над атомарными переменными семантики упорядочения доступа к памяти, но можно ввести и дополнительные ограничения упорядоченности путем использования *барьеров* (fences).

5.3.5. Барьеры

Библиотека атомарных операций была бы неполной без набора барьеров. Под ними подразумеваются операции, накладывающие ограничения на порядок доступа к памяти без изменения каких-либо данных и, как правило, в сочетании с атомарными операциями, использующими ограничения упорядочения `memory_order_relaxed`. *Барьеры* — это глобальные операции, влияющие на порядок выполнения других атомарных операций в том потоке, в котором устанавливается барьер. Их часто называют также *барьерами памяти*, потому что они помещают в код строку, за которую

не могут заходить конкретные операции. Помните, в подразделе 5.3.3 говорилось, что компилятор или процессор могут по своему усмотрению изменить порядок выполнения нестрогих операций над отдельными переменными. Барьеры ограничивают эту свободу и устанавливают отношения «происходит до» и «синхронизируется с», которых раньше не было.

Давайте, как показано в листинге 5.12, начнем с добавления барьеров между двумя атомарными операциями в каждом потоке из листинга 5.5.

Листинг 5.12. Нестрогие 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);          ←❶
    std::atomic_thread_fence(std::memory_order_release); ←❷
    y.store(true,std::memory_order_relaxed);          ←❸
}
void read_y_then_x()
{                                                     ❹
    while(!y.load(std::memory_order_relaxed));  ←┘
    std::atomic_thread_fence(std::memory_order_acquire);  ←❺
    if(x.load(std::memory_order_relaxed))  ←┐
            ++z;                           ❻
}
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); ←❼
}
```

Барьер освобождения ❷ синхронизируется с барьером захвата ❺, потому что загрузка из `y` в строке кода ❹ считывает значение, сохраненное кодом в строке ❸. Это означает, что сохранение в `x` в строке кода ❶ происходит до загрузки из `x` в строке кода ❻, поэтому должно быть считано значение `true` и утверждение `assert` в строке кода ❼ не сработает. Это сильно отличается от исходного варианта без использования барьеров, где сохранение в переменную `x` и загрузка из нее не были упорядочены и утверждение могло сработать. Обратите внимание на необходимость присутствия обоих барьеров: здесь для получения отношения «синхронизируется с» требуются освобождение в одном потоке и захват в другом потоке.

В данном случае барьер освобождения ❷ порождает эффект, аналогичный тому, который возник бы, если бы сохранение в `y` ❸ было помечено признаком `memory_order_release`, а не признаком `memory_order_relaxed`. Аналогично этому

барьер захвата ❺ действует так же, как действовала бы загрузка из y ❹ при наличии у нее признака memory_order_acquire. В этом заключается общий смысл использования всех барьеров: если операции захвата видят результат сохранения, происходящего после барьера освобождения, то барьер синхронизируется с операцией захвата, а если загрузка, выполняемая до барьера захвата, видит результат операции освобождения, то операция освобождения синхронизируется с барьером захвата. Барьеры, как и в показанном примере, могут быть с обеих сторон, и тогда, если загрузка, выполняемая до барьера захвата, видит значение, записанное сохранением, выполненным после барьера освобождения, барьер освобождения синхронизируется с барьером захвата.

Хотя барьерная синхронизация зависит от значений, считанных или записанных операциями до или после барьера, важно отметить, что точкой синхронизации служит сам барьер. Если взять функцию write_x_then_y из листинга 5.12 и переставить запись в x после барьера, как в следующем примере кода, то вычисление условия утверждения assert в true больше не гарантируется, 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, а следовательно, не упорядочены. Барьер вводит упорядочение, только когда стоит между сохранением в x и сохранением в y. Само по себе присутствие или отсутствие барьера не влияет ни на какие принудительные упорядочения, основанные на отношениях «происходит до», существующих благодаря другим атомарным операциям.

Данный пример и практически любой другой, приводившиеся до сих пор в этой главе, выстраивались полностью из переменных атомарного типа. Но реальная польза от применения атомарных операций для получения принудительной упорядоченности заключается в том, что они могут навязать упорядоченность неатомарным операциям и, как было показано в листинге 5.2, исключить вероятность неопределенного поведения, вызванного гонкой за данными.

5.3.6. Упорядочение неатомарных операций с помощью атомарных операций

Если заменить x из листинга 5.12 обычным неатомарным значением bool-типа (как в листинге 5.13), поведение гарантированно не изменится.

Листинг 5.13. Упорядочение, навязанное неатомарным операциям

```
#include <atomic>
#include <thread>
#include <assert.h>
bool x=false;
std::atomic<bool> y;
std::atomic<int> z;
```

❶ Теперь x является простой неатомарной переменной

```
void write_x_then_y()
{
    x=true;
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x)
        ++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);
}
```

❷ Сохранение в x до барьера

❸ Сохранение в y до барьера

Ожидание видимости записи из ❷

Этот код прочтет значение, записанное кодом в строке ❶ ❹

❺ Это утверждение не сработает

Барьеры по-прежнему обеспечивают принудительное упорядочение сохранения в `x` ❶ и в `y` ❷, а также загрузки из `y` ❸ и из `x` ❹, а между сохранением в `x` и загрузкой из `x` по-прежнему имеется отношение «происходит до», поэтому утверждение `assert` ❺ не сработает. Сохранение в `y` ❷ и загрузка из `y` ❸ по-прежнему должны быть атомарными, в противном случае возникнет состояние гонки за доступ к переменной `y`, но барьеры навязывают упорядочение операциям в отношении `x`, после того как считывающий поток увидел сохраненное значение `y`. Это принудительное упорядочение означает, что состояния гонки за доступ к переменной `x` не возникает, несмотря на то что ее значение изменяется одним, а считывается другим потоком.

Упорядочить неатомарные операции можно не только с помощью барьеров. Ранее в листинге 5.10 уже были показаны эффекты упорядочения, получаемые от применения пар признаков `memory_order_release` — `memory_order_consume`, устанавливающих порядок неатомарного доступа к объекту, память под который была распределена в динамическом режиме. Многие примеры, приведенные в данной главе, можно переписать, заменив `memory_order_relaxed`-операции простыми неатомарными операциями.

5.3.7. Упорядочение неатомарных операций

Особую важность часть отношения «происходит до» с формулировкой «находится в последовательности до» приобретает при упорядочении неатомарных операций посредством использования атомарных операций. Если неатомарная операция находится в последовательности до атомарной операции и эта атомарная операция выполняется до операции в другом потоке, неатомарная операция также выполня-

ется до выполнения этой операции в другом потоке. Именно отсюда берется упорядочение операций над x в листинге 5.13, и именно в этом заключается причина работоспособности примера из листинга 5.2. Кроме того, это является основой для таких средств синхронизации более высокого уровня, имеющихся в стандартной библиотеке C++, как мьютексы и условные переменные. Чтобы увидеть, как все это работает, рассмотрим простой мьютекс со спин-блокировкой из листинга 5.1.

Операция блокировки `lock()` представляет собой цикл, завершаемый по результатам выполнения функции `flag.test_and_set()`, использующей упорядочение `std::memory_order_acquire`, а операция разблокировки `unlock()` является вызовом функции `flag.clear()` с упорядочением `std::memory_order_release`. Когда первый поток вызывает `lock()`, флаг изначально сброшен, поэтому первый вызов функции `test_and_set()` установит флаг, возвратит значение `false`, показывающее, что теперь этот поток завладел блокировкой, и завершит цикл. Затем поток может свободно изменять данные, защищенные мьютексом. Любой другой поток, вызывающий `lock()` в это время, обнаружит, что флаг уже установлен, и будет заблокирован на выполнении в цикле функции `test_and_set()`.

Когда поток, завладевший блокировкой, завершит внесение изменений в защищенные данные, он вызовет функцию `unlock()`, которая, в свою очередь, вызовет `flag.clear()` с семантикой упорядочения `std::memory_order_release`. Затем этот процесс «синхронизируется с» (см. подраздел 5.3.1) последующим вызовом функции `flag.test_and_set()` из вызова `lock()` в другом потоке, так как у этого вызова имеется семантика упорядочения `std::memory_order_acquire`. Поскольку необходимо, чтобы изменение защищенных данных находилось в последовательности до вызова функции `unlock()`, это изменение «происходит до» `unlock()`, а следовательно, «происходит до» последующего вызова `lock()` из второго потока (благодаря отношению «синхронизируется с» между `unlock()` и `lock()`) и «происходит до» любого обращения к этим данным из второго потока после того, как он завладел блокировкой.

Хотя другие реализации мьютексов будут иметь иные внутренние операции, основной принцип будет тем же самым: `lock()` является операцией захвата во внутренней области памяти, а `unlock()` — операцией освобождения в той же области памяти.

Каждый из механизмов синхронизации, рассмотренных в главах 2–4, гарантирует упорядочение в понятиях отношения «синхронизируется с». Именно это позволяет вам использовать их для синхронизации доступа к вашим данным и предоставлять гарантии упорядочения. Далее приводятся отношения синхронизации, предоставляемые этими средствами.

std::thread

- Завершение работы конструктора `std::thread` синхронизируется с вызовом предоставленной функции или вызываемого объекта в новом потоке.
- Завершение потока синхронизируется с возвращением из успешного вызова `join` в отношении объекта `std::thread`, владеющего данным потоком.

std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex

- Все вызовы `lock` и `unlock` и успешные вызовы `try_lock`, `try_lock_for` или `try_lock_until` в отношении отдельно взятого объекта *мьютекса* формируют единый общий *порядок блокировки* мьютекса.

- Вызов `unlock` в отношении отдельно взятого объекта мьютекса синхронизируется с последующим вызовом `lock` либо с дальнейшим успешным вызовом `try_lock`, `try_lock_for` или `try_lock_until` в отношении данного объекта в порядке блокировки мьютекса.
- Неудачные вызовы `try_lock`, `try_lock_for` или `try_lock_until` не участвуют ни в каких отношениях синхронизации.

std::shared_mutex, std::shared_timed_mutex

- Все вызовы `lock`, `unlock`, `lock_shared` и `unlock_shared` и успешные вызовы `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`, `try_lock_shared_for` или `try_lock_shared_until` в отношении отдельно взятого объекта мьютекса формируют единый общий порядок блокировки мьютекса.
- Вызов `unlock` в отношении отдельно взятого объекта мьютекса синхронизируется с последующими вызовами `lock` или `shared_lock` либо с успешными вызовами `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`, `try_lock_shared_for` или `try_lock_shared_until` в отношении данного объекта в порядке блокировки мьютекса.
- Неудачные вызовы `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`, `try_lock_shared_for` или `try_lock_shared_until` не участвуют ни в каких отношениях синхронизации.

std::promise, std::future и std::shared_future

- Успешное завершение вызова `set_value` или `set_exception` в отношении отдельно взятого объекта `std::promise` синхронизируется с успешным возвращением из вызова `wait` или `get` либо вызова `wait_for` или `wait_until`, возвращающего состояние `std::future_status::ready` в отношении фьючерса, использующего совместно с промисом одно и то же асинхронное состояние.
- Деструктор отдельно взятого объекта `std::promise`, сохраняющий исключение `std::future_error` в совместно используемом асинхронном состоянии, связанном с промисом, синхронизируется с успешным возвращением из вызова `wait` или `get` либо из вызова `wait_for` или `wait_until`, возвращающего состояние `std::future_status::ready` в отношении фьючерса, использующего совместно с промисом одно асинхронное состояние.

std::packaged_task, std::future и std::shared_future

- Успешное завершение обращения к оператору вызова функции отдельно взятого объекта `std::packaged_task` синхронизируется с успешным возвращением из вызова `wait` или `get` либо из вызова `wait_for` или `wait_until`, возвращающего `std::future_status::ready` в отношении фьючерса, использующего совместно с упакованной задачей одно асинхронное состояние.
- Деструктор отдельно взятого объекта `std::packaged_task`, который сохраняет исключение `std::future_error` в совместно используемом асинхронном состоянии, связанном с упакованной задачей, синхронизируется с успешным возвращением

из вызова wait или get либо wait_for или wait_until, возвращающего состояние std::future_status::ready в отношении фьючерса, использующего совместно с упакованной задачей одно асинхронное состояние.

std::async, std::future и std::shared_future

- Завершение потока, выполняющего задачу, запущенную вызовом std::async с политикой std::launch::async, синхронизируется с успешным возвращением из вызова wait или get либо wait_for или wait_until, возвращающего std::future_status::ready в отношении фьючерса, использующего совместно с порожденной задачей одно асинхронное состояние.
- Завершение задачи, запущенной посредством вызова std::async с политикой std::launch::deferred, синхронизируется с успешным возвращением из вызова wait или get либо wait_for или wait_until, возвращающего std::future_status::ready в отношении фьючерса, использующего совместно с промисом одно асинхронное состояние.

std::experimental::future, std::experimental::shared_future и продолжения

- Событие, которое служит причиной готовности асинхронного совместно используемого состояния, синхронизируется с вызовом функции продолжения, запланированной в этом состоянии.
- Завершение функции продолжения синхронизируется с успешным возвращением из вызова wait или get либо wait_for или wait_until, возвращающего std::future_status::ready в отношении фьючерса, совместно использующего одно и то же асинхронное состояние с фьючерсом, возвращаемым из вызова than, запланировавшего продолжение, или с вызовом любого продолжения, запланированного для этого фьючерса.

std::experimental::latch

- Инициирование каждого вызова метода count_down или count_down_and_wait в конкретном экземпляре защелки std::experimental::latch синхронизируется с завершением каждого успешного вызова метода wait или count_down_and_wait в этой защелке.

std::experimental::barrier

- Инициирование каждого вызова arrive_and_wait или arrive_and_drop в отношении отдельно взятого экземпляра std::experimental::barrier синхронизируется с завершением каждого последующего успешного вызова arrive_and_wait на этом барьере.

std::experimental::flex_barrier

- Инициирование каждого вызова arrive_and_wait или arrive_and_drop в отношении отдельно взятого экземпляра std::experimental::flex_barrier синхронизируется с завершением каждого последующего успешного вызова arrive_and_wait на этом барьере.

- Инициирование каждого вызова `arrive_and_wait` или `arrive_and_drop` в отношении отдельно взятого экземпляра `std::experimental::flex_barrier` синхронизируется с последующим инициированием функции завершения на этом барьере.

std::condition_variable и std::condition_variable_any

- Условные переменные не обеспечивают никаких синхронизирующих отношений. Они представляют собой средства оптимизации по циклам активного ожидания, и синхронизация в целом обеспечивается операциями над связанным мьютексом.

Резюме

В данной главе шел разговор о низкоуровневых особенностях модели памяти C++ и атомарных операциях, составляющих основу синхронизации потоков. Кроме того, рассмотрены основные атомарные операции, предоставляемые специализациями шаблона класса `std::atomic<>`, а также обобщенный атомарный интерфейс, предоставляемый первичным шаблоном `std::atomic<>` и шаблоном `std::experimental::atomic_shared_ptr<>`, операции над этими типами и особенности различных вариантов упорядочения доступа к памяти.

Кроме этого, были рассмотрены барьеры и способы их попарного применения с операциями над атомарными типами для получения принудительного упорядочения. И наконец, мы вернулись к тому, с чего начали, и поговорили о способах использования атомарных операций для принудительного упорядочения неатомарных операций, реализуемых в разных потоках. Мы также обсудили синхронизирующие отношения, обеспечиваемые средствами более высокого уровня.

В следующей главе рассмотрим применение высокоуровневых средств синхронизации с атомарными операциями для конструирования эффективных контейнеров, предназначенных для конкурентного доступа к ним. И наконец напишем алгоритмы для параллельной обработки данных.

6 Разработка конкурентных структур данных с блокировками

В этой главе

- Что означает разработка структур данных, допускающих конкурентные обращения.
- Рекомендации по разработке таких структур.
- Примеры реализации структур данных, допускающих конкурентные обращения.

В предыдущей главе были рассмотрены низкоуровневые особенности атомарных операций и модель памяти. В этой главе мы отдохнем от низкоуровневых подробностей (хотя они нам еще понадобятся в главе 7) и поразмышляем над структурами данных.

Выбор структуры данных для решения проблем программирования может стать ключевой составляющей всего решения, и проблемы параллельного программирования не исключение. Если предполагается обращаться к структуре данных сразу из нескольких потоков, то либо она должна быть полностью неизменяемой, чтобы данные никогда не модифицировались и никакой синхронизации не требовалось, либо программу следует разрабатывать так, чтобы гарантировать корректную синхронизацию изменений между потоками. Один из вариантов заключается в применении для защиты данных отдельного мьютекса и внешней блокировки с использованием приемов, рассмотренных в главах 3 и 4, а другой — в разработке самой структуры данных, допускающей конкурентные обращения.

При разработке структуры данных, допускающей конкурентные обращения, можно воспользоваться основными строительными блоками многопоточных приложений из предыдущих глав, например мьютексами и условными переменными. Разумеется, нам уже встречалось несколько примеров, показывающих, как создать комбинацию из этих строительных блоков для разработки структур данных, допускающих безопасный конкурентный доступ сразу из нескольких потоков.

Сначала в этой главе мы рассмотрим ряд общих рекомендаций по созданию конкурентных структур данных. Затем, прежде чем переходить к более сложным структурам данных, еще раз поговорим о разработке базовых структур данных в плане применения основных строительных блоков блокировок и условных переменных. В главе 7 будет показано, как для создания структур данных без блокировок можно вернуться к основам и воспользоваться атомарными операциями, рассмотренными в главе 5.

Теперь завершим вводную часть и перейдем к рассмотрению всех составляющих разработки структур данных, допускающих конкурентные обращения.

6.1. Что означает разработка структур данных для конкурентного доступа

В наипростейшем представлении разработка структуры данных, допускающих конкурентные обращения, ведется для предоставления доступа к ней со стороны сразу нескольких потоков, выполняющих одинаковые или разные операции. При этом гарантируется, что каждый поток видит непротиворечивое представление данной структуры. Потеря или повреждение данных не допускаются, поддерживаются все инварианты и исключается возникновение проблемного состояния гонки за доступ к данным. Такая структура данных называется *потокобезопасной*. В общем, структура данных будет безопасна только для определенных видов конкурентного доступа. Вполне возможно, что несколько потоков будут одновременно выполнять над структурой данных один тип операции, а другая операция потребует исключительного доступа к ней одного из потоков. В иных случаях конкурентный доступ к структуре данных сразу нескольких потоков, выполняющих *разные* действия, может быть вполне безопасен, а обращение нескольких потоков, выполняющих *одно и то же* действие, вызовет проблемы.

Но это еще не все особенности разработки с учетом конкурентности. Здесь также уделяется внимание предоставлению потокам, обращающимся к структуре данных, возможности *конкурентных обращений*. В соответствии со своей природой мьютекс обеспечивает возможность *взаимного исключения*: в определенный момент завладеть блокировкой мьютекса может только один поток. Мьютекс защищает структуру данных, явно *предотвращая* конкурентный доступ к защищаемым данным.

Это называется *сериализацией*: потоки по очереди обращаются к данным, защищенным мьютексом, они должны получать к ним доступ не одновременно, а последовательно. Соответственно, чтобы допустить реальный конкурентный доступ, конструкция структуры данных должна быть хорошо продуманной. Одни структуры

данных лучше подходят для реального конкурентного доступа к ним, другие — хуже, но сама идея остается неизменной: чем меньше защищенная область, тем меньше операций подвергается сериализации и тем выше возможность конкурентных обращений к данным.

Прежде чем обратиться к конструкциям структур данных, уделим немного времени рассмотрению ряда простых рекомендаций и узнаем, на что следует обратить внимание при разработке структур данных для конкурентного доступа.

6.1.1. Рекомендации по разработке структур данных для конкурентного доступа

Как уже упоминалось, при разработке структур данных, допускающих конкурентный доступ, следует обратить внимание на два аспекта: гарантию *безопасности* обращений к данным и *обеспечение* настоящего конкурентного доступа к данным. Основы обеспечения потокобезопасности структуры данных рассматривались в главе 3.

- Нужно гарантировать, что ни один поток не сможет столкнуться с нарушением инвариантов структуры данных из-за действий другого потока.
- Следует воспрепятствовать возникновению состояния гонки, присущего интерфейсу структуры данных, предоставив функцию для завершенных операций, а не для их поэтапного выполнения.
- Нужно обратить внимание на поведение структуры данных при наличии исключений и обеспечить соблюдение инвариантов.
- Следует свести к минимуму вероятность взаимной блокировки при использовании структуры данных, ограничив область действия блокировок и по возможности исключив вложенные блокировки.

Прежде чем обдумывать любое из этих предложений, важно вспомнить о тех ограничениях, которые желательно наложить на действия пользователей структуры данных: если один поток обращается к структуре данных через конкретную функцию, то какие функции можно безопасно вызвать из других потоков?

Это вопрос первостепенной важности. Как правило, конструкторам и деструкторам требуется исключительный доступ к структуре данных, но обеспечение ее недоступности до тех пор, пока конструктор не завершит свою работу, или после того, как будет запущен деструктор, возлагается на пользователя. Если в структуре данных поддерживаются операция присваивания, функция `swap()` или копирующий конструктор, то разработчик структуры должен решить, насколько они безопасны при одновременном вызове вместе с другими операциями и не требуют ли они от пользователя обеспечить им исключительный доступ, притом что большинство функций для работы со структурой данных можно вызвать из нескольких потоков одновременно без каких-либо проблем.

Второй по важности вопрос, требующий рассмотрения, — это возможность обеспечения действительно конкурентного доступа. Подробных рекомендаций на этот

счет у меня нет, вместо них есть перечень вопросов, на которые разработчик структуры данных должен ответить сам себе.

- Может ли область действия блокировок ограничиваться таким образом, чтобы некоторые части операции выполнялись за пределами блокировки?
- Можно ли разные части структуры данных защитить разными мьютексами?
- Всем ли операциям требуется одинаковый уровень защиты?
- Можно ли простым изменением структуры данных расширить возможности конкурентного доступа, не затрагивая при этом семантику операций?

В основе всех этих вопросов лежит одна идея: как свести к минимуму необходимый объем сериализации и обеспечить наивысший уровень истинной конкурентности. Зачастую структуры данных допускают конкурентный доступ нескольких потоков, которые просто считывают из них данные, а тот поток, который записывает в нее данные, должен иметь исключительный доступ. Такой режим доступа поддерживается за счет использования конструкций, подобных `std::shared_mutex`. Вскоре будет показано, что для структуры данных также зачастую характерны поддержка конкурентного доступа потоков, выполняющих разные операции, и сериализация доступа тех потоков, которые пытаются выполнять одну и ту же операцию.

В самой простой потокобезопасной структуре для защиты данных обычно используются мьютексы и блокировки. Как следует из материалов главы 3, с их помощью можно решить целый ряд проблем, к тому же довольно легко гарантировать, что в конкретный момент доступ к структуре данных получит только один поток. Чтобы облегчить вам разработку потокобезопасных структур данных, в этой главе основное внимание будет уделено именно структурам данных, основанным на применении блокировок, а разработке структур данных, допускающих конкурентные обращения и свободных от блокировок, будет посвящена глава 7.

6.2. Конкурентные структуры данных с блокировками

Разработка структур данных, допускающих конкурентные обращения и применяющих блокировки, сводится к обеспечению блокировки при обращении к данным нужных мьютексов и к тому, чтобы продолжительность удержания такой блокировки была минимальной. Когда структура защищена всего одним мьютексом, выполнить эту задачу нелегко. В главе 3 говорилось, что при этом нужно убедиться в невозможности доступа к данным вне защиты, обеспеченной блокировкой мьютекса, а также в отсутствие присущего интерфейсу состояния гонки. При использовании отдельных мьютексов для защиты отдельных частей структуры данных проблем еще больше, поскольку возникает также вероятность взаимных блокировок, если операциям над структурой данных требуется блокировка более чем одного мьютекса. Теперь конструкцию структуры данных с несколькими мьютексами приходится продумывать еще тщательнее, чем конструкцию с одним мьютексом.

В этом разделе для разработки нескольких простых структур данных с использованием для их защиты мьютексов и блокировок будут применяться рекомендации из раздела 6.1.1. В каждом случае будут изыскиваться возможности повышения уровня конкурентного доступа без ущерба для потокобезопасности структуры данных.

Начнем с рассмотрения реализации стека из главы 3, так как это одна из самых простых структур данных, в которой задействуется всего один мьютекс. Потокобезопасна ли она? И насколько далека от достижения истинной конкурентности?

6.2.1. Потокобезопасный стек, использующий блокировки

Код потокобезопасного стека из главы 3 воспроизведен в листинге 6.1. Замысел состоял в написании потокобезопасной структуры данных, похожей на `std::stack<>`, которая поддерживает помещение элементов данных в стек и их извлечение из него.

Листинг 6.1. Определение класса для 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));     ❶
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();    ❷
        std::shared_ptr<T> const res(
          std::make_shared<T>(std::move(data.top())));  ❸
        data.pop();  ❹
        return res;
    }
```

```
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=std::move(data.top());        ❺
        data.pop();                         ❻
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

Посмотрим, в какой степени в этом коде применяются изложенные ранее рекомендации.

Во-первых, можно увидеть, что элементарная потокобезопасность обеспечивается защитой каждой компонентной функции блокировкой мьютекса `m`. Тем самым обеспечивается возможность доступа к данным в конкретный момент времени со стороны только одного потока. Поэтому, если каждой компонентной функцией поддерживаются инварианты, увидеть нарушение какого-либо из них не сможет ни один поток.

Во-вторых, существует вероятность состояния гонки между функцией `empty()` и любой из функций `pop()`, но, поскольку код выполняет явную проверку на пустоту стека при удержании блокировки в функции `pop()`, состояние гонки проблем не вызывает. За счет непосредственного возвращения извлеченного элемента данных в виде части вызова функции `pop()` удается избежать возможного состояния гонки, которое могло бы сложиться при использовании отдельных компонентных функций `top()` и `pop()` наподобие тех, что имеются в определении класса `std::stack<>`.

В-третьих, существует несколько возможных источников исключений. Блокировка мьютекса может выдать исключение, но подобное случается крайне редко, свидетельствуя о проблемах с мьютексом или нехватке системных ресурсов, и к тому же является самой первой операцией в каждой компонентной функции. Поскольку при этом изменения в данные не вносятся, опасности не возникает. Снятие блокировки с мьютекса не может вызвать сбой, поэтому считается абсолютно безопасной операцией, а применение `std::lock_guard<>` гарантирует, что мьютекс никогда не останется заблокированным.

При вызове `data.push()` ❶ исключение может быть выдано, либо если выдача исключения произошла при копировании или перемещении значения данных, либо если для расширения структуры данных, в которой они хранятся, не хватает свободной распределяемой памяти. В любом случае `std::stack<>` гарантирует безопасность, поэтому проблем не возникнет.

В первом переопределении функции `pop()` исключение `empty_stack` может выдать сам код ❷, но это не повлечет за собой никаких изменений и операция будет безопасной. Исключение может быть выдано и при создании объекта `res` ❸ по двум причинам: исключение, выданное `std::make_shared` из-за невозможности распределения памяти под новый объект и внутренние данные, необходимые для

подсчета ссылок, или же исключение, выданное конструктором копирования или конструктором перемещения элемента возвращаемых данных при копировании или перемещении в только что распределенную область памяти. В обоих случаях среда выполнения C++ и стандартная библиотека гарантируют отсутствие утечек памяти и корректное уничтожение нового объекта, если он будет создан. Поскольку стек, где хранятся данные, пока еще не был изменен, волноваться не о чем. Вызов `data.pop()` ❹ не выдаст исключений и не возвратит результат, поэтому переопределение функции `pop()` в отношении выдачи исключений опасности не представляет.

Второе переопределение функции `pop()` практически аналогично первому, за исключением того, что на этот раз исключение может быть выдано оператором присваивания копированием или оператором присваивания перемещением ❺, а не конструктором нового объекта и экземпляром `std::shared_ptr`. И в этом случае до вызова `data.pop()` ❻ в структуру данных не вносится никаких изменений, что по-прежнему гарантирует невыдачу исключений, следовательно, это переопределение в отношении выдачи исключений никакой опасности не представляет.

Наконец, функция `empty()` не изменяет никаких данных и также безопасна в отношении выдачи исключений.

Здесь есть два возможных источника возникновения взаимной блокировки из-за вызова пользовательского кода при удержании блокировки: конструктор копирования или конструктор перемещения (❶, ❸) и оператор присваивания копированием или присваивания перемещением ❺, применяемые к содержащимся в структуре элементам данных. Если эти функции либо вызывают компонентные функции в отношении стека, в который добавляется или из которого извлекается элемент, либо требуют блокировки любого типа, а при вызове компонентной функции стека удерживалась другая блокировка, появляется возможность взаимоблокировки. Но возложить ответственность за предотвращение подобных случаев лучше на пользователей стека, поскольку неразумно было бы ожидать, что добавление элемента в стек или его извлечение из стека обойдется без его копирования или распределения памяти под него.

Поскольку для защиты данных всеми компонентными функциями используется `std::lock_guard<>`, компонентные функции стека можно безопасно вызывать из любого количества потоков. Единственными небезопасными компонентными функциями являются конструкторы и деструкторы, но это не проблема, поскольку объект может быть и сконструирован, и уничтожен только один раз. Вызов компонентных функций для не до конца сконструированного или частично уничтоженного объекта ни к чему хорошему не приводит независимо от того, насколько правильно он завершится. Следовательно, пользователь должен гарантировать, что у других потоков не будет возможности обратиться к стеку, пока он не будет полностью сконструирован, и что все потоки станут порождать доступ к стеку до его уничтожения.

Хотя одновременный вызов компонентных функций из нескольких потоков благодаря использованию блокировок считается безопасным, в конкретный момент времени со структурой данных работает только один поток. Если наблюдается серьезная конкуренция при доступе к стеку, *сериализация* потоков может снизить производительность приложения, поскольку за время ожидания снятия блокировки

поток не выполняет никакой полезной работы. Кроме того, стек не предоставляет каких-либо средств ожидания добавляемого элемента, поэтому, если потоку приходится его дожидаться, он должен периодически вызывать функцию `empty()` или вызывать функцию `pop()` и перехватывать исключение `empty_stack`. Если понадобится такой сценарий, то данную реализацию стека удачной не назовешь: тут или ожидающий поток будет потреблять ценные ресурсы, или пользователь обязан будет создать внешний код ожидания и уведомления, используя, к примеру, условные переменные, что сделает внутреннюю блокировку ненужной и неоправданно расточительной. Способ включения этого ожидания в саму структуру данных с применением условной переменной был показан на примере очереди из главы 4, поэтому его и рассмотрим.

6.2.2. Потокобезопасная очередь с использованием блокировок и условных переменных

Потокобезопасная очередь из главы 4 воспроизведена в листинге 6.2. Стек был построен по образцу `std::stack<>`, а очередь — по образцу `std::queue<>`. И тут опять интерфейс отличается от стандартного адаптера контейнера из-за ограничений, связанных с созданием структуры данных, безопасной для конкурентного доступа к ней со стороны сразу нескольких потоков.

Листинг 6.2. Полное определение класса для потокобезопасной очереди с использованием условных переменных

```
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(new_value));
        data_cond.notify_one();                    ❶
    }
    void wait_and_pop(T& value)                    ❷
    {
        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()              ❸
```

```
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();}); ◄—❹
        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>(); ◄—❺
        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();
    }
};
```

Структура реализации очереди, показанная с листинге 6.2, похожа на стек из листинга 6.1, за исключением вызова `data_cond.notify_one()` в `push()` ❶ и функции `wait_and_pop()` (❷ и ❸). Два переопределения `try_pop()` почти идентичны функциям `pop()` из листинга 6.1, но здесь они не выдают исключение, если очередь пуста. Вместо этого они возвращают либо `bool`-значение, показывающее, было ли извлечено значение, либо в случае с переопределением, возвращающим указатель ❺, — `NULL`-указатель, если нечего извлекать. Точно так же можно было бы реализовать и стек. Если исключить функции `wait_and_pop()`, то анализ, проделанный для стека, применим и здесь.

Новые функции `wait_and_pop()` позволяют решить проблему ожидания поступления значения в очередь, с которой мы уже сталкивались, когда рассматривали стек: вместо периодического вызова `empty()` ожидающий поток может вызвать `wait_and_pop()`, а структура данных обработает ожидание с помощью условной переменной. Возвращения из вызова `data_cond.wait()` не будет, пока очередь содержит хотя бы один элемент, поэтому в данном месте программы беспокоиться за вероятность пустой очереди не придется и данные по-прежнему защищены блокировкой

мьютекса. Следовательно, эти функции не добавляют новых состояний гонки или вероятности взаимной блокировки, обеспечивая сохранность инвариантов.

А вот относительно безопасности при выдаче исключений есть неприятный нюанс: если в состоянии ожидания находится более одного потока, то при поступлении элемента в очередь на вызов `data_cond.notify_one()` среагирует только один поток. Но если затем этот поток выдаст исключение в `wait_and_pop()`, например при конструировании нового объекта `std::shared_ptr<>` ❹, то другие потоки разбужены не будут. Если это неприемлемо, данный вызов можно легко заменить вызовом `data_cond.notify_all()`, который разбудит все потоки, но большинство из них тут же снова впадет в спячку, как только обнаружится, что очередь опять опустела. Второй вариант заключается в наличии в `wait_and_pop()` на случай выдачи исключения вызова `notify_one()`, что позволит другому потоку попытаться извлечь сохраненное значение. Третий вариант предусматривает перемещение инициализации `std::shared_ptr<>` в вызов `push()` и сохранение экземпляров `std::shared_ptr<>`, а не непосредственных значений данных. Тогда копирование `std::shared_ptr<>` вне внутренней очереди `std::queue<>` не сможет выдать исключение и `wait_and_pop()` снова станет безопасной. В листинге 6.3 показана реализация очереди, переработанная с учетом этих соображений.

Листинг 6.3. Потокобезопасная очередь, содержащая экземпляры 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());  <-- ❶
        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());  <-- ❷
        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();  <-- ❸
```

```
        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(); ❹
        data_queue.pop();
        return res;
    }
    void push(T new_value)
    {
        std::shared_ptr<T> data(
            std::make_shared<T>(std::move(new_value))); ❺
        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();
    }
};
```

Основные последствия хранения данных, обернутых в `std::shared_ptr<>`, просты и понятны: `pop`-функции, принимающие ссылку на переменную для получения нового значения, должны теперь разыменовать сохраненный указатель (❶ и ❷), а `pop`-функции, возвращающие экземпляр `std::shared_ptr<>`, могут извлечь его из очереди (❸ и ❹), перед возвращением вызывающему коду.

В сохранении данных с помощью `std::shared_ptr<>` есть еще одно преимущество: распределение памяти под новый экземпляр теперь может проводиться за пределами блокировки в функции `push()` ❺, а в коде листинга 6.2 его приходилось выполнять при удержании блокировки в функции `pop()`. Поскольку обычно операция распределения памяти сопряжена с немалыми затратами, это обстоятельство может пойти на пользу производительности очереди за счет сокращения времени удержания мьютекса, что позволит другим потокам в высвободившемся промежутке времени выполнять операции над очередью.

Точно так же, как и в примере со стеком, использование мьютекса для защиты всей структуры данных ограничивает поддержку этой очередью конкурентности. Хотя на очереди в различных компонентных функциях могут быть заблокированы сразу несколько потоков, в конкретный момент выполнять какую-либо работу может только один из них. Отчасти это ограничение обусловлено тем, что в реализации задействуется `std::queue<>`: теперь за счет применения стандартного контейнера вы располагаете одним элементом данных, который либо защищен, либо нет. Если взять под контроль подробности реализации структуры данных, можно обеспечить более подробную детализацию блокировки и поднять возможности конкурентности на более высокий уровень.

6.2.3. Потокобезопасная очередь с использованием подробной детализации блокировок и условных переменных

В коде листингов 6.2 и 6.3 мы имели дело с одним защищенным элементом данных (`data_queue`), а следовательно, с одним мьютексом. Чтобы воспользоваться более подробной детализацией блокировки, нужно проникнуть в саму очередь, в ее составляющие, и связать по одному мьютексу с каждым отдельно взятым элементом данных.

Самой простой структурой данных для очереди является односвязный список (рис. 6.1). Очередь состоит из `head` — указателя на первый элемент списка, и каждый элемент в ней указывает на следующий элемент. Элементы данных удаляются из очереди путем замены значения в указателе `head` указателем на следующий элемент с последующим возвращением данных, на которые ссылалось прежнее значение `head`.

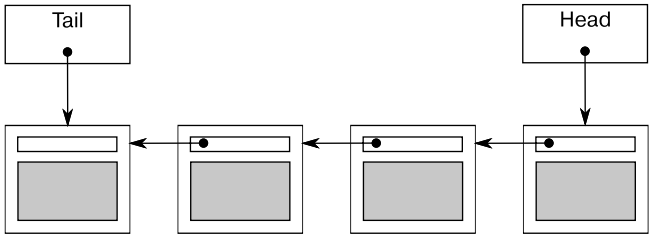


Рис. 6.1. Очередь, использующая односвязный список

Элементы добавляются в очередь с другого конца. Для этого у нее есть указатель `tail`, ссылающийся на последний элемент списка. Новые узлы добавляются изменением значения указателя `next` таким образом, чтобы он указывал на новый узел, и последующим обновлением указателя `tail`, чтобы он указывал на новый элемент. Когда список пуст, указатели `head` и `tail` имеют значение `NULL`.

В листинге 6.4 показана простая реализация очереди, созданной на основе урезанной версии интерфейса к очереди из листинга 6.2. В ней есть только одна функция `try_pop()` и нет функции `wait_and_pop()`, поскольку эта очередь поддерживает только однопоточное использование.

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;  ❶
    node* tail;                  ❷
public:
    queue(): tail(nullptr)
    {}
    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);  ❸
        if(!head)
            tail=nullptr;
        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);  ❹
        }
        else
        {
            head=std::move(p);  ❺
        }
        tail=new_tail;  ❻
    }
};
```

Прежде всего заметьте, что в коде листинга 6.4 для управления узлами задействуются указатели `std::unique_ptr<node>`, тем самым гарантируется, что они (и данные, на которые они ссылаются) будут удаляться, как только станут не нужны, и явно использовать функцию `delete` не потребуется. Цепочка владения управляется с `head`, а вот `tail` — всего лишь указатель на последний узел, так как он должен ссылаться на узел, уже принадлежащий `std::unique_ptr <node>`.

В однопоточном контексте эта реализация работает неплохо, однако попытка использовать более подробную детализацию в многопоточной среде выявит два проблемных места. При наличии двух элементов данных, `head` ❶ и `tail` ❷, можно воспользоваться двумя мьютексами, по одному для защиты `head` и `tail`, но тут как раз и проявятся две проблемы.

Наиболее явная будет выражаться в том, что функция `push()` может изменять как `head` ❺, так и `tail` ❼, следовательно, ей придется заблокировать оба мьютекса. Проблема, конечно, неприятная, но разрешимая, поскольку заблокировать оба мьютекса можно. А главная проблема состоит в том, что обе функции, `push()` и `pop()`, обращаются к имеющемуся в этом узле указателю `next`: `push()` обновляет `tail->next` ❹, а `try_pop()` считывает `head->next` ❸. Если в очереди всего один элемент, то `head==tail`, следовательно, `head->next` и `tail->next` — один и тот же объект, нуждающийся в защите. Поскольку, не прочитав `head` и `tail`, понять, что это один и тот же объект, невозможно, теперь нужно заблокировать один и тот же мьютекс как в `push()`, так и в `try_pop()`. Следовательно, ситуация не улучшилась. Можно ли решить эту дилемму?

Обеспечение конкурентности за счет разделения данных

Проблему можно решить, заранее разместив в очереди пустой фиктивный узел и тем самым обеспечив постоянное наличие в ней хотя бы одного узла, чтобы отделить узел, к которому обращаются в голове, от узла, к которому обращаются в хвосте. Теперь при пустой очереди как `head`, так и `tail` указывают на фиктивный узел и не содержат значение `NULL`. Это нас вполне устраивает, поскольку `try_pop()` не обращается к `head->next`, когда очередь пуста. Если к очереди добавляется узел (то есть в ней появляется один настоящий узел), `head` и `tail` указывают на разные узлы, поэтому состояния гонки при обращениях `head->next` и `tail->next` не возникает. Недостатком такого решения является то, что применение фиктивных узлов требует добавления дополнительного уровня косвенности для хранения данных по указателю. Новая реализация показана в листинге 6.5.

Листинг 6.5. Простая очередь с фиктивным узлом

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;   ←❶
        std::unique_ptr<node> next;
    };
    std::unique_ptr<node> head;
    node* tail;
public:
    queue():
        head(new node),tail(head.get()) ←❷
    {}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(head.get()==tail) ←❸
        {
```

```
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(head->data);    ←❹
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);      ←❺
        return res;                  ←❻
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));  ←❼
        std::unique_ptr<node> p(new node);  ←❽
        tail->data=new_data;             ←❾
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
};
```

В `try_pop()` внесены минимальные изменения. Во-первых, `head` сравнивается с `tail` ❸, а не со значением `NULL`, так как присутствие фиктивного узла означает, что у `head` никогда не будет значения `NULL`. Поскольку `head` относится к указателю `std::unique_ptr<node>`, для выполнения операции сравнения следует вызвать `head.get()`. Во-вторых, так как в `node` теперь хранится указатель на данные ❶, этот указатель можно извлечь напрямую ❹, избавившись от необходимости создавать новый экземпляр `T`. А вот функция `push()` изменилась довольно сильно. Сначала нужно создать в куче новый экземпляр `T` и получить его в собственность в `std::shared_ptr<>` ❼ (обратите внимание на использование `std::make_shared` для того, чтобы исключить издержки от повторного распределения памяти под счетчик ссылок). Созданный новый узел станет новым фиктивным узлом, поэтому предоставлять `new_value` конструктору ❽ не нужно. Вместо этого для старого фиктивного узла устанавливаются данные только что распределенной копии `new_value` ❾. И наконец, чтобы существовал фиктивный узел, его нужно создать в конструкторе ❷.

Наверняка вас интересует, что дают эти изменения и как они помогают сделать очередь потокобезопасной. Итак, `push()` теперь обращается только к `tail`, но не к `head`, что уже хорошо. Функция `try_pop()` обращается как к `head`, так и к `tail`, но `tail` требуется только для начального сравнения, следовательно, блокировка удерживается недолго. Главный выигрыш состоит в том, что при наличии фиктивного узла `try_pop()` и `push()` никогда не работают с одним и тем же узлом, следовательно, мьютекс, охватывающий всю очередь, больше не нужен. Можно иметь один мьютекс для `head` и один для `tail`. А куда же ставятся блокировки?

Наша цель — максимально раскрыть возможности для конкурентности, следовательно, продолжительность удержания блокировок хотелось бы свести к минимуму. С функцией `push()` все просто: мьютекс должен быть заблокирован при всех обращениях к `tail`, то есть он блокируется после распределения памяти под новый узел ❽ и до присваивания данных текущему хвостовому узлу ❾. Затем блокировку нужно удерживать, пока функция не будет выполнена до конца.

С функцией `try_pop()` дела обстоят сложнее. Сперва нужно заблокировать мьютекс на обращении к `head` и удерживать его, пока работа с `head` не будет закончена. Этим мьютексом определяется, какой поток извлекает данные, поэтому блокировку требуется выполнить в первую очередь. После изменения значения `head` ❺ мьютекс можно разблокировать: в момент возвращения результата ❻ он уже не нужен. Остается только выяснить, необходима ли блокировка хвостового мьютекса при обращении к `tail`. Поскольку обращаться к `tail` придется всего один раз, мьютекс можно заблокировать только на время чтения. Лучше всего сделать это, заключив его в функцию. Так как код, которому требуется мьютекс, заблокированный на обращении к `head`, фактически является всего лишь частью компонентного кода, понятно, что и его тоже лучше заключить в функцию. Окончательная версия кода показана в листинге 6.6.

Листинг 6.6. Потокобезопасная очередь с более подробной детализацией блокировок

```
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;
    }
};
```

Оценим этот код с точки зрения его соответствия рекомендациям, приведенным в подразделе 6.1.1. Прежде чем выискивать нарушения инвариантов, надо определить, что они сводятся к следующим формулировкам:

- `tail->next==nullptr`;
- `tail->data==nullptr`;
- `head==tail` означает пустой список;
- у списка с одним элементом `head->next==tail`;
- для каждого имеющегося в списке узла `x`, где `x!=tail`, `x->data` указывает на экземпляр `T`, а `x->next` — на следующий узел в списке. Если `x->next==tail`, то `x` — последний узел списка;
- следуя по `next`-узлам от `head`, в конечном счете можно оказаться в `tail`.

Сама по себе функция `push()` проста и понятна: мьютекс `tail_mutex` защищает только изменения в структуре данных, и инвариант ими сохраняется, поскольку новый хвостовой узел является пустым, а указатели `data` и `next` для старого хвостового узла, который теперь является последним настоящим узлом списка, установлены правильно.

Все самое интересное находится в функции `try_pop()`. Оказывается, блокировка мьютекса `tail_mutex` нужна не только для защиты чтения самого указателя `tail`, но и для гарантии того, что состояние гонки не возникнет при чтении данных из `head`. При отсутствии этого мьютекса высока вероятность того, что одновременно одним потоком будет вызвана функция `try_pop()`, а другим — функция `push()`, а порядок выполнения их операций окажется не определен. Хотя блокировка мьютекса удерживается каждой компонентной функцией или удерживаются *разные* мьютексы, у них появляется возможность обращения к одним и тем же данным, ведь все данные в очереди появляются благодаря вызову функции `push()`. Из-за того что потоки могут обращаться к одним и тем же данным, не определяя конкретный порядок, возникнет, как было показано в главе 5, состояние гонки, сопряженное с неопределенным поведением. Благодаря блокировке мьютекса `tail_mutex` в функции

get_tail() решаются все проблемы. Поскольку вызов get_tail() блокирует тот же самый мьютекс, который блокирует и вызов push(), между двумя вызовами определяется порядок. Вызов get_tail() происходит либо до вызова push(), и в таком случае код этой функции видит старое значение tail, либо после вызова push(), и в таком случае код видит новое значение tail и новые данные, прикрепленные к предыдущему значению tail.

Важно также, что вызов get_tail() происходит во время блокировки мьютекса head_mutex. В противном случае вызов pop_head() мог бы застрять между вызовом get_tail() и блокировкой мьютекса head_mutex, из-за того что другие потоки вызвали try_pop() (а следовательно, и pop_head()) и захватили блокировку первыми, не давая выполнять работу исходному потоку:

```
std::unique_ptr<node> pop_head()                    ← Неправильная реализация
{
    node* const old_tail=get_tail();                ← Получение старого значения tail
    std::lock_guard<std::mutex> head_lock(head_mutex);   вне блокировки head_mutex
                                                         ❶
    if(head.get()==old_tail)                        ←❷
    {
        return nullptr;
    }
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next);                 ←┐
    return old_head;                                  ❸
}
```

В этом неправильном сценарии, где вызов get_tail() ❶ выполнен вне области действия блокировки, может обнаружиться, что и head, и tail изменились к тому времени, когда ваш исходный поток смог заблокировать мьютекс head_mutex, и возвращенный узел tail не только перестал быть хвостовым, но даже уже не является частью списка. Как следствие, сравнение head с old_tail ❷ даст сбой, даже если head является последним узлом. Поэтому при обновлении head ❸ у вас в итоге может получиться, что head переместится за tail и окажется за концом списка, разрушив структуру данных. В правильной реализации из листинга 6.6 вызов get_tail() выполняется под блокировкой мьютекса head_mutex. Тем самым гарантируется, что никакие другие потоки не могут изменить head, и tail будет только продвигаться дальше по списку по мере добавления новых узлов в вызовах функции push(), соблюдая абсолютную безопасность. Указатель head никогда не сможет перейти за значение, возвращенное вызовом get_tail(), поэтому инварианты не будут нарушены.

Как только за счет обновления head функция pop_head() удалит узел из очереди, мьютекс будет разблокирован и try_pop() сможет извлечь данные и удалить узел, если он существовал (или возвратить экземпляр std::shared_ptr<> со значением NULL, если его не было), соблюдая безопасность в расчете на то, что доступ к данному узлу можно получить лишь из одного потока.

Внешний интерфейс является подмножеством интерфейса из листинга 6.2, следовательно, к нему применим прежний анализ: у него нет склонности к состоянию гонки.

Интереснее будет разобраться с исключениями. Поскольку схемы распределения памяти под данные изменились, исключения теперь могут выдаваться в разных местах. Единственными операциями в `try_pop()`, способными выдавать исключения, являются блокировки мьютексов, и данные не изменяются, пока блокировки не будут установлены. Поэтому в отношении выдачи исключений функция `try_pop()` безопасна. В то же время функция `push()` распределяет в куче память под новые экземпляры `T` и `node`, и при каждом из этих распределений может быть выдано исключение. Но оба объекта, под которые только что была распределена память, присваиваются интеллектуальным указателям, и при выдаче исключения эта память будет освобождена. После установки блокировки никакие другие операции в `push()` не могут выдать исключение, поэтому вам опять не о чем беспокоиться и функция `push()` в отношении выдачи исключений также безопасна.

Поскольку изменения в интерфейс не вносились, новые внешние возможности для взаимной блокировки не возникали. Нет и никаких внутренних возможностей. Единственным местом установки двух блокировок является функция `pop_head()`, которая всегда блокирует `head_mutex`, а затем `tail_mutex`, поэтому взаимная блокировка никогда не возникает.

Остаются только вопросы, касающиеся возможности конкурентности. У этой структуры данных существенно более широкие возможности предоставления такого доступа, чем у структуры из листинга 6.2, поскольку у блокировок более подробная детализация и гораздо больше работы выполняется за их пределами. Например, в функции `push()` распределение памяти под новый узел и новый элемент данных происходит без удержания блокировки. Следовательно, его могут без проблем выполнять одновременно несколько потоков. В конкретный момент времени добавить свой новый узел в список может только один поток, но код для этого представляет собой всего лишь несколько присваиваний значений указателям, поэтому блокировка удерживается совсем недолго, если сравнивать с реализацией на основе применения `std::queue<>`, где она удерживается на протяжении всех операций распределения памяти внутри `std::queue<>`.

К тому же функция `try_pop()` удерживает блокировку `tail_mutex` только на время, необходимое для защиты считывания данных из указателя `tail`. Следовательно, почти все, что делается внутри вызова `try_pop()`, может происходить одновременно с вызовом функции `push()`. Кроме того, при удержании блокировки `head_mutex` выполняется минимальный объем операций, затратная операция `delete` (в деструкторе указателя `node`) протекает вне блокировки. За счет этого будет увеличиваться количество возможных конкурентных вызовов функции `try_pop()`: в конкретный момент вызвать функцию `pop_head()` может только один поток, но затем удалить свои старые узлы и безопасно возвратить данные могут сразу несколько потоков.

Ожидание поступления элемента

Итак, код в листинге 6.6 предоставляет потокобезопасную очередь с более подробной детализацией блокировок, но он поддерживает только функцию `try_pop()` без каких-либо ее переопределений. А как насчет поддержки весьма удобных функций `wait_and_pop()` из листинга 6.2? Можно ли реализовать идентичный интерфейс с применением более подробной детализации блокировок?

Ответ положительный, но возникает вопрос, как это сделать. Внести изменения в `push()` не составит труда: нужно лишь, как в листинге 6.2, добавить в конец функции вызов `data_cond.notify_one()`. Но не все так просто. Из-за необходимости получить максимум возможностей конкурентности к структуре данных мы воспользовались более подробной детализацией блокировок. Если блокировка мьютекса будет удерживаться в течение всего времени вызова `notify_one()`, как в листинге 6.2, то при пробуждении уведомляемого потока до снятия блокировки с мьютекса ему придется дожидаться этого снятия. В то же время если снять блокировку мьютекса *до* вызова `notify_one()`, то ожидающий поток может заблокировать этот мьютекс сразу же после пробуждения (при условии, что его не опередит какой-нибудь другой поток). Усовершенствование получится незначительным, но в некоторых случаях оно может сыграть весьма важную роль.

С функцией `wait_and_pop()` дело обстоит несколько сложнее, поскольку нужно решить, где будет ожидание, каким — предикат и какой мьютекс следует заблокировать. Будет ожидаться выполнение условия «очередь не пуста», которое можно представить в виде выражения `head!=tail`. При такой формулировке условия потребуется блокировка обоих мьютексов, `head_mutex` и `tail_mutex`, но при рассмотрении кода листинга 6.6 уже было решено, что блокировать `tail_mutex` нужно не для сравнения как такового, а только для чтения `tail`, поэтому аналогичную логику можно применить и здесь. Если создать предикат вида `head!=get_tail()`, потребуется только удержать блокировку мьютекса `head_mutex`, и ею можно воспользоваться для вызова `data_cond.wait()`. После добавления логики ожидания вся остальная реализация будет такой же, как и у функции `try_pop()`.

Второе переопределение `try_pop()` и соответствующее переопределение `wait_and_pop()` нужно будет тщательно продумать. Если заменить возвращение указателя `std::shared_ptr<>`, извлеченного из `old_head`, присваиванием методом копирования параметра `value`, то возможны проблемы с безопасностью выдачи исключений. К этому моменту элемент данных уже был удален из очереди и мьютекс разблокирован, осталось только возвратить данные вызывающему коду. Но если присваивание копированием выдаст исключение (что вполне возможно), элемент данных будет утрачен, поскольку вернуть его в то же место в очереди уже не получится.

Если фактический тип `T`, используемый в качестве аргумента шаблона, обладает не выдающим исключений оператором присваивания методом перемещения или не выдающей исключений операцией обмена (swap), то можно воспользоваться и этим вариантом, но предпочтительнее было бы обобщенное решение, подходящее для любого типа `T`. В данном случае, прежде чем удалять узел из списка, придется перемещать потенциально способную на выдачу исключения операцию в область действия блокировки.

Значит, понадобится еще одно переопределение `pop_head()`, извлекающее сохраненное значение для внесения изменения в список.

А вот функция `empty()` сравнительно проста: блокировка `head_mutex` и проверка условия `head==get_tail()` (см. код листинга 6.10). Код очереди в окончательном виде показан в листингах 6.7–6.10.

Листинг 6.7. Потокобезопасная очередь с блокировками и ожиданием: внутренний код 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);
    bool empty();
};
```

Помещение в очередь новых узлов выполняется довольно просто — реализация, показанная в листинге 6.8, близка к той, что была приведена ранее.

Листинг 6.8. Потокобезопасная очередь с блокировками и ожиданием: помещение в очередь 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 упоминалось, вся сложность сосредоточена в `pop`-компоненте, использующем для упрощения ряд вспомогательных функций. В листинге 6.9 показана реализация функции `wait_and_pop()` и связанных с ней вспомогательных функций.

Листинг 6.9. Потокобезопасная очередь с блокировками 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()  ←❶
    {
        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() ←❷
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
        return std::move(head_lock);            ←❸
    }
    std::unique_ptr<node> wait_pop_head()
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data()); ←❹
        return pop_head();
    }
    std::unique_ptr<node> wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data()); ←❺
        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);
    }
}
```

Реализация pop-компонента, показанная в листинге 6.9, содержит несколько небольших вспомогательных функций, упрощающих код и устраняющих возможное дублирование: функцию `pop_head()` ❶, изменяющую список для удаления элемента `head`, и функцию `wait_for_data()` ❷, ожидающую появления в очереди каких-либо извлекаемых данных. Функция `wait_for_data()` заслуживает особого внимания, поскольку в ней не только задается ожидание на условной переменной с применением в качестве предиката лямбда-функции, но и возвращается экземпляр блокировки вызывающему коду ❸. Тем самым гарантируется, что при изменении данных соот-

ветствующим переопределением функции `wait_pop_head()` в строках кода ❹ и ❺ удерживается та же самая блокировка. Кроме того, функция `pop_head()` повторно используется в коде функции `try_pop()` (листинг 6.10).

Листинг 6.10. Потокобезопасная очередь с блокировками 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;
    }
    bool empty()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        return (head.get()==get_tail());
    }
};
```

Данная реализация очереди послужит основой для рассматриваемой в главе 7 очереди без применения блокировок. Речь идет о *неограниченной* очереди, в которую потоки могут и далее помещать новые значения, пока не закончится свободная память, даже если значения не будут удаляться. Альтернативой ей служит *ограниченная* очередь с фиксированной при ее создании максимальной длиной. Как только ограниченная очередь заполнится, попытки помещения в нее дополнительных элементов

либо будут давать сбой, либо станут блокироваться, пока при извлечении элемента не освободится место. Ограниченные очереди пригодятся для равномерного распределения работы между потоками на основе выполняемых задач (см. главу 8). Это не даст потоку или потокам, наполняющим очередь, сильно опережать работу потока или потоков, считывающих элементы из очереди.

Показанную в книге неограниченную очередь можно легко превратить в очередь ограниченной длины, воспользовавшись ожиданием на условной переменной в функции `push()`. Вместо ожидания появления в очереди элементов, как это сделано в функции `pop()`, придется ждать, когда количество элементов в ней станет меньше максимально допустимого. Дальнейшие рассуждения по поводу ограниченных очередей выходят за рамки книги, поэтому оставим эту тему и перейдем к более сложным структурам данных.

6.3. Разработка более сложных структур данных с использованием блокировок

Стеки и очереди не отличаются особой сложностью: их интерфейс крайне прост, а предназначение — узконаправленное. Но подобной простотой отличаются далеко не все структуры данных, большинство из них поддерживает весьма широкий диапазон операций. В принципе, это может расширить возможность конкурентности, но при этом значительно усложнится защита данных, поскольку нужно будет учесть схемы множественного доступа к ним. При разработке конкурентных структур данных важно учесть характер выполняемых операций.

Чтобы понять суть возникающих проблем, рассмотрим структуру поисковой таблицы.

6.3.1. Создание потокобезопасной поисковой таблицы с использованием блокировок

В поисковой таблице, или в словаре, значения одного типа (ключевого) связаны со значениями того же или другого типа (отображаемого). В общем, суть такой структуры заключается в том, чтобы позволить коду запрашивать данные, связанные с заданным ключом. В стандартной библиотеке C++ такая возможность предоставляется с помощью ассоциативных контейнеров `std::map<>`, `std::multimap<>`, `std::unordered_map<>` и `std::unordered_multimap<>`.

Принцип использования поисковой таблицы не такой, как у стека или очереди, где практически каждая операция изменяет их тем или иным образом, либо добавляя, либо удаляя элемент. Поисковая таблица может изменяться крайне редко. Одним из примеров такого сценария служит простой DNS-кэш из листинга 3.13, который представляется сильно усеченным интерфейсом по сравнению с `std::map<>`. Как было показано на примере стека и очереди, интерфейсы стандартных контейнеров не подходят для тех случаев, когда от структуры данных требуется обеспечить конкурентность сразу нескольким потокам. Конструкциям их интерфейсов присущи состояния гонки, поэтому их нужно урезать и пересматривать.

Самой серьезной проблемой интерфейса `std::map<>` с точки зрения конкурентности являются итераторы. Хотя можно обзавестись итератором, предоставляющим безопасный доступ к контейнеру даже при условии, что другие потоки могут обращаться к контейнеру и вносить в него изменения, легкой эту задачу не назовешь. Корректно управляемые итераторы требуют умения справиться, к примеру, с такой проблемой, когда другой поток удаляет элемент, на который ссылается итератор. Ее решение может оказаться довольно сложным. На первый взгляд, создавая интерфейс потокобезопасной поисковой таблицы, итераторы нужно выбросить. При условии весьма серьезной привязки интерфейса `std::map<>` и других ассоциативных контейнеров стандартной библиотеки к итераторам лучше будет, наверное, отставить их и разработать интерфейс с нуля.

Для поисковой таблицы понадобится всего несколько основных операций.

- Добавление новой пары «ключ — значение».
- Изменение значения, связанного с заданным ключом.
- Удаление ключа и связанного с ним значения.
- Получение значения, связанного с заданным ключом, если таковой имеется.

Есть также несколько операций, выполняемых над всем контейнером, из которых можно извлечь пользу, например проверка контейнера на пустоту, получение мгновенного образа полного списка ключей или полного набора пар «ключ — значение».

Если придерживаться простых рекомендаций по обеспечению потокобезопасности, например не возвращать ссылки и накладывать простую блокировку мьютекса целиком на всю компонентную функцию, то все эти операции будут безопасны: они станут выполняться либо до изменения, реализуемого другим потоком, либо после него. Наибольшая вероятность состояния гонки возникает при добавлении новой пары «ключ — значение»: если два потока добавляют новое значение, то только один из них будет первым, следовательно, второй со своей задачей не справится. В качестве одного из решений можно свести добавление и изменение в одну компонентную функцию, как было сделано для DNS-кэша в коде листинга 3.13.

С точки зрения интерфейса остается лишь один интересный момент, касающийся получения связанного значения, где о наличии ключа говорится: «Если таковой имеется». Один из вариантов — позволить пользователю предоставить результат по умолчанию, возвращаемый при отсутствии ключа:

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

В таком случае, если значение по умолчанию `default_value` не было предоставлено явным образом, можно воспользоваться созданным заранее для случая по умолчанию экземпляром `mapped_type`. Можно пойти еще дальше и возвратить вместо простого экземпляра `mapped_type` объект `std::pair<mapped_type,bool>`, где `bool` показывает, было ли значение. Еще один вариант — возвратить интеллектуальный указатель на значение: если значение указателя `NULL`, то значения для возвращения не было.

Как уже упоминалось, после принятия решения о конструкции интерфейса (при условии, что интерфейсу не свойственно состояние гонки) потокобезопасность можно гарантировать использованием единственного мьютекса и простой блокировки,

устанавливаемой в каждой компонентной функции для защиты исходной структуры данных. Но тем самым будут утеряны предоставляемые отдельными функциями возможности конкурентности для чтения и изменения структуры данных. Один из вариантов решения этой проблемы заключается в применении мьютекса, поддерживающего несколько читающих потоков или один записывающий поток, например `std::shared_mutex`, который задействовался в коде листинга 3.13. Разумеется, такой подход расширит возможности конкурентности, но изменять структуру данных в конкретный момент времени сможет только один поток. В идеале хотелось бы добиться более впечатляющего результата.

Разработка отображаемой структуры данных для подробной детализации блокировок

Как и в случае с очередью, рассмотренной в подразделе 6.2.3, чтобы допустить применение подробно детализированной блокировки, нужно тщательно продумать все детали структуры данных, а не заключать ее в уже существующий контейнер, например в `std::map<>`. Реализовать ассоциативный контейнер наподобие нашей поисковой таблицы можно тремя широко используемыми способами:

- в виде бинарного дерева, такого как красно-черное дерево;
- в виде отсортированного массива;
- в виде хеш-таблицы.

Бинарное дерево не предоставляет слишком большого пространства для расширения возможностей конкурентности: каждые поиск или изменение должны начинаться с обращения к корневому узлу, который по этой причине следует заблокировать. Блокировку можно снять, как только обращающийся поток спустится вниз по дереву, однако это ненамного лучше единой блокировки, распространенной на всю структуру данных.

Отсортированный массив еще хуже, поскольку заранее невозможно сказать, в каком месте массива находятся искомые данные, поэтому нужна единая блокировка на весь массив.

Остается хеш-таблица. При фиксированном количестве сегментов вопрос принадлежности ключа сегменту — исключительно свойство ключа и его хеш-функции. Это означает, что можно применять безопасную отдельную блокировку каждого сегмента. При повторном использовании мьютекса, который поддерживает несколько считывающих потоков или один записывающий поток, возможность конкурентности увеличивается в N раз, где N — количество сегментов. Недостатком является необходимость задействования для ключа хорошей хеш-функции. Стандартная библиотека C++ предоставляет шаблон `std::hash<>`, которым можно воспользоваться для этой цели. Он уже приспособлен к основным типам, таким как `int`, и к распространенным библиотечным типам, таким как `std::string`, а пользователь может легко приспособить его и для других типов ключей. Если принять как пример стандартные неупорядоченные контейнеры и взять в качестве параметра шаблона, который будет использоваться для хеширования, тип функционального объекта, пользователь может выбрать, следует ли приспосабливать `std::hash<>` для его типа ключа или же предоставить отдельную хеш-функцию.

Взглянем на код. На что может быть похожа реализация потокобезопасной поисковой таблицы? Один из возможных вариантов показан в листинге 6.11.

Листинг 6.11. Потокобезопасная поисковая 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 std::shared_mutex mutex;            ←❶

        bucket_iterator find_entry_for(Key const& key) const  ←❷
        {
            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
        {
            std::shared_lock<std::shared_mutex> lock(mutex);   ←❸
            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<std::shared_mutex> lock(mutex);   ←❹
            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<std::shared_mutex> lock(mutex);   ←❺
            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;       ❻
    Hash hasher;
    bucket_type& get_bucket(Key const& key) const     ❼
    {
        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);  ❽
    }
    void add_or_update_mapping(Key const& key,Value const& value)
    {
        get_bucket(key).add_or_update_mapping(key,value);  ❾
    }
    void remove_mapping(Key const& key)
    {
        get_bucket(key).remove_mapping(key);  ❿
    }
};
```

В этой реализации для хранения сегментов используется вектор `std::vector<std::unique_ptr<bucket_type>>` ❻, позволяющий задавать в конструкторе количество сегментов. По умолчанию берется произвольное простое число 19 — с количествами сегментов, выраженными простыми числами, хеш-таблицы работают лучше. Каждый сегмент защищен экземпляром мьютекса `std::shared_mutex` ❶, допускающим выполнение множества конкурентных считываний или единственный вызов любой из функций, вносящих изменения в отношении каждого сегмента.

Поскольку количество сегментов зафиксировано, функция `get_bucket()` ❼ может вызываться без блокировок (❽, ❾ и ❿), а затем мьютекс сегмента можно заблокировать либо для совместного владения (только для чтения) ❸, либо для исключительного владения (для чтения — записи) (❹ и ❺) в зависимости от применимости к каждой функции.

Чтобы определить наличие записи в сегменте, всеми тремя функциями в отношении сегмента используется компонентная функция `find_entry_for()` ❷. В каждом

сегменте содержится всего лишь список пар «ключ — значение» `std::list<>`, поэтому добавление и удаление записей выполняется легко.

Одновременность доступа и надлежащую защиту данных с помощью блокировки мьютексов мы рассмотрели, а как насчет безопасности выдачи исключений? Функция `value_for` не вносит никаких изменений, значит, с ней все в порядке: если она выдаст исключение, это не повлияет на структуру данных. Функция `remove_mapping` вносит изменения в список за счет вызова функции `erase`, но она гарантированно не выдает исключение, стало быть, в ней тоже все безопасно. Остается только функция `add_or_update_mapping`, которая может выдать исключение в любой из двух ветвей `if`. Функция `push_back` в этом смысле безопасна и в случае выдачи исключений сохранит исходное состояние списка в неприкосновенности. Следовательно, и с этой ветвью все в порядке. Единственную проблему создает присваивание в случае замены существующего значения: если оператор присваивания выдаст исключение, остается полагаться на то, что он не изменит исходное состояние. Но это не повлияет на структуру данных в целом, так как имеет отношение только к типу, предоставленному пользователем, на которого свободно можно возложить решение данной проблемы.

В начале раздела уже упоминалось, что одной из предпочтительных особенностей такой поисковой таблицы было бы получение мгновенного образа ее текущего состояния, например, в виде объекта `std::map<>`. Для этого может потребоваться заблокировать весь контейнер, чтобы обеспечить извлечение целостной копии состояния, и без блокировки всех сегментов здесь не обойтись. Поскольку обычные операции над поисковой таблицей требуют в конкретный момент блокировки только одного сегмента, это станет единственной операцией, запрашивающей блокировки сразу всех сегментов. Поэтому их блокировка в одном и том же порядке (например, с приращением индекса сегмента) делает невозможной взаимную блокировку. Соответствующая реализация показана в листинге 6.12.

Листинг 6.12. Получение содержимого threadsafe_lookup_table в виде объекта std::map<>

```
std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
    std::vector<std::unique_lock<std::shared_mutex> > locks;
    for(unsigned i=0;i<buckets.size();++i)
    {
        locks.push_back(
            std::unique_lock<std::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 поисковой таблицы, показанная в листинге 6.11, расширяет возможности одновременного доступа к данным всей таблицы за счет блокировки каждого сегмента по отдельности и использования мьютекса `std::shared_mutex`, позволяющего выполнять конкурентное чтение каждого сегмента. А нельзя ли расширить возможности одновременного доступа к данным с помощью более подробной детализации блокировки? Ответ на этот вопрос будет дан в следующем разделе при разборе решения на основе применения потокобезопасного контейнера списка с поддержкой итератора.

6.3.2. Создание потокобезопасного списка с использованием блокировок

Список является одной из наиболее востребованных структур данных, поэтому стремление создать потокобезопасный список не вызывает вопросов. Все зависит от того, какие средства вам нужны, и следует выбрать средство с поддержкой итераторов, от добавления которого в отображение (map) я отказался, так как оно было слишком сложным. Основной проблемой поддержки итератора в стиле STL является необходимость сохранения в нем какой-либо ссылки на внутреннюю структуру данных контейнера. Если контейнер можно изменить из другого потока, эта ссылка должна оставаться действительной, что потребует удержания итератором блокировки на какую-то часть структуры. Учитывая, что время жизни итератора STL-стиля никак не контролируется контейнером, от применения такого итератора придется отказаться.

Альтернативой станет включение функций итерации, таких как `for_each`, в сам контейнер в качестве его составной части. Тем самым вся ответственность за итерацию и блокировку будет возложена на контейнер, но это противоречит рекомендациям по предотвращению взаимных блокировок, изложенным в главе 3. Чтобы функция `for_each` выполняла полезную работу, она должна вызывать код, предоставленный пользователем, и при этом удерживать внутреннюю блокировку. Но этим дело не ограничивается. Она также должна передавать ссылку на каждый элемент контейнера пользовательскому коду, чтобы тот мог обратиться к этому элементу. Этого можно избежать, передав пользовательскому коду копии всех элементов, но если элементы данных будут большими, такой вариант станет слишком затратным.

Итак, на данный момент ответственность за предотвращение взаимных блокировок из-за установки блокировок в операциях, предоставленных пользователем, а также за недопущение состояний гонки за данными из-за хранения ссылок для обращений к ним вне блокировок будет возложена на пользователя. При использовании списка таблицей поиска гарантируется полная безопасность, поскольку вам известно, что ничего предосудительного происходить не будет.

Остается только вопрос о составе операций, предлагаемых списком. Если вернуться к коду листингов 6.11 и 6.12, можно понять, какого рода операции потребуются:

- добавление элемента к списку;
- удаление из списка элемента, отвечающего конкретному условию;

- поиск в списке элемента, отвечающего конкретному условию;
- обновление элемента, отвечающего конкретному условию;
- копирование всех элементов списка в другой контейнер.

Чтобы списочный контейнер приобрел универсальность, было бы неплохо добавить еще несколько операций, например вставку элемента в конкретное место, но для нашей поисковой таблицы этого не нужно, поэтому пусть решение данной задачи станет упражнением для читателей.

Замысел, положенный в основу более подробной детализации блокировки связанного списка, заключается в наличии мьютекса для каждого узла. Если список растет, мьютексов становится слишком много! Преимущество данного подхода состоит в том, что операции над отдельными частями списка выполняются действительно одновременно: каждая операция удерживает блокировки только тех узлов, которые ее интересуют, и снимает блокировку каждого узла по мере перемещения к следующему узлу. Реализация такого списка показана в листинге 6.13.

Листинг 6.13. Потокобезопасный список с поддержкой итерации

```
template<typename T>
class threadsafe_list
{
    struct node       ←❶
    {
        std::mutex m;
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
        node():           ←┐
            next()          ❷
        {}
        node(T const& value):        ←❸
            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));    ←❹
        std::lock_guard<std::mutex> lk(head.m);        ←❺
        new_node->next=std::move(head.next);    ←┐
        head.next=std::move(new_node);            ❻
    }
    template<typename Function>
```

```
    void for_each(Function f)          ❼
    {
        node* current=&head;                                ❽
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())   ❾
        {
            std::unique_lock<std::mutex> next_lk(next->m);  ❿
            lk.unlock();          ⓫
            f(*next->data);   ⓬
            current=next;
            lk=std::move(next_lk);  ⓭
        }
    }
    template<typename Predicate>
    std::shared_ptr<T> find_first_if(Predicate p)  ⓮
    {
        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))  ⓯
            {
                return next->data;     ⓰
            }
            current=next;
            lk=std::move(next_lk);
        }
    return std::shared_ptr<T>();
    }
    template<typename Predicate>
    void remove_if(Predicate p)  ⓱
    {
        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))                              ⓲
            {
                std::unique_ptr<node> old_next=std::move(current->next);
                current->next=std::move(next->next);
                next_lk.unlock();                                ⓳
            }                       ⓴
            else
            {
                lk.unlock();      ㉑
                current=next;
                lk=std::move(next_lk);
            }
        }
    }
};
```

Шаблон `threadsafe_list<>` из листинга 6.13 является однонаправленным списком, где каждый элемент представлен структурой типа `node` (узел) ❶. В качестве головы списка `head`, изначально имеющей `next`-указатель со значением `NULL` ❷, используется сконструированная с установками по умолчанию структура `node`. Новые узлы добавляются с помощью функции `push_front()`: сначала создается новый узел ❹ и под сохраняемые в нем данные распределяется место в куче ❸, а указателю `next` присваивается значение `NULL`. Затем нужно заблокировать мьютекс для узла `head`, чтобы получить соответствующее значение указателя `next` ❺, и вставить узел в начало списка, установив для `head.next` значение указателя на новый узел ❻. Пока нас все устраивает: чтобы добавить к списку новую запись, требуется заблокировать всего один мьютекс, следовательно, нет риска взаимной блокировки. К тому же медленное распределение памяти происходит вне блокировки: получается, что блокировка защищает только обновление двух значений указателей, что не может дать сбой. Теперь перейдем к функциям, выполняющим итерацию.

Сначала рассмотрим `for_each()` ❼. Эта операция получает функцию некоторого типа, применяемую ко всем элементам списка. Как и в большинстве стандартных библиотечных алгоритмов, она получает эту функцию по значению и будет работать вместе либо с ней, либо с объектом типа, имеющим оператор вызова функции. В данном случае функция должна принимать в качестве единственного параметра значение типа `T`. Здесь выполняется эстафетная блокировка с перекрытием. Сначала блокируется мьютекс на узле `head` ❽. После этого можно будет безопасно получить указатель на следующий узел `next` (используя `get()`, поскольку владение указателем не принимается). Если значение этого указателя не `NULL` ❾, блокируется мьютекс на этом узле ❿, чтобы обработать данные. Получив блокировку на данном узле, можно снять блокировку с предыдущего узла ⓫ и вызвать указанную функцию ⓬. Когда функция завершит работу, можно будет обновить указатель `current` на обработанный узел и с помощью функции `move` передать владение блокировкой от `next_lk` в `lk` ⓭. Поскольку `for_each` передает каждый элемент данных непосредственно предоставленной функции, можно при необходимости воспользоваться этим для обновления элемента или его копирования в другой контейнер либо сделать все, что угодно. Если отклонений в поведении функции не будет, никакой опасности не возникнет, поскольку мьютекс для узла, содержащего элемент данных, удерживается в течение всего вызова.

Функция `find_first_if()` ⓮ похожа на `for_each()`, основное отличие в том, что предоставляемый предикат должен возвращать `true`, чтобы показать соответствие, или `false`, если оно не найдено ⓯. Как только соответствующий элемент будет определен, возвращаются содержащиеся в нем данные ⓰, а поиск прекращается. То же самое можно было бы сделать с помощью функции `for_each()`, но после найденного соответствия она продолжит ненужную обработку остальной части списка.

Функция `remove_if()` ⓱ немного отличается от предыдущей функции, поскольку она должна обновлять список. Воспользоваться для этого функцией `for_each()` нельзя. Если предикат возвращает `true` ⓲, узел из списка удаляется путем обновления значения `current->next` ⓳. После удаления блокировку мьютекса для узла `next` можно снять. Узел удаляется, когда объект `std::unique_ptr<node>`, в который

он был перемещен, выходит из области видимости ⓴. В данном случае значение `current` не обновляется, так как необходимо проверить новый узел `next`. Если предикат возвращает `false`, нужно перемещаться по списку, как и раньше ㉑.

Возможно ли при таком количестве мьютексов возникновение взаимных блокировок или состояний гонки? При правильном поведении предоставляемых предикатов и функций это полностью исключается. Итерация всегда проходит в одном направлении, всегда начинается с узла `head` и всегда блокирует следующий мьютекс, прежде чем разблокировать текущий, следовательно, иной порядок блокировки в других потоках полностью исключен. Единственным возможным кандидатом на провоцирование состояния гонки является удаление намеченного для этого узла в функции `remove_if()` ⓴, поскольку оно выполняется после разблокировки мьютекса (уничтожение заблокированного мьютекса приводит к неопределенному поведению). Но если немного подумать, можно прийти к выводу, что это абсолютно безопасно, так как на прежнем узле (`current`) мьютекс все еще удерживается, поэтому никакой новый поток не сможет предпринять попытку завладеть блокировкой на удаляемом узле.

А как насчет возможностей конкурентности? Вся эта более подробная детализация затевалась для расширения именно этих возможностей по сравнению с теми, которые появлялись при использовании одного мьютекса. Удалось ли достичь желаемого? Конечно, удалось: разные потоки могут одновременно работать с разными узлами списка, обрабатывая каждый элемент с помощью `for_each()`, выполняя поиск с применением `find_first_if()` или удаляя элементы с помощью `remove_if()`. Но поскольку мьютексы для каждого узла должны блокироваться по порядку, потоки не могут обгонять друг друга. Если один поток тратит много времени на обработку конкретного узла, другим потокам, подошедшим именно к этому узлу, придется ждать.

Резюме

В начале главы речь шла о том, что подразумевается под разработкой структуры данных для реализации конкурентности, и приводились соответствующие рекомендации. Затем были рассмотрены примеры практического применения этих рекомендаций при реализации наиболее востребованных структур данных с возможностью конкурентных обращений (стек, очередь, хеш-таблица и связанный список) за счет использования блокировок и предотвращения состояний гонки за доступ к данным. Теперь вы сможете оценить конструкцию собственной структуры данных и подумать, где в ней имеется потенциал для конкурентности и где может возникнуть состояние гонки.

В главе 7 будет показано, как при соблюдении прежнего набора рекомендаций отказаться от блокировок и реализовать необходимые ограничения, направленные на упорядочение доступа к данным за счет низкоуровневых атомарных операций.

7 Разработка конкурентных структур данных без блокировок

В этой главе

- Реализация структуры данных для конкурентного доступа без применения блокировок.
- Приемы управления памятью в структурах данных без блокировок.
- Простые рекомендации по созданию структур данных без блокировок.

В предыдущей главе рассматривались общие вопросы разработки структур данных, допускающих конкурентный доступ, и были даны рекомендации, позволяющие продумывать их с точки зрения обеспечения безопасности. Затем мы изучили несколько наиболее востребованных структур данных и примеры их реализации, в которых для защиты совместно используемых данных применялись мьютексы и блокировки. В двух первых примерах для защиты всей структуры задействовался всего один мьютекс, а в остальных для защиты меньших частей структуры данных и достижения более высокого уровня конкурентности при доступе к структуре данных использовалось несколько мьютексов.

Мьютексы предоставляют весьма эффективный механизм обеспечения безопасного доступа к структуре данных сразу нескольких потоков без состояний гонки или нарушений инвариантов. К тому же относительно несложно предсказать поведение использующего эти мьютексы кода: он либо блокирует защищающий данные мьютекс, либо нет. Но не все так радужно: в главе 3 было показано, что неправильное применение блокировок может привести к взаимным блокировкам. На примере

очередей на основе использования блокировок и поисковых таблиц проиллюстрировано, как более подробная детализация блокировки способна повлиять на возможности по-настоящему конкурентного доступа. Если бы можно было создать структуры данных, допускающие конкурентный доступ без блокировок, реально было бы избавиться от упомянутых проблем. В результате получилась бы структура данных, *свободная от блокировок*.

В этой главе рассмотрим способы применения свойств атомарных операций, позволяющих упорядочить доступ к памяти (знакомство с ними состоялось в главе 5), к созданию структур данных, свободных от блокировок. Все, что было усвоено при изучении главы 5, станет жизненно необходимым для освоения материалов данной главы. При конструировании таких структур данных следует проявлять предельную осторожность, поскольку их осмысление дается нелегко, а условия, при которых можно выявить недостатки конструкции, возникают крайне редко. Сначала нужно будет разобраться, чем именно для структур данных определяется свобода от блокировок, затем перейти к причинам, побуждающим к использованию таких структур, и только потом разобрать пару примеров и дать несколько общих рекомендаций.

7.1. Определения и выводы

Алгоритмы и структуры данных, в которых для синхронизации доступа к данным задействуются мьютексы, условные переменные и фьючерсы, называются *блокирующими* структурами данных и алгоритмами. Приложения вызывают библиотечные функции, приостанавливающие выполнение потока, пока другой поток совершает какое-либо действие. Такие библиотечные вызовы называются *блокирующими*, поскольку в ходе выполнения поток не может преодолеть данное место программы, пока блокировка не будет снята. Обычно операционная система полностью останавливает заблокированный поток и распределяет его кванты времени другим потокам, пока он не будет разблокирован соответствующим действием со стороны другого потока — того, который снимает блокировку с мьютекса, или уведомляет условную переменную, или устанавливает готовность фьючерса.

Структуры данных и алгоритмы, не использующие блокирующие библиотечные функции, называются *неблокирующими*. Но не все подобные структуры данных свободны от блокировок, поэтому рассмотрим различные типы структур данных, не подвергаемых блокировкам.

7.1.1. Типы структур данных, не подвергаемых блокировкам

В главе 5 была показана реализация основного мьютекса, применяемого в качестве спин-блокировки шаблон класса `std::atomic_flag`. Код этой реализации воспроизведен в листинге 7.1.

Листинг 7.1. Реализация мьютекса со спин-блокировкой, использующего 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);
    }
};
```

В этом коде не вызываются никакие блокирующие функции: функция `lock()` находится в цикле, пока вызов `test_and_set()` не вернет `false`. Именно поэтому и появилось название *«спин-блокировка»* — код «вращается» (spin) в цикле. Здесь нет блокирующих вызовов, следовательно, любой код, применяющий данный мьютекс для защиты совместно используемых данных, считается *неблокирующим*. Но он не свободен от блокировок. Это все тот же мьютекс, который по-прежнему может быть одновременно заблокирован только одним потоком. Поэтому в большинстве случаев недостаточно знать, что какой-то код является неблокирующим. Нужно разобраться, какие из более конкретных понятий, определения которых приведены далее (если таковые имеются), применимы для того или иного случая. К таким понятиям относятся:

- *свободный от помех* (Obstruction-Free) — если все другие потоки находятся в режиме паузы, то любой рассматриваемый поток завершит свои операции за ограниченное число шагов;
- *свободный от блокировок* (Lock-Free) — если над структурой данных работают сразу несколько потоков, то после ограниченного числа шагов один из них завершит свою операцию;
- *свободный от ожиданий* (Wait-Free) — каждый поток, работающий над структурой данных, завершит свою операцию за ограниченное число шагов, даже если другие потоки также работают над этой же структурой данных.

Алгоритмы, свободные от помех, практически бесполезны, так как паузы в других потоках случаются нечасто, поэтому от них больше пользы в качестве представлений о неудачной реализации структур данных без блокировок. Подробнее поговорим о том, что связано с этими понятиями, начиная с кода, свободного от блокировок, и разберемся в структурах данных, относящихся к этой категории.

7.1.2. Структуры данных, свободные от блокировок

Чтобы квалифицировать структуру данных как свободную от блокировок, нужно, чтобы одновременный доступ к ней могли получать сразу несколько потоков. При этом не обязательно, чтобы эти потоки выполняли одни и те же операции: очередь, свободная от блокировок должна позволять одному потоку помещать, а другому — извлекать данные, но пресекать попытки помещения новых записей двумя потоками одновременно. При этом, если один из потоков, обращающийся к структуре данных, приостановлен диспетчером на половине своей операции, нужно, чтобы другие потоки все равно могли завершить свои операции, не ожидая, когда приостановленный поток возобновит выполнение.

В алгоритмах, использующих над структурой данных операции сравнения — обмена, часто имеются циклы. Причина применения операции сравнения — обмена заключается в том, что в ходе операции другой поток мог изменить данные. В таком случае коду нужно будет выполнить откат части своей операции, прежде чем снова пытаться выполнить сравнение — обмен. Такой код может по-прежнему обходиться без блокировок при условии, что сравнение — обмен завершится успешно, если другие потоки будут приостановлены. Если этого не случится, то получится спин-блокировка, не приводящая к блокировке данных, но и не свободная от блокировок.

Применение алгоритмов, свободных от блокировок, но имеющих подобные циклы, может привести к тому, что один поток *зависнет*. Если очередной поток выполняет операции не вовремя, свою работу может начать другой поток, но первому придется раз за разом повторять свою операцию. Структура данных, в которой удается обойти эту проблему, называется свободной от ожиданий, а также свободной от блокировок.

7.1.3. Структуры данных, свободные от ожиданий

Свободными от ожиданий являются структуры данных без блокировок, с дополнительным свойством: каждый обращающийся к структуре данных поток может завершить свою операцию за ограниченное число шагов независимо от поведения других потоков. Алгоритмы, в которых может предприниматься неограниченное число попыток из-за помех со стороны других потоков, свободными от ожиданий не считаются. Эта особенность характерна для большинства примеров, приводимых в данной главе, — в них имеется цикл `while` в операциях `compare_exchange_weak` или `compare_exchange_strong`, не имеющий верхнего предела количества повторений. Диспетчеризация потоков операционной системой может означать, что один поток может проходить цикл огромное количество раз, а другие потоки — всего несколько раз. Поэтому такие операции не считаются свободными от ожиданий.

Создать структуры данных, свободные от ожиданий, нелегко. Чтобы любой поток мог завершать свои операции за ограниченное число шагов, нужно обеспечить возможность выполнения каждой операции за один проход и гарантировать, что шаги, выполняемые одним потоком, не станут причиной сбоя операции, выполняемой другим потоком. Эти условия могут существенно усложнить общие алгоритмы выполнения различных операций.

Учитывая сложности, сопряженные с созданием работоспособных структур данных, свободных от блокировок и от ожиданий, вы должны иметь весьма веские причины для их создания и быть уверенными в том, что получаемые преимущества перевесят все затраты. Поэтому изучим все положения, влияющие на этот баланс.

7.1.4. Все за и против создания структур данных, свободных от блокировок

Когда нужно принимать решение, основной причиной использования структур данных, свободных от блокировок, становится достижение максимальной конкурентности. Применение контейнеров, основанных на блокировках, всегда сопряжено с возникновением ситуаций, при которых один поток должен быть заблокирован и, прежде чем продолжить выполнение, ожидать, пока другой поток не завершит свою операцию. Цель блокировки мьютекса — предотвращение конкурентности путем взаимного исключения. При использовании структуры данных, свободной от блокировок, подвижки у какого-либо потока бывают с каждым шагом. А когда задействуется структура данных, свободная от ожиданий, подвижки могут быть у каждого потока независимо от того, чем будут заняты другие потоки, то есть необходимость в ожидании исчезнет. Неплохо обладать такой возможностью, но достичь ее бывает крайне трудно. Проще всего остановиться на создании чего-либо подобного спин-блокировке.

Второй причиной использования структур данных, свободных от блокировок, является надежность. Если поток завершится, не сняв блокировки, структура данных будет испорчена навсегда. Но если такое случится с потоком на середине выполнения операции над структурой данных без блокировок, потеряются только данные этого потока и все остальные потоки смогут продолжить работу в обычном режиме.

Но здесь есть и обратная сторона: если нельзя исключить потоки из доступа к структуре данных, нужно действовать осмотрительно, гарантируя поддержку инвариантов или выбирая доступные альтернативные инварианты. Кроме того, нужно обратить внимание на ограничения порядка доступа к данным, накладываемые на операции. Во избежание неопределенного поведения, связанного с состоянием гонки за данными, для изменений необходимо использовать атомарные операции. Но и этого недостаточно. Нужно убедиться, что изменения становятся видны другим потокам в правильном порядке. Из этого следует, что создание потокобезопасных структур данных без применения блокировок значительно сложнее создания структур с блокировками.

Ввиду отсутствия блокировок обычные взаимоблокировки в структурах данных Lock-Free невозможны, хотя вместо них появляется возможность возникновения динамических взаимоблокировок. Такие *взаимные блокировки* возникают, когда сразу два потока пытаются изменить структуру данных, но для каждого из них изменения, внесенные другим потоком, требуют перезапуска операции, поэтому оба зацикливаются на повторении попытки.

Представьте, что два человека пытаются пройти в узкий проход. Делая это одновременно, они застревают, и им нужно выйти и повторить попытку. Если кто-то из них не будет первым (по договоренности, из-за расторопности или чистой случайности), цикл повторится. Как и в этом простом примере, динамические взаимоблокировки, как правило, недолговечны, поскольку зависят от четкости диспетчеризации потоков. Поэтому чаще всего все сводится к снижению производительности, а не к возникновению долгосрочных проблем. Но подобных блокировок все равно следует избегать. По сути, код, свободный от ожиданий, не может страдать от динамических взаимоблокировок, так как всегда есть верхний предел количества шагов, необходимых для выполнения операции. Но есть и обратная сторона: алгоритм, скорее всего, окажется сложнее альтернативного варианта и может потребовать большего количества шагов, даже когда другие потоки не обращаются к структуре данных.

В этом проявляется еще одна негативная сторона кода, свободного от блокировок и ожиданий: хотя он позволяет увеличить вероятность конкурентности для операций над структурами данных и сократить время, затрачиваемое отдельно взятым потоком на ожидание, общая производительность может *снизиться*. Атомарные операции, используемые для кода, свободного от блокировок, могут выполняться гораздо медленнее неатомарных операций, и, скорее всего, в структуре данных без блокировок таких операций будет гораздо больше, чем в структуре данных, рассчитанной на применение в коде блокировки мьютексов. Но это еще не все. Аппаратура должна синхронизировать доступ к данным между потоками, обращающимися к одним и тем же атомарным переменным. В главе 8 будет показано, что эффект переключения кэша, связанный с обращением нескольких потоков к одним и тем же атомарным переменным, может существенно снизить производительность. Как и во всех других случаях, прежде чем отдавать предпочтение какому-либо из направлений, важно проконтролировать соответствующие аспекты производительности (наибольшее время ожидания, среднее время ожидания, общее время выполнения или что-то еще) для обоих типов структур данных, как основанных на применении блокировок, так и свободных от них.

А теперь рассмотрим несколько примеров.

7.2. Примеры структур данных, свободных от блокировок

Чтобы продемонстрировать приемы конструирования структур данных, свободных от блокировок, будет рассмотрен ряд простых примеров реализации таких структур. Каждый пример будет содержать описание полезной структуры данных, а также использоваться для рассмотрения конкретных аспектов конструкции структур данных, свободных от блокировок.

Как уже упоминалось, структуры данных, свободные от блокировок, создаются на основе атомарных операций и связанных с ними гарантий упорядоченного обращения к памяти, обеспечивающих правильный порядок видимости данных другими потоками. Сначала во всех атомарных операциях будет применен порядок

обращения к памяти по умолчанию (`memory_order_seq_cst`), так как в нем проще разобраться (напомню, что для всех операций с семантикой `memory_order_seq_cst` формируется общий порядок доступа к памяти). Но в последующих примерах некоторые ограничения порядка доступа к памяти будут сведены к использованию семантики `memory_order_acquire`, `memory_order_release` или даже `memory_order_relaxed`. Хотя мьютексы непосредственно не задействованы ни в одном из примеров, следует помнить, что гарантию отсутствия блокировок в реализации дает только установка флага `std::atomic_flag`. На некоторых платформах в, казалось бы, свободном от блокировок коде возможны внутренние блокировки, присущие реализации стандартной библиотеки C++ (подробности приведены в главе 5). Для таких платформ лучше подойдет простая структура данных, основанная на применении, но дело не только в этом: прежде чем выбрать вариант реализации, нужно выработать четкие требования и исследовать результаты профилирования различных вариантов, соответствующих им.

Итак, вернемся к самой простой структуре данных — к стеку.

7.2.1. Создание потокобезопасного стека без блокировок

Разобраться в основном замысле предназначения стека нетрудно: узлы извлекаются в порядке, обратном тому, в котором они добавлялись, — последним добавлен, первым извлечен (last in, first out, LIFO). Поэтому важно обеспечить возможность немедленного извлечения другим потоком только что добавленного в стек значения, а также возвращение конкретного значения только в одном потоке. Простейший стек представляет собой связанный список: указатель `head` служит идентификатором первого узла (который будет извлечен при следующем обращении к стеку), а затем каждый узел указывает поочередно на следующий узел.

По этой схеме добавить узел довольно просто.

1. Создать новый узел.
2. Установить для его указателя `next` текущее значение `head`.
3. Установить для `head` указатель на новый узел.

В однопоточной среде такая схема вполне работоспособна, но, если изменения в стек вносят и другие потоки, этого будет недостаточно. Важно то, что при добавлении узлов двумя потоками между шагами 2 и 3 возникает состояние гонки: второй поток может изменить значение `head` между моментом считывания его вашим потоком при выполнении шага 2 и вашим обновлением при выполнении шага 3. Получится, что изменения, внесенные другим потоком, будут отменены или события пойдут по еще более неприятному сценарию. Прежде чем взяться за решение данной проблемы, важно отметить, что после обновления `head`, превращающего его в указатель на новый узел, другой поток сможет прочитать этот узел. Поэтому важно тщательно подготовить новый узел *до того*, как `head` станет указывать на него, поскольку после этого изменить узел будет невозможно.

И как справиться с этим досадным состоянием гонки? Нужно на шаге 3 воспользоваться атомарной операцией сравнения — обмена, чтобы гарантировать,

что значение head не было изменено с момента его чтения при выполнении шага 2. Если изменение все же произошло, можно вернуться в начало цикла и предпринять новую попытку. Способ реализации потокобезопасной функции push() без блокировок показан в листинге 7.2.

Листинг 7.2. Реализация push() без блокировок

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        T data;
        node* next;
        node(T const& data_): ←❶
        data(data_)
        {}
    };
    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data); ←❷
        new_node->next=head.load(); ←❸
        while(!head.compare_exchange_weak(new_node->next,new_node)); ←❹
    }
};
```

Этот код в точности соответствует показанной ранее схеме из трех шагов: создание нового узла ❷, установка для next-указателя узла текущего значения указателя head ❸ и установка для head значения, указывающего на новый узел ❹. Поскольку сама структура node заполняется из конструктора node ❶, узел готов к работе сразу же после его создания, то есть менее серьезная проблема уже решена. Затем, чтобы удостовериться, что в указателе head по-прежнему содержится то же самое значение, которое было сохранено в new_node->next ❸, используется функция compare_exchange_weak() и, если так оно и есть, это значение устанавливается равным new_node. В этом фрагменте кода также задействуется весьма привлекательное свойство операций сравнения — обмена: если возвращается значение false, показывающее неудачное сравнение (например, из-за того, что значение head было изменено другим потоком), то значение, предоставленное в качестве первого параметра (new_node->next), обновляется текущим значением head. Поэтому нет необходимости перезагружать head при каждом прохождении цикла — компилятор сделает это сам. Кроме того, поскольку при неудачном сравнении сразу выполняется переход в начало цикла, появляется возможность задействовать функцию compare_exchange_weak, что может привести к созданию в некоторых архитектурах более оптимального кода, чем при использовании функции compare_exchange_strong (см. главу 5).

Итак, операции pop() пока нет, но реализацию операции push() уже можно быстро сверить с рекомендациями. Единственным местом выдачи исключения

является код конструирования нового узла node ❶, но он все подчистит за собой самостоятельно, и на этот момент список еще не будет изменен, стало быть, безопасность будет обеспечена. Поскольку данные создаются для того, чтобы сохранять их в качестве части узла node, а для обновления указателя head используется функция compare_exchange_weak(), проблемные условия гонки здесь не создаются. Если операция сравнения — обмена завершается успешно, узел попадает в список и его можно извлечь. Блокировок нет, значит, нет и вероятности возникновения взаимных блокировок и ваша функция push() получает высокий балл.

Теперь, когда уже есть средства добавления данных в стек, нужен способ их извлечения из него. На первый взгляд, в этом нет ничего сложного.

1. Считать текущее значение head.
2. Считать head->next.
3. Установить для head значение head->next.
4. Вернуть значение поля data из извлекаемого узла node.
5. Удалить извлеченный узел.

Однако при наличии нескольких потоков дело усложняется. Если элемент удаляется из стека двумя потоками, оба они на шаге 1 могут прочитать одно и то же значение head. Если затем один поток пройдет весь путь до шага 5, пока другой доберется до шага 2, второй поток будет разыменовывать висячий указатель. Это одна из самых серьезных проблем при создании кода, свободного от блокировок, поэтому пока оставим в покое шаг 5 и утечку узлов.

Но этим проблемы не ограничиваются. Вот еще одна: если два потока прочитают одно и то же значение head, оба они вернут один и тот же узел. Тем самым будет нарушен сам замысел структуры данных в виде стека, поэтому такого развития событий нужно избежать. Проблему можно решить так же, как устраняется состояние гонки в функции push(): для обновления head нужно воспользоваться операцией сравнения — обмена. Если эта операция даст сбой, значит, либо был вставлен новый узел, либо извлекаемый узел уже извлечен другим потоком. В любом случае следует вернуться к шагу 1 (хотя вызов операции сравнения — обмена сам перечитывает head).

Если операция сравнения — обмена завершится успешно, значит, ваш поток был единственным, извлекшим заданный узел из стека, поэтому можно спокойно выполнить шаг 4. Первая попытка создать код функции pop() выглядит так:

```
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;
    }
};
```

Этот код лаконичен и привлекателен, но все же, кроме утечки узлов, у него есть еще две проблемы. Начнем с того, что этот код не работает с пустым списком: если указатель `head` нулевой, то при попытке чтения указателя `next` поведение будет неопределенным. Это легко исправить проверкой на `nullptr` в цикле `while` и либо выдать исключение, если стек пуст, либо вернуть булево значение, показывающее успех или неудачу.

Вторая проблема связана с безопасностью выдачи исключений. Когда в главе 3 впервые был представлен потокобезопасный стек, было показано, что при возвращении объекта по значению вопрос безопасности выдачи исключений оставался нерешенным: если исключение выдавалось при копировании возвращаемого значения, это значение терялось. В таком случае вполне приемлемым было решение передавать результат по ссылке: это позволяло гарантировать, что при выдаче исключения стек останется неприкосновенным. К сожалению, здесь мы этого себе позволить не можем, можно лишь безопасно скопировать данные, убедившись, что только данный поток возвращает узел, а это означает, что узел уже удален из стека. Следовательно, передача целевого возвращаемого значения по ссылке больше не считается приоритетной и можно просто вернуть значение. Если нужно обеспечить безопасность возвращения значения, следует воспользоваться другим вариантом, упомянутым в главе 3, — возвращением интеллектуального указателя на значение данных.

Если возвращать интеллектуальный указатель, то, чтобы показать отсутствие возвращаемого значения, можно вернуть `nullptr`, но для этого потребуется распределить в куче память для данных. Если проделать распределение в куче в рамках выполнения функции `pop()`, лучше не станет, поскольку при этом может быть выдано исключение. Взамен можно распределить память при помещении данных в стек с помощью функции `push()` — все равно придется распределять память под узел `node`. Возвращение `std::shared_ptr<>` не приведет к выдаче исключения, следовательно, теперь функция `pop()` в этом смысле безопасна. Если собрать все это вместе, получится код, показанный в листинге 7.3.

Листинг 7.3. Свободный от блокировок стек, допускающий утечку узлов

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;
        node(T const& data_):
            data(std::make_shared<T>(data_))
        {}
    };
    std::atomic<node*> head;
public:
    void push(T const& data)
    {
```

❶ Теперь данные соотносятся с указателем

❷ Создание std::shared_ptr для только что распределенного T

```
        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()
    {
        node* old_head=head.load();
        while(old_head &&
            !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
    }
};
```

❸ Перед разыменованием old_head убеждаемся, что это ненулевой указатель

❹

Теперь данные соотносятся с указателем ❶, поэтому в конструкторе узла нужно распределить для них память в куче ❷. Следует также перед разыменованием указателя `old_head` проверить его в цикле функции `compare_exchange_weak()` на ненулевое содержимое ❸. И наконец, либо возвращаются данные, связанные с узлом, если он есть, либо нулевой указатель, если узла нет ❹. Код свободен от блокировок, но не свободен от ожиданий, поскольку теоретически циклы `while` как в `push()`, так и в `pop()` при продолжающихся неудачных завершениях функции `compare_exchange_weak()` могут выполняться бесконечно.

Если есть прибирающий за вами сборщик мусора (как в управляемых языках, таких как C# или Java), на этом можно было бы и закончить: старый узел был бы убран и утилизирован, если бы не было обращений к нему потоков. Но компиляторов C++, поставляемых со сборщиками мусора, мало, поэтому убирать за собой придется самостоятельно.

7.2.2. Устранение утечек: управление памятью в структурах данных, свободных от блокировок

При первом подходе к функции `pop()` мы выбрали вариант с утечкой узлов, чтобы избежать состояния гонки, возникающего, когда один поток удаляет узел, а другой продолжает удерживать указатель на него готовым к разыменованию. Но утечка памяти в любой серьезной программе на C++ недопустима, поэтому с ней что-то нужно делать. Рассмотрим, как решить эту проблему.

Основная трудность заключается в том, что нужно освободить занятую узлом память, но мы не можем сделать это, пока нет уверенности, что никакой другой поток не хранит указателей на нее. Если бы только один поток мог вызывать `pop()` в отношении конкретного экземпляра стека, то и делать бы ничего не пришлось. Узлы создаются при вызовах функции `push()`, которая не обращается к содержимому уже существующих узлов, поэтому к рассматриваемому узлу могут обращаться только поток, добавляющий узел в стек, а также любые потоки, вызвавшие функцию `pop()`. Функция `push()` не обращается к узлу после его добавления в стек, так что, кроме потока, вызвавшего `pop()`, этот узел больше никого не интересует, и его можно безопасно удалить.

В то же время, если приходится справляться сразу с несколькими потоками, вызывающими функцию pop() в отношении одного и того же экземпляра стека, нужен способ отслеживания момента безопасного удаления узла. Следовательно, для узлов node нужно создать специализированный сборщик мусора. Сейчас это может напугать, и хотя задача действительно не из легких, все не так уж плохо: проверять нужно только те узлы node, к которым идет обращение из функции pop(). Узлы, с которыми имеет дело функция push(), нас не интересуют, поскольку, пока они не попадут в стек, доступ к ним имеется только из одного потока, а обращение к одному и тому же узлу со стороны сразу нескольких потоков возможно при вызове функции pop().

Если потоков, вызывающих pop(), нет, то можно абсолютно спокойно удалить все узлы, которые именно этого и дожидаются. Поэтому если после извлечения данных добавлять узлы к списку удаляемых, то их все можно удалить, как только не будет потоков, вызывающих pop(). А как узнать об отсутствии потоков, вызывающих pop()? Да очень просто — их нужно подсчитать. Если добавлять к счетчику единицу при входе и отнимать единицу при выходе, то при его нулевом значении можно будет вполне безопасно удалить узлы из списка удаляемых. Это должен быть атомарный счетчик, тогда доступ к нему сразу из нескольких потоков будет безопасен. В листинге 7.4 показана измененная функция pop(), а в листинге 7.5 приведены функции, помогающие реализовать ее.

Листинг 7.4. Высвобождение памяти, занимаемой узлами в момент отсутствия вызова потоками функции pop()

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<unsigned> threads_in_pop;
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop()
    {
        ++threads_in_pop;
        node* old_head=head.load();
        while(old_head &&
            !head.compare_exchange_weak(old_head,old_head->next));
        std::shared_ptr<T> res;
        if(old_head)
        {
            res.swap(old_head->data);
        }
        try_reclaim(old_head);
        return res;
    }
};
```

❶ Атомарная переменная

❷ Увеличение значения счетчика до выполнения действий

❸ Извлечение данных из узла вместо копирования указателя

❹ Высвобождение памяти, распределенной под удаленные узлы, если это возможно

Атомарная переменная threads_in_pop ❶ используется для подсчета потоков, пытающихся в данный момент извлечь элемент из стека. Ее значение увеличивается на единицу при запуске функции pop() ❷ и уменьшается на единицу внутри функции try_reclaim(), вызываемой после удаления узла ❹. Поскольку удаление самого узла мы, возможно, собираемся отложить, для удаления данных из него можно восполь-

зоваться функцией `swap()` ❸, а не копировать указатель, тогда данные, как только станут не нужны, будут удаляться автоматически, вместо того чтобы их сохранять, поскольку в еще не удаленном узле на них будет иметься ссылка. Код, помещаемый в функцию `try_reclaim()`, показан в листинге 7.5.

Листинг 7.5. Механизм утилизации памяти на основе подсчета 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)                                          ❶
        {
            node* nodes_to_delete=to_be_deleted.exchange(nullptr);
            if(!--threads_in_pop)
            {
                delete_nodes(nodes_to_delete);                         ❹
            }
            else if(nodes_to_delete)                                   ❺
            {
                chain_pending_nodes(nodes_to_delete);                  ❻
            }
            delete old_head;                                           ❼
        }
        else
        {
            chain_pending_node(old_head);                              ❽
            --threads_in_pop;
        }
    }
    void chain_pending_nodes(node* nodes)
    {
        node* last=nodes;
        while(node* const next=last->next)
        {
            last=next;
        }
        chain_pending_nodes(nodes,last);
    }
    void chain_pending_nodes(node* first,node* last)
    {
        last->next=to_be_deleted;                                      ❿
```

❷ Утверждение списка подлежащих удалению узлов

❸ Это единственный поток, выполняющий код функции pop()?

❾ Проход по цепочке указателей next к концу списка

```
        while(!to_be_deleted.compare_exchange_weak(
            last->next,first));                          ⓫ Цикл, гарантирующий корректность last->next
    }
    void chain_pending_node(node* n)
    {
        chain_pending_nodes(n,n);                        ⓬
    }
};
```

Если при попытке утилизировать узел счетчик потоков, выполняющих извлечение, `threads_in_pop` равен единице ❶, значит, данный поток в этот момент единственный, выполняющий функцию `pop()`. Следовательно, будет безопасным удаление только что изъятого из списка узла ❼, равно как и узлов, удаление которых было отложено. Если значение счетчика не равно единице, удалять какие-либо узлы небезопасно, так что придется добавить узел к списку ожидающих удаления ❽.

Представим, что значение `threads_in_pop` равно единице. Теперь нужно попробовать утилизировать узлы, удаление которых было отложено. Если этого не сделать, они так и останутся отложенными, пока стек не будет удален. Для этого сначала с помощью атомарной операции `exchange` утверждается список ❷, после чего значение счетчика `threads_in_pop` уменьшается на единицу ❸. Если после уменьшения счетчик обнуляется, значит, обращаться к списку узлов, удаление которых было отложено, не может никакой другой поток. Могут существовать новые узлы, чье удаление откладывается, но в данный момент это не помеха, так как узлы из этого списка можно спокойно утилизировать. После этого для прохода по списку и удаления узлов можно вызвать функцию `delete_nodes` ❹.

Если значение счетчика после вычитания единицы не равно нулю, утилизировать узлы небезопасно, поэтому, если они есть ❺, их опять нужно выстроить в список узлов, ожидающих удаления ❻. Этот может произойти, если к структуре данных одновременно обращаются несколько потоков. Функция `pop()` могла быть вызвана другими потоками между первым тестированием значения счетчика `threads_in_pop` ❶ и утверждением списка ❷ с возможным добавлением новых узлов к списку удаляемых, по-прежнему доступному для одного или нескольких других потоков. На рис. 7.1 поток C добавляет узел Y к списку удаляемых, притом что поток B по-прежнему ссылается на него по указателю `old_head` и будет обращаться к его указателю `next` и считывать значение этого указателя. Поэтому поток A не сможет удалить узлы, не спровоцировав неопределенное поведение потока B.

Чтобы узлы, ожидающие удаления, встроились в цепочку списка узлов, ожидающих удаления, их нужно связать вместе, для чего следует повторно воспользоваться их указателем `next`. Если существующая цепочка повторно включена в список, то выполняется проход по ней, чтобы найти ее конец ❾, указатель `next` последнего узла заменяется текущим указателем `to_be_deleted` ❿, а в качестве нового указателя `to_be_deleted` сохраняется указатель на первый узел цепочки ⓫. Чтобы не допустить утечки узлов, добавленных другим потоком, придется задействовать в цикле вызов функции `compare_exchange_weak`. Это позволяет обновить указатель `next` с конца цепочки, если он был изменен. Добавление в список единственного узла можно отнести к особому случаю, когда первый узел добавляемой цепочки совпадает с ее последним узлом ⓬.

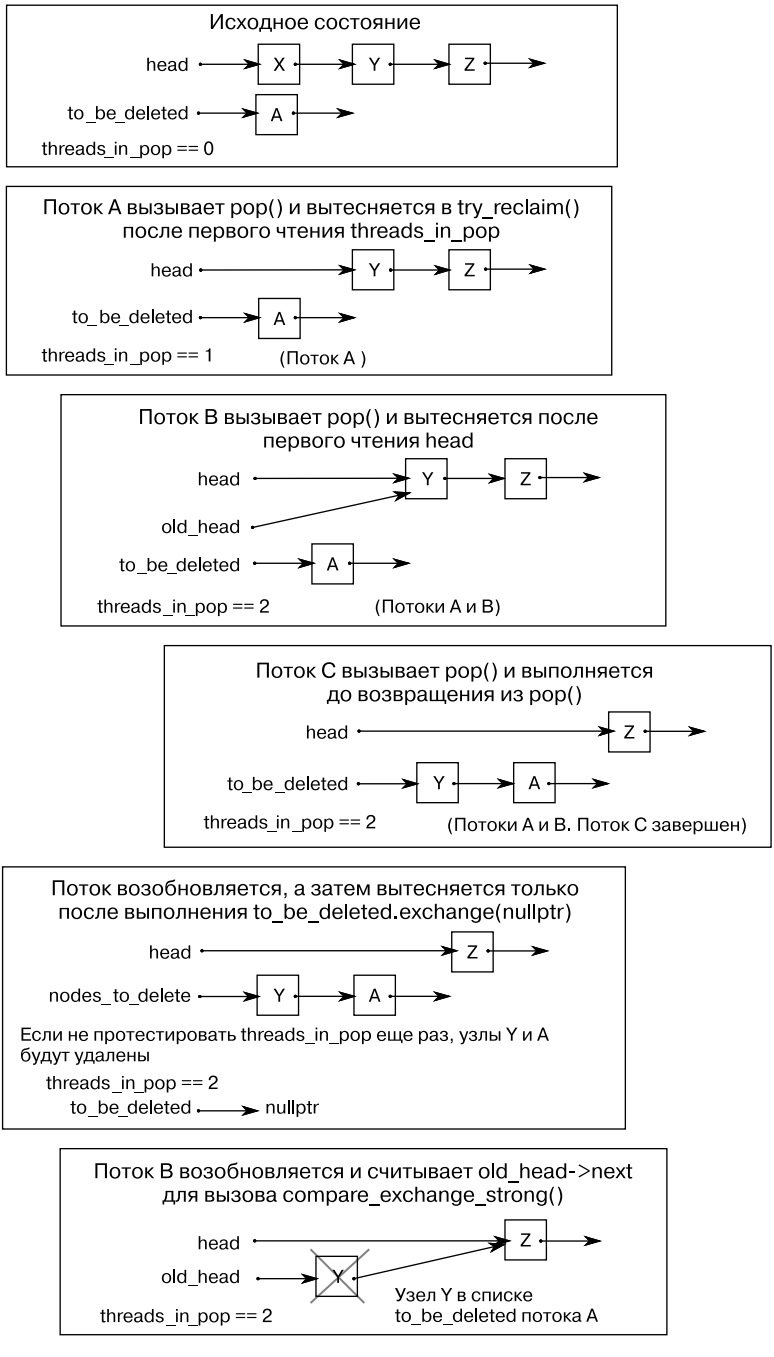


Рис. 7.1. Функция pop() одновременно вызывается тремя потоками, показывая причину, по которой значение счетчика threads_in_pop нужно проверять после утверждения удаляемых узлов в try_reclaim()

В условиях низкой загруженности, когда возникают моменты затишья и ни один из потоков не вызывает функцию `pop()`, этот код работает вполне приемлемо. Но потенциально благоприятная ситуация весьма кратковременна, что заставляет до утилизации проверять уменьшение значения счетчика `threads_in_pop` до нуля ❸, по той же причине такая проверка проводится до удаления только что изъятого узла ❼. Удаление узла может занять довольно много времени, и хотелось бы, чтобы период, в течение которого другие узлы могут изменить список, был как можно короче. Чем больше времени между моментом определения потоком равенства `threads_in_pop` единице и попыткой удаления узлов, тем выше вероятность вызова функции `pop()` другим потоком, при котором `threads_in_pop` уже не будет равен единице, что не позволит удалить узлы.

В условиях высокой загруженности такого затишья может не случиться из-за того, что одни потоки приступили к выполнению функции `pop()`, прежде чем ее выполнение было завершено другими потоками. При таком сценарии развития событий список `to_be_deleted` будет расти безгранично и снова возникнет утечка памяти. Если периодов затишья не предвидится, нужно найти альтернативный механизм утилизации узлов. Ключом послужит определение момента отсутствия обращения потоков к конкретному узлу, что позволит его утилизировать. Пока наиболее простым механизмом, достойным рассмотрения, является использование указателей опасности (hazard pointers).

7.2.3. Определение узлов, не подлежащих утилизации, с помощью указателей опасности

Понятие «*указатели опасности*» относится к технологии, предложенной Магидом Майклом (Maged Michael)[1]. Такое название появилось из-за того, что удалять узел, на который все еще могут ссылаться другие потоки, опасно. Если действительно есть потоки, ссылающиеся на данный узел и пытающиеся обратиться к узлу по этой ссылке, то поведение станет неопределенным. Основной замысел состоит в том, что, если поток собирается обратиться к объекту, удаление которого может потребоваться другому потоку, сначала он устанавливает для ссылки на объект указатель опасности, информирующий другой поток, что удалять объект действительно опасно. После того как необходимость в объекте исчезнет, указатель опасности будет снят. Вам доводилось смотреть соревнование по гребле между командами Оксфорда и Кембриджа? Там на старте гонки используется аналогичный механизм: рулевые обеих лодок могут поднять руку, показывая неготовность. Когда рука у любого из рулевых поднята, судья не может дать старт гонке. Если у обоих рулевых руки опущены, можно давать старт гонке. Но рулевой может снова поднять руку, если гонка еще не стартовала, но ситуация, на его взгляд, изменилась.

---

[1] *Michael M. M.* Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes // PODC'02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, 2002.

Когда потоку нужно удалить объект, он сначала должен проверить указатели опасности, принадлежащие другим потокам системы. Если на объект не ссылается ни один из указателей опасности, его можно спокойно удалить. В противном случае от этого нужно воздержаться. Периодически список объектов с отложенным удалением проверяется на предмет возможности удаления любого из объектов.

Описание на таком высоком уровне выглядит довольно просто, но как все это можно реализовать на C++?

Прежде всего нужно определить место для хранения указателя на объект, к которому идет обращение, то есть решить, где именно будет находиться сам указатель опасности. Это место должны видеть все потоки, и для каждого потока, способного обращаться к структуре данных, нужен один из таких указателей. Правильно и эффективно разместить эти указатели непросто, поэтому отложим решение на потом и представим, что есть функция получения указателя опасности для текущего потока `get_hazard_pointer_for_current_thread()`, возвращающая ссылку на указатель опасности. Указатель опасности нужно установить перед чтением указателя, подлежащего разыменованию, — в данном случае указателя `head` из 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 следует сделать в цикле `while`, чтобы убедиться, что узел `node` не был удален в промежутке между чтением старого указателя `head` ❶ и установкой указателя опасности ❷. В это время об обращении конкретно к этому узлу не знает ни один прочий поток. К счастью, если хочется удалить старый узел `head`, сам указатель `head` должен быть изменен. И это можно проверить, находясь в цикле, пока не удастся узнать, что указатель `head` по-прежнему имеет то же значение, которое было установлено для указателя опасности ❸. Подобное использование указателей опасности основано на факте безопасного применения значения указателя после того, как объект, на который он ссылался, был удален. Если воспользоваться реализацией `new` и `delete` по умолчанию, то чисто технически будет получено неопределенное поведение, поэтому нужно либо гарантировать, что для имеющейся реализации это вполне допустимо, либо задействовать специализированный распределитель, допускающий подобное использование.

После установки указателя опасности можно продолжить выполнение кода функции `pop()`, уверившись, что никакой другой поток не удалит узлы прямо у вас из-под носа. Или же почти уверившись: при каждой перезагрузке `old_head` необходимо

обновлять указатель опасности, прежде чем разыменовывать только что прочитанное значение указателя. Как только узел будет извлечен из списка, указатель опасности можно очистить. Если других указателей опасности, ссылающихся на ваш узел, нет, его можно спокойно удалить. В противном случае его следует добавить к списку узлов, ожидающих удаления. Полная реализация функции `pop()`, использующей эту схему, показана в листинге 7.6.

Листинг 7.6. Реализация функции pop(), применяющей указатели опасности

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();
    do
    {
        node* temp;
        do                                    ❶ Выполнение цикла, пока указатель
        {                                       опасности не будет 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);                        ❷ Очистка указателя опасности по завершении
    std::shared_ptr<T> res;
    if(old_head)
    {
        res.swap(old_head->data);
        if(outstanding_hazard_pointers_for(old_head))   ❸ Проверка наличия
        {                                                 указателей опасности,
            reclaim_later(old_head);  ❹                   ссылающихся на узел,
        }                                                 перед удалением узла
        else
        {
            delete old_head;      ❺
        }
        delete_nodes_with_no_hazards();  ❻
    }
    return res;
}
```

Прежде всего цикл установки указателя опасности был перенесен во внешний цикл, предназначенный для перечитывания `old_head` в случае неудачного завершения операции сравнения — замены ❶. Операция `compare_exchange_strong()` используется здесь потому, что работа выполняется внутри цикла `while`: ложный отказ при выполнении `compare_exchange_weak()` привел бы к ненужной переустановке указателя опасности. Тем самым гарантируется правильная установка указателя опасности до разыменования `old_head`. После заявления прав на узел указатель опасности можно очистить ❷. Получив узел, нужно проверить, не ссылаются ли на него указатели опасности, принадлежащие другим потокам ❸. Если такие указатели есть, узел пока

удалять нельзя, а нужно поместить в список узлов, утилизируемых позже ❹. В противном случае его можно удалить прямо сейчас ❺. И наконец, вставляется вызов функции для проверки наличия узлов, для которых приходилось вызывать функцию `reclaim_later()`. Если указателей опасности, ссылающихся на такие узлы, больше нет, их можно спокойно удалить ❻. А те узлы, на которые по-прежнему имеются указатели опасности, останутся на попечении другого потока, вызывающего `pop()`.

В новых функциях `get_hazard_pointer_for_current_thread()`, `reclaim_later()`, `outstanding_hazard_pointers_for()` и `delete_nodes_with_no_hazards()` скрыто множество деталей, поэтому подробнее разберем их и посмотрим, как они работают.

Конкретная схема распределения памяти под экземпляры указателей опасности, принадлежащие потокам, использующаяся в функции `get_hazard_pointer_for_current_thread()`, для логики программы неважна (хотя, как будет показано позже, она может повлиять на эффективность работы). Пока обойдемся простой структурой — массивом фиксированного размера с парами, составленными из идентификаторов потоков и указателей. Функция `get_hazard_pointer_for_current_thread()` ищет в данном массиве первый свободный элемент и устанавливает для имеющейся в нем записи идентификатора значение идентификатора текущего потока. Когда поток завершается, элемент освобождается за счет переустановки записи идентификатора на сконструированное по умолчанию значение `std::thread::id()`. Все это показано в листинге 7.7.

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()))
            {
                hp=&hazard_pointers[i];
                break;
            }
        }
```

Попытка заявления прав на владение указателем опасности

```
        if(!hp)    ←❶
        {
            throw std::runtime_error("No hazard pointers available");
        }
    }
    std::atomic<void*>& get_pointer()
    {
        return hp->pointer;
    }
    ~hp_owner()     ←❷
    {
        hp->pointer.store(nullptr);
        hp->id.store(std::thread::id());
    }
};
std::atomic<void*>& get_hazard_pointer_for_current_thread()  ←❸
{
    thread_local static hp_owner hazard;  ←┐  У каждого потока имеется
    return hazard.get_pointer();  ←┐       │  свой указатель опасности
}                                  ❺      ❹
```

Реализация самой функции `get_hazard_pointer_for_current_thread()`, казалось бы, проста ❸: в ней есть переменная `thread_local` типа `hp_owner` ❹, в которой хранится указатель опасности для текущего потока. Затем указатель возвращается из этого объекта ❺. Работает это следующим образом: в первый раз, когда *каждый поток* вызывает эту функцию, создается новый экземпляр `hp_owner`. Далее конструктор для этого нового экземпляра ❶ находит в таблице, состоящей из пар «владелец — указатель», запись без владельца. Для проверки отсутствия владельца у записи и утверждения прав владения ею для текущего потока задействуется функция `compare_exchange_strong()` ❷. Если выполнение `compare_exchange_strong()` завершается неудачей, значит, этой записью владеет другой поток, и происходит перемещение на другую запись. Если замена удалась, значит, записью успешно завладел текущий поток, поэтому она сохраняется и поиск останавливается ❸. Если поиск подошел к концу списка, а свободной записи так и не нашлось ❹, значит, потоков, использующих указатели опасности, слишком много, поэтому выдается исключение.

После создания для данного потока экземпляра `hp_owner` дальнейшие обращения проходят намного быстрее, так как указатель кэшируется, следовательно, повторно сканировать таблицу уже не нужно.

Когда каждый поток завершается, экземпляр `hp_owner`, если он был для него создан, уничтожается. И прежде, чем идентификатор владельца будет установлен на `std::thread::id()`, что позволит другому потоку чуть позже воспользоваться записью, деструктор переустанавливает значение указателя на `nullptr` ❺.

С такой реализацией `get_hazard_pointer_for_current_thread()` реализация функции `outstanding_hazard_pointers_for()` упрощается и сводится к сканированию таблицы указателей опасности в поисках записей:

```
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;
}
```

Можно даже не проверять, есть ли владелец у каждой записи: у бесхозных записей будет нулевой указатель, поэтому сравнение в любом случае вернет `false`, что упрощает код.

Теперь функции `reclaim_later()` и `delete_nodes_with_no_hazards()` могут работать с простым связанным списком: `reclaim_later()` добавляет к нему узлы, а `delete_nodes_with_no_hazards()` сканирует его, удаляя записи без указателей опасности. Эта реализация показана в листинге 7.8.

Листинг 7.8. Простая реализация функций утилизации

```
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):   ❶
        data(p),
        deleter(&do_delete<T>),
        next(0)
    {}
    ~data_to_reclaim()
    {
        deleter(data);   ❷
    }
};
std::atomic<data_to_reclaim*> nodes_to_reclaim;
void add_to_reclaim_list(data_to_reclaim* node)  ❸
{
    node->next=nodes_to_reclaim.load();
    while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}
template<typename T>
void reclaim_later(T* data)    ❹
{
    add_to_reclaim_list(new data_to_reclaim(data));  ❺
}
void delete_nodes_with_no_hazards()
{
```

```
    data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);  ❻
    while(current)
    {
        data_to_reclaim* const next=current->next;
        if(!outstanding_hazard_pointers_for(current->data))  ❼
        {
            delete current;      ❽
        }
        else
        {
            add_to_reclaim_list(current);  ❾
        }
        current=next;
    }
}
```

Прежде всего, надеюсь, вы заметили, что `reclaim_later()` не обычная функция, а шаблон функции ❹. Дело в том, что указатели опасности являются механизмом общего назначения, поэтому не хотелось бы привязываться только к узлам стека. Для хранения указателей уже использовался тип `std::atomic<void*>`. Поэтому требуется справляться с любым типом указателя, но воспользоваться `void*` нельзя, так как по возможности нужно удалять элементы данных, а это требует реального типа указателя. Вскоре будет показано, что с этим неплохо справляется конструктор `data_to_reclaim`: `reclaim_later()` создает для вашего указателя новый экземпляр `data_to_reclaim` и добавляет его к списку утилизации ❺. Сама функция `add_to_reclaim_list()` ❸, подобно ранее встречавшейся конструкции, является простым циклом с запуском функции `compare_exchange_weak()` в отношении головного элемента списка.

Вернемся к конструктору `data_to_reclaim` ❶, он также является шаблоном. Данные, которые требуется удалить, он сохраняет в виде указателя `void*` в компоненте `data`, после чего сохраняет указатель в соответствующем экземпляре `do_delete()` — простой функции, приводящей предоставленный тип `void*` к выбранному типу указателя, а затем удаляет указываемый объект. Шаблон `std::function<>` служит безопасной оболочкой указателя на функцию, чтобы впоследствии деструктор `data_to_reclaim` смог удалить данные вызовом сохраненной функции ❷.

Деструктор `data_to_reclaim` при добавлении узлов к списку не вызывается, его вызов происходит, когда на данный узел не ссылается ни один указатель опасности. За это отвечает функция `delete_nodes_with_no_hazards()`.

Сначала функция `delete_nodes_with_no_hazards()` заявляет права на весь список узлов, подлежащих утилизации, вызывая простую функцию `exchange()` ❻. Этот простой, но очень важный шаг гарантирует, что попытка утилизации данного конкретного набора узлов будет предприниматься только этим потоком. Другие потоки теперь могут свободно добавлять к списку последующие узлы или даже пытаться утилизировать их, не влияя на текущий поток.

Пока в списке еще есть узлы, поочередно проверяется каждый из них, чтобы определить, не осталось ли указателей опасности ❼. Если их нет, можно свободно удалить запись, очистив хранящиеся данные ❽. В противном случае элемент возвращается в список для последующей утилизации ❾.

Хотя эта простая реализация вполне справляется с безопасной утилизацией удаленных узлов, она существенно увеличивает издержки процесса. Сканирование массива указателей опасности требует проверки атомарных переменных `max_hazard_pointers`, проводимой при каждом вызове функции `pop()`. По своей природе атомарные операции весьма неспешны — зачастую на центральных процессорах настольных машин они выполняются в 100 раз медленнее, чем эквивалентные им неатомарные операции, что превращает вызов `pop()` в весьма затратную операцию. Список указателей опасности сканируется в поисках узла, намеченного к удалению. Также этот список сканируется для каждого узла в списке ожидания. Понятно, что этот замысел нельзя признать удачным. Количество узлов в списке вполне может достигать числа, указанного в `max_hazard_pointers`, и все они проверяются на соответствие числу сохраненных указателей опасности (`max_hazard_pointers`). Нет, должно же быть более подходящее решение!

Более удачные стратегии утилизации с использованием указателей опасности

Более удачное решение, конечно же, есть. То, что было показано, — примитивная реализация указателей опасности, помогающая раскрыть суть технологии. Первое, что можно сделать, — променять память на производительность. Вместо проверки каждого узла в списке утилизации при каждом вызове `pop()` вообще не будут предприниматься попытки утилизации каких-либо узлов, пока их число не превысит значения, хранящегося в `max_hazard_pointers`. Тем самым гарантируется возможность утилизации как минимум одного узла. Если дожидаться, пока в списке не окажется `max_hazard_pointers+1` узлов, существенного улучшения не произойдет. Как только наберется `max_hazard_pointers` узлов, попытка утилизации узлов будет предприниматься при большинстве вызовов `pop()`, и от этого лучше не станет. Но если дождаться, когда в списке окажется `2*max_hazard_pointers` узлов, то из них активными будут не более `max_hazard_pointers` узлов, следовательно, будет гарантирована возможность утилизации как минимум `max_hazard_pointers` узлов. Затем состоится по меньшей мере `max_hazard_pointers` вызовов `pop()`, прежде чем снова будет предпринята попытка утилизации каких-либо узлов. Это более удачное решение. Вместо того чтобы при каждом вызове `pop()` проверять `max_hazard_pointers` узлов (и при этом еще не факт, что удастся какие-то из них утилизировать), проверка `2*max_hazard_pointers` узлов выполняется через каждые `max_hazard_pointers` вызовов `pop()` и утилизируются минимум `max_hazard_pointers` узлов. По сути, на каждый вызов `pop()` приходится проверка двух узлов и утилизация одного узла.

Но даже у этого решения есть недостатки, кроме повышенного потребления памяти за счет разросшегося списка утилизации и большего количества потенциально утилизируемых узлов: теперь придется подсчитывать узлы в списке утилизации, что подразумевает использование атомарного счетчика, и по-прежнему за доступ к списку утилизации будут конкурировать несколько потоков. Если есть лишняя память, можно променять ее повышенный расход на применение еще более эффективной схемы утилизации, при которой каждый поток хранит в своей локальной переменной собственный список утилизации. Тогда атомарные переменные не понадобятся ни для счетчика, ни для доступа к списку. Вместо этого память будет распределена

под `max_hazard_pointers*max_hazard_pointers` узлов. Если поток завершится до утилизации всех его узлов, их, как и раньше, можно будет сохранять в глобальном списке и добавлять к локальному списку следующего потока, выполняющего утилизацию.

Еще один недостаток указателей опасности состоит в том, что они защищены патентом, поданным компанией IBM[1]. Хотя я считаю, что срок действия этого патента уже истек, но, если программный продукт создается для использования в стране, где такие патенты действительны, стоит воспользоваться услугами юриста, чтобы он провел проверку, или же озаботиться наличием подходящего лицензионного соглашения. Эта же проблема актуальна для многих технологий утилизации памяти, свободных от блокировок: исследования в данной области ведутся весьма активно, поэтому крупные компании обзаводятся патентами где только можно. Может возникнуть вполне справедливый вопрос: зачем я столько страниц отвел описанию технологии, которую некоторые просто не смогут использовать? Во-первых, может появиться возможность воспользоваться технологией, не тратясь на лицензию. Например, если программа разрабатывается с открытым кодом на условиях лицензии GPL[2], она может подпадать под заявление компании IBM об отказе от правопритязаний[3]. Во-вторых, что более важно, рассматривая технологию, мы познакомились с очень важными понятиями, которые требуется осмыслить при написании кода, свободного от блокировок, например с затратностью атомарных операций. И наконец, есть предложения учесть указатели опасности, пересматривая в будущем стандарт языка C++[4], поэтому стоит узнать, как они работают, даже если есть надежда в будущем задействовать их реализацию, предоставленную поставщиком вашего компилятора.

А есть ли незапатентованные технологии утилизации памяти, подходящие для применения в коде, свободном от блокировок? К счастью, есть. Одной из них является подсчет ссылок.

7.2.4. Определение используемых узлов путем подсчета ссылок

В подразделе 7.2.2 было показано, что проблема с удалением узлов сводится к определению того, к каким узлам все еще есть обращения читающих потоков. Если бы удалось точно определить, на какие узлы есть ссылки и когда к этим узлам не об-

---

1 *Maged M. M.* U. S. Patent and Trademark Office application number 20040107227, Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation.

2 GNU General Public License: http://www.gnu.org/licenses/gpl.html.

3 IBM Statement of Non-Assertion of Named Patents Against OSS: http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf.

4 P0566: Proposed Wording for Concurrent Data Structures: Hazard Pointer and ReadCopyUpdate (RCU), Michael Wong, Maged M. Michael, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0566r5.pdf.

ращаются никакие потоки, их можно было бы спокойно удалить. С указателями опасности эта проблема решается за счет сохранения списка используемых узлов. С подсчетом ссылок она решается путем сохранения числа потоков, обращающихся к каждому узлу.

Несмотря на обманчивую привлекательность и простоту этой идеи, реализовать ее на практике непросто. Прежде всего можно подумать, что подойдет что-то вроде `std::shared_ptr<>`: в конце концов, это и есть указатель с подсчетом ссылок. К сожалению, хотя некоторые операции над `std::shared_ptr<>` и являются атомарными, они не гарантируют свободы от блокировок. Хотя сам по себе класс `std::shared_ptr<>` ничем не отличается от любых других, выполняющих операции над атомарными типами, он предназначен для использования в множестве разных контекстов, и придание атомарным операциям свободы от блокировок не обошлось бы без повышенных издержек во всех вариантах применения этого класса. Если на вашей платформе предоставляются реализации, для которых функция `std::atomic_is_lock_free(&some_shared_ptr)` возвращает `true`, то все проблемы утилизации памяти отпадают. Просто воспользуйтесь для списка объектами `std::shared_ptr<node>`, как показано в листинге 7.9. Обратите внимание на необходимость очистить указатель `next` от изъятого узла во избежание вероятности глубокого вложения при уничтожении узлов `node`, когда уничтожается последний указатель `std:: shared_ptr`, ссылающийся на данный узел.

Листинг 7.9. Свободный от блокировок стек с использованием свободной от блокировок реализации класса 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=std::atomic_load(&head);
        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,std::atomic_load(&old_head->next)));
        if(old_head) {
            std::atomic_store(&old_head->next,std::shared_ptr<node>());
            return old_head->data;
        }
        return std::shared_ptr<T>();
    }
    ~lock_free_stack(){
        while(pop());
    }
};
```

Мало того что в реализации `std::shared_ptr<>` атомарные операции без блокировки встречаются крайне редко, но и постоянно помнить об использовании атомарных операций нелегко. При наличии доступной реализации помочь в этом может спецификация Concurrency TS, потому что в соответствии с ней под заголовком `<experimental/atomic>` предоставляется класс `std::experimental::atomic_shared_ptr<T>`. Он почти эквивалентен теоретическому классу `std::atomic<std::shared_ptr<T>>`, за исключением того, что `std::shared_ptr<T>` нельзя применять с `std::atomic<>`, так как он имеет нетривиальную семантику копирования, гарантирующую корректную обработку счетчика ссылок. Класс `std::experimental::atomic_shared_ptr<T>` обрабатывает подсчет ссылок правильно, обеспечивая при этом выполнение атомарных операций. Как и другие атомарные типы, рассмотренные в главе 5, он может или не может быть свободным от блокировок в любой конкретной реализации. В связи с этим код листинга 7.9 можно переписать так, как показано в листинге 7.10. Посмотрите, как все упрощается, когда не нужно помнить о включении вызовов `atomic_load` и `atomic_store`.

Листинг 7.10. Реализация стека с использованием std::experimental::atomic_shared_ptr<>

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::experimental::atomic_shared_ptr<node> next;
        node(T const& data_):
            data(std::make_shared<T>(data_))
        {}
    };
    std::experimental::atomic_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(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
```

```
        std::shared_ptr<node> old_head=head.load();
        while(old_head && !head.compare_exchange_weak(
                old_head,old_head->next.load()));
        if(old_head) {
            old_head->next=std::shared_ptr<node>();
            return old_head->data;
        }
        return std::shared_ptr<T>();
    }
    ~lock_free_stack(){
        while(pop());
    }
};
```

В весьма вероятном случае, когда применяемая вами реализация `std::shared_ptr<>` не свободна от блокировок либо в ней не предоставляется свободный от блокировок класс `std::experimental::atomic_shared_ptr<>`, справляться с подсчетом ссылок придется самостоятельно.

В одном из возможных приемов для каждого узла используется не один, а два счетчика ссылок: внутренний и внешний. Сумма их значений — это общее количество ссылок на узел. Внешний счетчик хранится вместе с указателем на узел и увеличивается на единицу при каждом чтении указателя. Когда код завершит чтение узла, показатель внутреннего счетчика уменьшится на единицу. Когда простая операция чтения указателя завершится, она оставит внешний счетчик увеличенным на единицу, а внутренний счетчик — уменьшенным на единицу.

Когда внешнее сопряжение счетчика с указателем станет ненужным, то есть узел окажется недоступным из места, доступного нескольким потокам, внутренний счетчик увеличится на значение внешнего счетчика за вычетом единицы, а внешний счетчик будет сброшен. Когда внутренний счетчик обнулится, ссылок на узел не останется и его можно будет спокойно удалить. Однако применение атомарных операций для обновления совместно используемых данных останется все таким же важным.

Теперь рассмотрим реализацию свободного от блокировок стека, применяющего данную технологию для обеспечения утилизации узлов, только когда ее выполнение безопасно. В листинге 7.11 показаны внутренняя структура данных и простая и понятная реализация функции `push()`.

Листинг 7.11. Помещение узла в свободный от блокировок стек с использованием раздельных счетчиков ссылок

```
template<typename T>
class lock_free_stack
{
private:
    struct node;
    struct counted_node_ptr   ←❶
    {
        int external_count;
        node* ptr;
    };
    struct node
```

```
    {
        std::shared_ptr<T> data;                    ❷
        std::atomic<int> internal_count;  ◀┘
        counted_node_ptr next;          ◀──❸
        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };
    std::atomic<counted_node_ptr> head;  ◀──❹
public:
    ~lock_free_stack()
    {
        while(pop());
    }
    void push(T const& data)  ◀──❺
    {
        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));
    }
};
```

Начнем с того, что внешний счетчик заключен вместе с указателем на узел в структуру `counted_node_ptr` ❶. Затем она может использоваться в структуре `node` для указателя `next` ❸ наряду с внутренним счетчиком ❷. Поскольку `counted_node_ptr` является простой структурой `struct`, ею можно воспользоваться с шаблоном `std::atomic<>` для представления входящего в список головного элемента `head` ❹.

На платформах, поддерживающих операцию сравнения и обмена двойного слова, размер этой структуры достаточно мал для того, чтобы тип `std::atomic<counted_node_ptr>` был свободен от блокировок. Если на вашей платформе это не так, стоит, наверное, воспользоваться версией `std::shared_ptr<>` из листинга 7.9, поскольку в `std::atomic<>`, если тип достаточно велик для атомарных инструкций платформы, для гарантии атомарности будет использоваться мьютекс, показывая, что ваш якобы свободный от блокировок алгоритм таковым, по сути, не является. Если вы согласны ограничить размер счетчика и знаете, что ваша платформа имеет лишние разряды в указателе (например, из-за того, что адресное пространство занимает всего 48 разрядов, а указатель состоит из 64 разрядов), то можно сделать иначе и сохранить счетчик внутри лишних разрядов указателя, чтобы все это поместилось в одном машинном слове. Но для таких приемов нужно обладать конкретными сведениями о платформе, а это выходит за рамки тем, рассматриваемых в данной книге.

Функция `push()` относительно проста ❺. В ней создается объект `counted_node_ptr`, ссылающийся на только что распределенный узел со связанными данными, а в качестве значения `next` узла `node` устанавливается текущее значение `head`. Затем с помощью функции `compare_exchange_weak()` можно установить значение `head`, как это делалось в коде предыдущих листингов. Счетчики устанавливаются таким образом, чтобы внутренний счетчик `internal_count` имел нулевое значение, а внешний

счетчик external_count содержал единицу. Поскольку это новый узел, на него пока есть только одна внешняя ссылка (сам указатель head).

Как обычно, все сложности высвечиваются в реализации функции pop(), показанной в листинге 7.12.

Листинг 7.12. Извлечение узла из свободного от блокировок стека с использованием разделенного счетчика ссылок

```
template<typename T>
class lock_free_stack
{
private:
    // остальные части такие же, как в листинге 7.11
    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)); ❶
        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;    ❷
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next)) ❸
            {
                std::shared_ptr<T> res;      ❹
                res.swap(ptr->data);
                int const count_increase=old_head.external_count-2; ❺
                if(ptr->internal_count.fetch_add(count_increase)==  ❻
                    -count_increase)
                {
                    delete ptr;  ❼
                }
                return res;
            }
            else if(ptr->internal_count.fetch_sub(1)==1)
            {
                delete ptr;  ❽
            }
        }
    }
};
```

На этот раз, загрузив значение head, нужно в первую очередь увеличить счетчик внешних ссылок на узел head, показав, что на него есть ссылка, и гарантировав тем самым, что его разыменование пройдет безопасно. Если разыменовать указатель *до* увеличения показания счетчика ссылок на единицу, другой поток сможет освободить узел до того, как к нему произойдет обращение, оставив подвисший указатель. В этом и кроется главная причина использования разделенного счетчика ссылок: повышая значение внешнего счетчика ссылок на единицу, удается гарантировать, что он останется достоверным на протяжении всего обращения к нему. Увеличение показания на единицу происходит в цикле compare_exchange_strong() ❶, где выполняются сравнение и установка значений всей структуры, чтобы гарантировать, что за это время указатель не был изменен другим потоком.

После увеличения показания счетчика на единицу можно свободно разыменовать поле ptr того значения, которое было загружено из head, чтобы получить доступ к указываемому узлу ❷. Если указатель нулевой, значит, это конец списка, записей в нем больше нет. Если указатель не является нулевым, можно попытаться удалить узел, вызвав в отношении head функцию compare_exchange_strong() ❸.

Если выполнение функции compare_exchange_strong() завершится успешно, значит, узлом удалось завладеть и теперь, готовясь к возвращению данных, их можно выгрузить из узла ❹. Тем самым гарантируется, что данные не будут сохраняться только потому, что у других обращающихся к стеку потоков все еще есть указатели на этот узел. Затем можно сложить показания внешнего и внутреннего счетчиков ссылок на узел с помощью атомарной операции fetch_add ❻. Если счетчик ссылок обнулился, *предыдущее* значение (возвращенное функцией fetch_add) было противоположно по знаку только что прибавленному, и в этом случае узел можно удалить. Важно отметить, что прибавленное значение на две единицы меньше значения внешнего счетчика ❺: узел был удален из списка, поэтому счетчик уменьшен на единицу, из этого потока обращений к узлу больше не будет — счетчик сбрасывается еще на единицу. Независимо от того, удален узел или нет, работа завершена и можно возвратить данные ❼.

Если выполнить операцию сравнения — обмена не удается ❸, значит, другой поток уже удалил узел или добавил к стеку новый узел. В любом случае нужно все сделать заново со свежим значением head, возвращенным вызовом операции сравнения — обмена. Но сначала нужно уменьшить на единицу счетчик ссылок на узел, который пытались удалить. Этот поток больше не будет к нему обращаться. Если это был последний поток, удерживающий ссылку (поскольку узел был удален из стека другим потоком), у внутреннего счетчика ссылок будет значение 1, следовательно, вычитание числа 1 приведет к его обнулению. В этом случае узел можно удалить прямо сейчас, до прохождения цикла ❽.

До сих пор для всех атомарных операций использовалось упорядочение доступа к памяти по умолчанию: std::memory_order_seq_cst. В большинстве систем оно более затратно в смысле времени выполнения и издержек синхронизации, чем другие способы упорядочения доступа к памяти, в некоторых системах эта разница особенно заметна. Теперь, выправив логику структуры данных, можно подумать об ослаблении требований к упорядочению доступа к памяти: не хотелось бы при ис-

пользовании стека нести неоправданные издержки. Прежде чем завершить работу со стеком и перейти к созданию свободной от блокировок очереди, давайте исследуем операции со стеком и зададимся вопросом: можно ли воспользоваться менее строгим упорядочением доступа к памяти при выполнении некоторых операций, сохранив уровень безопасности?

7.2.5. Применение модели памяти к свободному от блокировок стеку

Прежде чем изменять порядок доступа к памяти, нужно исследовать операции и понять, какими должны быть отношения между ними. Затем можно вернуться к определению минимально строгого упорядочения доступа к памяти, обеспечивающего такие отношения. Для этого следует посмотреть на складывающуюся ситуацию с позиции потоков в нескольких различных сценариях. Наипростейший из возможных сценариев возникает, когда один поток помещает элемент данных в стек, а другой поток чуть позже извлекает из него элемент данных. С него и начнем.

В данном простом случае участвуют три вида данных. Первым является структура `counted_node_ptr`, используемая для передачи данных, имеется в виду `head`. Вторым — структура узла, на который ссылается `head`. И третьим — элемент данных, на который указывает этот узел.

Поток, выполняющий функцию `push()`, сначала создает элемент данных и узел `node`, а затем устанавливает значение `head`. Поток, выполняющий функцию `pop()`, сначала загружает значение `head`, затем выполняет в цикле операцию сравнения — обмена в отношении `head`, чтобы увеличить показание счетчика ссылок на единицу, после чего считывает структуру узла для получения значения `next`. Требуемое отношение можно увидеть прямо здесь: значение `next` — это простой неатомарный объект, поэтому для его безопасного чтения должно существовать отношение «происходит до» между сохранением (выполняемым потоком, помещающим элемент) и загрузкой (выполняемой потоком, извлекающим элемент). Поскольку единственной атомарной операцией в `push()` является `compare_exchange_weak()`, а для получения отношения «происходит до» между потоками требуется операция освобождения (`release`), для функции `compare_exchange_weak()` нужно задать упорядочение `std::memory_order_release` или выбрать еще более строгий вариант. Если вызов `compare_exchange_weak()` завершается неудачно, ничего не изменяется и выполнение цикла продолжается, и в данном случае требуется только упорядочение с семантикой `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 как насчет кода функции pop()? Чтобы получить нужное отношение «происходит до», следует, прежде чем обращаться к next, обзавестись операцией с семантикой не слабее std::memory_order_acquire. Указатель, который разыменовывается для доступа к полю next, — это старое значение, считанное функцией compare_exchange_strong(), вызванной из increase_head_count(), поэтому в случае ее успешного выполнения нужно применить именно такое упорядочение. Как и в вызове push(), если обмен оказался неудачным, цикл просто повторяется, в этом случае можно воспользоваться нестрогим упорядочением:

```
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() завершится успешно, станет понятно, что в прочитанном значении было поле ptr со значением, которое теперь хранится в old_counter. Поскольку сохранение в push() было операцией освобождения, а данный вызов compare_exchange_strong() является операцией захвата, то сохранение «синхронизируется с» загрузкой и получается отношение «происходит до». Следовательно, сохранение в поле ptr при выполнении функции push() «происходит до» обращения к ptr->next в функции pop(), чем и обеспечивается безопасность.

Следует отметить, что при таком анализе порядок доступа к памяти в начальном вызове head.load() не имеет значения, поэтому для него можно спокойно применить семантику std::memory_order_relaxed.

Теперь на очереди операция compare_exchange_strong(), устанавливающая для head значение old_head.ptr->next. Нужно ли что-то от этой операции, чтобы гарантировать целостность данных в этом потоке? Если обмен завершится успешно, произойдет обращение к ptr->data, следовательно, нужно обеспечить, чтобы сохранение в ptr->data в потоке, выполняющем функцию push(), произошло до загрузки. Но такая гарантия уже существует: операция захвата в increase_head_count() гарантирует наличие отношения «синхронизируется с» между сохранением в потоке, выполняющем push(), и операцией сравнения — обмена. Поскольку сохранение данных в потоке, выполняющем push(), находится до сохранения в head, а вызов increase_head_count() — до загрузки ptr->data, получается отношение «происходит до» и все складывается удачно, даже если для операции сравнения — обмена в pop() используется семантика std::memory_order_relaxed. Единственным другим местом, где изменятся ptr->data, является изучаемый в данный момент вызов функции swap(), и никакой иной поток не может работать с тем же самым узлом. В этом, собственно, и заключается смысл проведения операции сравнения — обмена.

Если выполнение функции compare_exchange_strong() завершится неудачей, новое значение old_head не будет затрагиваться до следующего прохождения цикла, и так как уже решено, что для increase_head_count() вполне хватит применения семантики std::memory_order_acquire, то здесь также будет вполне достаточно задействовать семантику std::memory_order_relaxed.

А что же другие потоки? Нужно ли здесь что-то более строгое, чтобы гарантировать безопасность их работы? Ответ отрицательный, поскольку head изменяется только при выполнении операций сравнения — обмена. Поскольку они относятся к операциям чтения — изменения — записи, из них формируется часть последовательности освобождений, начинающейся операцией сравнения — обмена в функции push(). Поэтому compare_exchange_weak() в push() «синхронизируется с» вызовом compare_exchange_strong() в increase_head_count(), в котором выполняется чтение сохраненного значения, даже если тем временем изменения в head будут внесены множеством других потоков.

Мы почти у цели, осталось только разобраться с операциями fetch_add() для изменения счетчика ссылок. Поток, дошедший до возвращения данных из узла, может продолжить свою работу в полной уверенности, что никакой другой поток не способен изменить данные узла. Но любой поток, которому не удалось извлечь данные узла, знает, что они были изменены другим потоком: поток, имевший успех, использовал для извлечения элемента данных, на который делалась ссылка, функцию swap(). Поэтому, чтобы избежать состояния гонки за данными, нужно гарантировать, что функция swap() выполняется до удаления delete. Это несложно сделать, воспользовавшись для fetch_add(), вызываемой в ветви успешного возвращения, семантикой std::memory_order_release, а для fetch_add(), вызываемой в ветви повторного прохождения цикла, — семантикой std::memory_order_acquire. Но все же это излишне: только один поток выполняет delete (тот самый, что устанавливает счетчик в ноль), следовательно, операцию захвата нужно выполнить только ему. К счастью, поскольку функция fetch_add() является операцией чтения — изменения — записи, она составляет часть последовательности освобождений, поэтому можно достичь желаемого с помощью дополнительного вызова функции load(). Если в ветви повторного прохождения цикла счетчик в результате уменьшения на единицу обнуляется, в ней же можно перезагрузить счетчик ссылок с семантикой std::memory_order_acquire, чтобы обеспечить нужное отношение «синхронизируется с», а для самой функции fetch_add() можно воспользоваться семантикой std::memory_order_relaxed. В окончательном виде реализация стека с новой версией pop() показана в листинге 7.13.

Листинг 7.13. Свободный от блокировок стек с подсчетом ссылок и нестрогими 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;
            }
        }
    }
};
```

Наш упорный труд завершился улучшением стека. Продуманное применение менее строгих операций позволило повысить производительность без внесения каких-либо ошибок. Как видите, теперь код реализации `pop()` составляет 37 строк вместо восьми строк в аналогичной реализации `pop()` для стека с блокировками из листинга 6.1 и семи строк базового свободного от блокировок стека без управления доступом к памяти из листинга 7.2. Когда мы перейдем к исследованию приемов создания свободной от блокировок очереди, нам встретится похожая схема. Основные сложности в свободном от блокировок коде будут связаны с управлением доступом к памяти.

7.2.6. Создание потокобезопасной очереди без блокировок

Создать очередь немного сложнее, чем стек, поскольку операции `push()` и `pop()` в очереди обращаются к различным частям этой структуры данных, а в стеке — к одному и тому же узлу `head`. Следовательно, для нее нужна и другая синхронизация. Требуется, чтобы изменения, сделанные в одном конце, были видимы обращениям к другому концу. Но структура функции `try_pop()` для очереди в листинге 6.6 не сильно отличается от структуры функции `pop()` для простого свободного от блокировок стека в листинге 7.2, поэтому вполне разумно полагать, что код, свободный от блокировок, также не будет иметь существенных различий. Посмотрим, насколько это справедливо.

Если взять за основу код листинга 6.6, понадобятся два указателя на node: один — для головы списка (head), второй — для хвоста (tail). Обращение к ним будет из нескольких потоков, поэтому лучше бы они были атомарными, чтобы можно было обойтись без соответствующих мьютексов. Начнем с этого небольшого изменения и посмотрим, к чему оно приведет. Результат показан в листинге 7.14.

Листинг 7.14. Свободная от блокировок очередь с одним поставщиком 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())       ←❶
        {
            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);  ❷
        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;                              ❸
        node* const old_tail=tail.load();              ❹
        old_tail->data.swap(new_data);                 ❺
        old_tail->next=p;                              ❻
        tail.store(p);                                 ❼
    }
};
```

На первый взгляд, особых недостатков нет, и если функция `push()` вызывается в определенный момент всего одним потоком, код кажется вполне подходящим. В данном случае важно, чтобы между `push()` и `pop()` было отношение «происходит до», гарантирующее безопасность извлечения данных `data`. Сохранение в `tail` ❼ синхронизировано с загрузкой из `tail` ❶: сохранение указателя на `data` предыдущего узла ❺ находится в последовательности до сохранения в `tail`, а загрузка из `tail` находится в последовательности до загрузки из указателя на данные `data` ❷, значит, сохранение в `data` происходит до загрузки и все в порядке. То есть получается вполне подходящая *очередь с одним поставщиком и одним потребителем* (single-producer, single-consumer, SPSC).

Проблемы начинаются, когда несколько потоков одновременно вызывают функции `push()` или `pop()`. Сначала рассмотрим функцию `push()`. Если есть два потока, одновременно вызывающие `push()`, то оба они распределяют память под новые узлы, чтобы получился новый пустой узел ❸, считывают *одно и то же* значение для `tail` ❹ и, соответственно, при установке значений указателей `data` и `next` ❺ и ❻ обновляют компонентные данные одного и того же узла. А это состояние гонки за данными!

Аналогичные проблемы связаны и с функцией `pop_head()`. Если два потока вызывают ее одновременно, то оба они прочитают одно и то же значение `head`, а затем запишут вместо старого значения одно и то же значение указателя `next`. Теперь в обоих потоках будет считаться, что извлечен один и тот же узел, а это прямой путь к созданию аварийной ситуации. Нужно не только обеспечить, чтобы функция `pop()` применялась к заданному элементу только одним потоком, но и принять меры к тому, чтобы другие потоки могли свободно обращаться к компоненту `next`, принадлежащему узлу, считанному ими из `head`. С точно такой же проблемой приходилось сталкиваться в функции `pop()` для свободного от блокировок стека, поэтому любые ее решения можно применить и в данном случае.

Таким образом, проблему `pop()` можно считать решенной, а как насчет `push()`? Здесь проблема в том, что для получения между `push()` и `pop()` нужного отношения «происходит до» элементы данных пустого узла следует заполнять до обновления указателя `tail`. Но это означает, что конкурентные вызовы функции `push()` приводят к состоянию гонки за одними и теми же данными, поскольку происходит считывание одного и того же указателя `tail`.

Функция push(), рассчитанная на работу в нескольких потоках

Одним из вариантов может стать добавление пустого узла между существующими узлами. Тогда единственной частью текущего узла `tail`, нуждающейся в обновлении, станет указатель `next`, который в таком случае можно было бы сделать атомарным. Если поток сумел успешно изменить значение указателя `next` с `nullptr` на свой новый узел, значит, он успешно добавил указатель, в противном случае ему придется начать все заново и перечитать `tail`. Для этого потребуется внести небольшие изменения в `pop()`, чтобы узлы с нулевыми указателями на данные игнорировались и выполнялось новое прохождение цикла. Недостаток здесь заключается в том, что при каждом вызове `pop()` неизменно придется удалять два узла, а также выполнять удвоенное распределение памяти.

Второй вариант таков: сделать указатель на данные атомарным и устанавливать его с применением операции сравнения — обмена. Если вызов успешен, значит, это хвостовой узел и можно спокойно устанавливать для `next` значение указателя на новый узел, а затем обновлять `tail`. Если операция сравнения — обмена завершается неудачно из-за того, что данные уже сохранены другим потоком, выполняется новый проход цикла, перечитывается `tail` и делается новая попытка. Если атомарные операции с `std::shared_ptr<>` свободны от блокировок, все в порядке. Если же нет, нужен альтернативный вариант. Можно сделать так, чтобы `pop()` возвращала `std::unique_ptr<>` (в конечном счете ссылка на объект только одна) и сохраняла данные в очереди в виде простого указателя. Тогда появится возможность сохранить ее как `std::atomic<T*>`, что обеспечит поддержку столь необходимого вызова функции `compare_exchange_strong()`. Если для поддержки в `pop()` вызовов со стороны нескольких потоков воспользоваться схемой подсчета ссылок из листинга 7.12, то `push()` приобретет следующий вид (листинг 7.15).

Листинг 7.15. Первая (неудачная) попытка переделки 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();  ←❶
        T* old_data=nullptr;
        if(old_tail->data.compare_exchange_strong(
            old_data,new_data.get()))  ←❷
        {
            old_tail->next=new_next;
            tail.store(new_next.ptr);  ←❸
            new_data.release();
            break;
        }
    }
}
```

Предполагаемое здесь состояние гонки устраняется за счет использования схемы подсчета ссылок, но с вызовами `push()` связано не только это состояние гонки. Если посмотреть на переделанную версию `push()` в листинге 7.15, обнаружится уже знакомый по стеку шаблон: загрузка атомарного указателя ❶ и его разыменование ❷. Между этими двумя операциями указатель может быть обновлен другим потоком ❸, что в итоге выльется в освобождение памяти, распределенной под узел (в `pop()`). Если оно произойдет раньше разыменования указателя, возникнет неопределенное поведение. Вот так! Захочется добавить в `tail` внешний счетчик, как делалось для `head`? Но у каждого узла уже имеется внешний счетчик в указателе `next`, принадлежащем предыдущему узлу в очереди. Наличие двух внешних счетчиков для одного и того же узла потребует внесения изменений в схему подсчета ссылок, чтобы избежать преждевременного удаления узла. Справиться с задачей можно за счет дополнительного подсчета количества внешних счетчиков внутри структуры `node` и его уменьшения при удалении каждого внешнего счетчика (наряду с прибавлением значения соответствующего внешнего счетчика к значению внутреннего счетчика). Если внутренний счетчик равен нулю, а внешние счетчики отсутствуют, узел можно спокойно удалить. Такой прием впервые был замечен в проекте Джо Сейга (Joe Seigh) *Atomic Ptr Plus Project* (http://atomic-ptr-plus.sourceforge.net/). На что становится похож код функции `push()` при использовании данной схемы, показано в листинге 7.16.

Листинг 7.16. Реализация функции push() для свободной от блокировок очереди с подсчетом 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;    <--❶
    struct node_counter
    {
        unsigned internal_count:30;
        unsigned external_counters:2;    <--❷
    };
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;  <--❸
        counted_node_ptr next;
        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
```

```
            new_count.external_counters=2;  ❹
            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);    ❺
            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(   ❻
                old_data,new_data.get()))
            {
                old_tail.ptr->next=new_next;
                old_tail=tail.exchange(new_next);
                free_external_counter(old_tail);  ❼
                new_data.release();
                break;
            }
            old_tail.ptr->release_ref();
        }
    }
};
```

Теперь в коде листинга 7.16 `tail` имеет такой же тип `atomic<counted_node_ptr>`, как и `head` ❶, а в структуре `node` есть компонент `count`, заменивший прежний компонент `internal_count` ❸. Компонент `count` представляет собой структуру, содержащую `internal_count` и дополнительный компонент `external_counters` ❷. Следует отметить, что под `external_counters` нужно выделить всего два бита, поскольку таких счетчиков может быть не более двух. За счет использования для него битового поля и определения под `internal_count` 30-разрядного значения общий размер счетчика удалось ограничить 32 битами. Тем самым раскрываются возможности применения больших значений внутреннего счетчика, и при этом гарантируется, что вся структура помещается внутри машинного слова на 32- и 64-разрядных машинах. Вскоре будет показано, что во избежание состояния гонки эти счетчики важно обновлять вместе как единое целое. Хранение структуры в пределах машинного слова повышает вероятность того, что атомарные операции на многих платформах смогут обойтись без блокировок.

Инициализация `node` выполняется за счет обнуления `internal_count` и установки для `external_counters` значения `2` ❹, поскольку, как только новый узел добавляется к очереди, на него уже имеются две ссылки — из `tail` и из указателя `next` предыдущего узла. Сама функция `push()` похожа на вариант, показанный в листинге 7.15, за ис-

ключением того, что перед разыменованием значения, загруженного из `tail`, с целью вызова функции `compare_exchange_strong()` в отношении компонента узла `data` ❻ вызывается новая функция `increase_external_count()`, повышающая значение счетчика на единицу ❺, а после этого вызывается функция `free_external_counter()` в отношении старого значения хвоста ❼.

Закончив с функцией `push()`, обратимся к `pop()`. Она показана в листинге 7.17 и представляет собой смесь логики подсчета ссылок из реализации `pop()` (см. листинг 7.12) и логики извлечения узла из очереди (см. листинг 7.14).

Листинг 7.17. Извлечение узла из свободной от блокировок очереди с использованием подсчета ссылок на хвост

```
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); ←❶
        for(;;)
        {
            increase_external_count(head,old_head);  ←❷
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                ptr->release_ref();        ←❸
                return std::unique_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))  ←❹
            {
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);       ←❺
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref(); ←❻
        }
    }
};
```

Начнем с того, что загрузка значения `old_head` предшествует входу в цикл ❶ и происходит до того, как в загруженном значении внешний счетчик увеличится на единицу ❷. Если узел `head` ничем не отличается от узла `tail`, ссылку можно освободить ❸ и возвратить нулевой указатель, поскольку данных в очереди нет. Если данные в ней есть, захочется попытаться заявить на них свои права. Это делается вызовом функции `compare_exchange_strong()` ❹. При этом, как и в случае со стеком из листинга 7.12, сравниваются выступающие как единое целое внешний счетчик

и указатель: если один из них был изменен, нужно после освобождения ссылки ❻ пройти цикл заново. Если обмен прошел успешно, значит, вы заявили, что данные в узле ваши, и теперь можете вернуть их вызывающему объекту после сброса значения внешнего счетчика по причине извлечения узла ❺. После освобождения обоих внешних счетчиков и обнуления внутреннего счетчика можно удалить и сам узел. Функции подсчета ссылок, занимающиеся всем этим, показаны в листингах 7.18–7.20.

Листинг 7.18. Освобождение ссылки на узел в очереди, свободной от 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;     ←❶
            }
            while(!count.compare_exchange_strong(   ←❷
                old_counter,new_counter,
                std::memory_order_acquire,std::memory_order_relaxed));
            if(!new_counter.internal_count &&
                !new_counter.external_counters)
            {
                delete this;   ←❸
            }
        }
    };
};
```

Реализация `node::release_ref()` претерпела незначительные изменения по сравнению с аналогичным кодом реализации функции `lock_free_stack::pop()` из листинга 7.12. Там приходилось иметь дело с единственным внешним счетчиком, и можно было воспользоваться простой функцией `fetch_sub`, а теперь следует атомарно обновлять всю структуру `count`, даже если нужно всего лишь изменить поле `internal_count` ❶. Поэтому здесь требуется применить цикл сравнения — обмена ❷. Если после уменьшения на единицу `internal_count` оба счетчика, как внутренний, так и внешний, обнуляются, значит, ссылка была последней и узел можно удалить ❸.

Листинг 7.19. Получение новой ссылки на узел в очереди, свободной от 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;
    }
};
```

В коде листинга 7.19 все по-другому. На этот раз вместо освобождения ссылки происходит получение новой ссылки и увеличение внешнего счетчика на единицу. Код функции `increase_external_count()` аналогичен коду функции `increase_head_count()` из листинга 7.13, за исключением его преобразования в статическую компонентную функцию, которая не работает с резидентным счетчиком, а получает в качестве первого параметра обновляемый внешний счетчик.

Листинг 7.20. Освобождение счетчика внешних ссылок на узел в очереди, свободной от 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;                    ←❶
            --new_counter.external_counters;
            new_counter.internal_count+=count_increase;  ←❷
        }
        while(!ptr->count.compare_exchange_strong(      ←❸
                old_counter,new_counter,
                std::memory_order_acquire,std::memory_order_relaxed));
        if(!new_counter.internal_count &&
            !new_counter.external_counters)
        {
            delete ptr;  ←❹
        }
    }
};
```

Дополнением функции `increase_external_count()` служит функция `free_external_counter()`. Ее код похож на эквивалентный код из функции `lock_free_stack::pop()`, показанный в листинге 7.12, но изменен для подсчета `external_counters`. Этой функцией в одном вызове `compare_exchange_strong()` обновляются два счетчика во всей структуре `count` ❸ точно так же, как это делалось при уменьшении на единицу `internal_count` в `release_ref()`. Значение `internal_count` обновляется, как в коде листинга 7.12 ❷, а значение `external_counters` уменьшается на единицу ❶. Если теперь *оба* значения станут нулевыми, значит, ссылок на узел больше нет и его можно спокойно удалить ❹. Чтобы не возникало состояние гонки, это должно быть сделано единым действием (поэтому для него требуется цикл сравнения — обмена). Если они будут обновляться отдельно друг от друга, то каждый из двух потоков может посчитать, что он последний, и оба станут удалять узел, что выльется в неопределенное поведение.

Теперь все доведено до рабочего состояния и опасность возникновения состояния гонки ликвидирована, однако остается нерешенной проблема производительности. После того как один поток начал операцию `push()`, успешно завершив выполнение функции `compare_exchange_strong()` в отношении `old_tail.ptr->data` (❺ из листинга 7.16), никакой другой поток не сможет выполнить операцию `push()`. Любой поток, предпринимающий такую попытку, будет видеть новое значение, а не `nullptr`, что приведет к неудачному завершению вызова `compare_exchange_strong()` и заставит этот поток пройти цикл заново. Получается активное ожидание, впустую потребляющее процессорное время. Следовательно, фактически это блокировка. Первый вызов `push()` блокирует другие потоки до своего завершения, поэтому данный код больше нельзя называть свободным от блокировки. Но и это еще не все: в отличие от ситуации, когда при наличии заблокированных потоков операционная система может отдать приоритет потоку, который удерживает блокировку на мьютексе, здесь она не способна на такое и заблокированные потоки будут впустую тратить время центрального процессора, пока первый поток не завершит свою работу. Так что нужно воспользоваться еще одним приемом из арсенала средств избавления от блокировок: ожидающий поток может помочь тому потоку, который выполняет операцию `push()`.

Создание очереди, свободной от блокировок, с помощью другого потока

Чтобы снова сделать код свободным от блокировок, нужно придумать, как заставить ожидающий поток заняться чем-то полезным, когда затормозился поток, который выполняет операцию `push()`. Один из вариантов — помочь застопорившемуся потоку, взять на себя часть его работы.

В данном случае имеется четкое представление о том, что нужно сделать: указатель `next` в хвостовом узле должен быть нацелен на новый пустой узел, и тогда должен быть обновлен сам указатель `tail`. Дело в том, что пустые узлы не отличаются друг от друга, поэтому неважно, каким пустым узлом воспользоваться — созданным потоком, который успешно поместил данные, или одним из потоков, ожидающих

возможности помещения данных. Если указатель `next` сделать в узле атомарным, то затем для установки указателя можно будет воспользоваться функцией `compare_exchange_strong()`. После установки указателя `next` для установки `tail` можно будет применить цикл `compare_exchange_weak()`, убеждаясь, что он по-прежнему ссылается на тот же самый исходный узел. Если будет не так, значит, узел обновлен каким-то другим потоком и можно прекратить попытку и повторить цикл. Для этого нужно слегка изменить код функции `pop()`, в частности, для того, чтобы загрузить указатель `next`. Изменения можно увидеть в листинге 7.21.

Листинг 7.21. Измененный код функции pop(), допускающий помощь со стороны функции 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;  ←❶
    };
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();        ←❷
            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 уже упоминалось, изменения не отличаются особой сложностью: теперь указатель `next` стал атомарным ❶, следовательно, атомарность присуща и операции `load` в строке ❷. В данном примере используется упорядочение по умолчанию `memory_order_seq_cst`, поэтому явный вызов `load()` можно опустить и полагаться на загрузку в подразумеваемом преобразовании в `counted_node_ptr`. Однако вставка

явного вызова напоминает, куда можно будет позже добавить явное упорядочение доступа к памяти.

Код функции `push()` значительно сложнее. Он показан в листинге 7.22.

Листинг 7.22. Пример кода функции push(), способствующего работе очереди, свободной от блокировок

```
template<typename T>
class lock_free_queue
{
private:
    void set_new_tail(counted_node_ptr &old_tail,   ←❶
                      counted_node_ptr const &new_tail)
    {
        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail) &&   ←❷
              old_tail.ptr==current_tail_ptr);
        if(old_tail.ptr==current_tail_ptr)   ←❸
            free_external_counter(old_tail);     ←❹
        else
            current_tail_ptr->release_ref();   ←❺
    }
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(   ←❻
                old_data,new_data.get()))
            {
                counted_node_ptr old_next={0};
                if(!old_tail.ptr->next.compare_exchange_strong(  ←❼
                    old_next,new_next))
                {                                    ❽
                    delete new_next.ptr;   ←┘
                    new_next=old_next;       ←❾
                }
                set_new_tail(old_tail, new_next);
                new_data.release();
                break;
            }
            else       ←❿
            {
                counted_node_ptr old_next={0};
                if(old_tail.ptr->next.compare_exchange_strong(  ←⓫
                        old_next,new_next))
```

```
                {                                                    ⓬
                    old_next=new_next;                    ◀──┘
                    new_next.ptr=new node;                ◀──⓭
                }
                set_new_tail(old_tail, old_next);         ◀──⓮
            }
        }
    }
};
```

Код похож на исходный код функции `push()` из листинга 7.16, но имеет ряд важных отличий. Если указатель `data` установлен ❻, нужно справиться со случаем, когда помощь исходит от другого потока. Теперь для содействия появилось условие `else` ❿.

Когда указатель `data` в узле уже установлен ❻, эта новая версия функции `push()` обновляет указатель `next`, используя функцию `compare_exchange_strong()` ❼. Функцию `compare_exchange_strong()` применяют, чтобы избежать цикла. Если обмен не получается, становится понятно, что указатель `next` уже установлен другим потоком, поэтому надобность в новом узле, под который вначале была распределена память, отпадает и его можно удалить ❽. Кроме того, для обновления `tail` нужно воспользоваться значением `next`, установленным другим потоком ❾.

Обновление указателя `tail` было выделено в функцию `set_new_tail()` ❶. В ней для обновления `tail` используется цикл `compare_exchange_weak()` ❷, поскольку, если другие потоки пытаются выполнить операцию `push()` и поместить новый узел, то часть, относящаяся к `external_count`, может быть изменена, а потерять ее не хотелось бы. Нужно также позаботиться о том, чтобы не произошло замены значения, если другой поток уже произвел успешное изменение, в противном случае это может закончиться петлями в очереди, чего совершенно не хотелось бы. Следовательно, нужно сделать так, чтобы та часть загруженного значения, которая относится к `ptr`, при неудачном завершении операции сравнения — обмена оставалась прежней. Если после выхода из цикла `ptr` сохранит значение ❸, то значение `tail` должно быть установлено успешно, следовательно, нужно освободить старый внешний счетчик ❹. Если значение `ptr` уже другое, значит, счетчик освобожден другим потоком, следовательно, нужно освободить одну ссылку, удерживаемую этим потоком ❺.

Если потоку, вызывавшему `push()`, не удалось установить указатель `data` при текущем прохождении цикла, он в состоянии помочь завершить обновление более успешному потоку. Сначала предпринимается попытка обновить указатель `next`, нацелив его на новый узел, память под который распределена этим потоком ⓫. Если это удастся сделать, потребуется воспользоваться узлом, под который распределена память, в качестве нового узла `tail` ⓬ и распределить память под еще один новый узел в ожидании возможности помещения элемента в очередь ⓭. Затем можно попытаться установить узел `tail`, вызвав функцию `set_new_tail` перед новым прохождением цикла ⓮.

Можно заметить, что для такого скромного фрагмента кода слишком много `new` и `delete`, а причина в том, что память под новые узлы распределяется в `push()`,

а уничтожаются они в pop(). Поэтому эффективность распределения памяти существенно влияет на производительность этого кода: неудачный распределитель может полностью разрушить свойства масштабируемости подобного свободного от блокировок контейнера. Рассказ о выборе и реализации таких распределителей не вписывается в тему книги, но все же важно понимать, что единственный способ определения качества распределителя заключается в его практическом испытании и замерах производительности кода до и после применения. К общепринятым технологиям оптимизации распределения памяти относятся задействование отдельного распределителя памяти в каждом потоке и использование списка свободных узлов для повторного применения в качестве альтернативы их возвращения распределителю.

Примеров пока вполне достаточно. А теперь рассмотрим ряд рекомендаций по созданию структур данных без блокировок, выработанных на их основе.

7.3. Рекомендации по созданию структур данных без блокировок

Внимательно изучив все примеры, приведенные в данной главе, можно было оценить сложность получения работоспособного кода, свободного от блокировок. Создать собственные структуры данных поможет ряд советов, на которые следует обратить внимание. Основные рекомендации, относящиеся к конкурентным структурам данных (изложены в начале главы 6), по-прежнему в силе, но их недостаточно. На основе примеров мною выведен ряд полезных рекомендаций, к которым можно будет обращаться при разработке структур данных, свободных от блокировок.

7.3.1. Для создания прототипа используйте std::memory_order_seq_cst

Разобраться в порядке доступа к памяти, определяемом семантикой std::memory_order_seq_cst, гораздо проще, чем с любым другим упорядочением, поскольку все операции с такой семантикой выстраиваются в четком порядке. Во всех примерах этой главы упорядочение начиналось с применения std::memory_order_seq_cst, и только потом, по мере проработки основных операций, смягчались ограничения порядка доступа к памяти. В этом смысле задействование других схем упорядочения доступа к памяти является *оптимизацией*, с которой не стоит спешить. В общем, определить, в отношении каких операций можно применить смягчение, возможно, только получив полное представление обо всем коде, допущенном к работе в недрах структуры данных. Попытки нарушить этот порядок только усложнят ситуацию. Она усложняется еще и тем, что код может работать при тестировании, но гарантий полной работоспособности это не дает. Пока не будет средств проверки алгоритма, способных систематически тестировать все возможные комбинации обзора данных

со стороны потоков, согласующиеся с упомянутыми гарантиями упорядочения доступа к памяти (а средства для решения этой задачи имеются), просто выполнить код будет недостаточно.

7.3.2. Воспользуйтесь схемой утилизации памяти, свободной от блокировок

Управление памятью — одна из самых серьезных сложностей при разработке кода, свободного от блокировок. Важно не допустить удаления объектов при наличии ссылок на них из других потоков, но при этом, чтобы избежать чрезмерного потребления памяти, стремиться удалить их как можно скорее. В этой главе были показаны три метода обеспечения безопасности утилизации памяти.

- Дождаться отсутствия обращений к структуре данных от имеющихся потоков и удалить все объекты, ожидающие этой операции.
- Воспользоваться указателями опасности для выявления потока, обращающегося к конкретному объекту.
- Подсчитать ссылки на объекты и не допускать удаления этих объектов, пока есть ссылки.

В любом случае суть замысла заключается в использовании приемов отслеживания количества потоков, обращающихся к конкретному объекту, и удалении любого объекта только тогда, когда на него уже нет ссылок. Существует множество других способов утилизации памяти о структурах данных, свободных от блокировок. Например, идеальным будет сценарий со сборщиком мусора. Гораздо проще создавать алгоритмы, если известно, что сборщик мусора освободит память, занимаемую узлами, не раньше того момента, когда они перестанут применяться.

Еще одной альтернативой является повторное использование узлов и полное освобождение занимаемой ими памяти при уничтожении всей структуры данных. Поскольку узлы применяются повторно, память никогда не выпадает из оборота, следовательно, исчезают и трудности, связанные с устранением неопределенного поведения. Минус в том, что на первый план выходит так называемая *проблема ABA*.

7.3.3. Остерегайтесь проблемы ABA

Остерегаться проблемы ABA приходится при использовании любого алгоритма, основанного на операции сравнения — обмена. Она возникает следующим образом.

1. Поток 1 считывает значение атомарной переменной `x` и обнаруживает, что ее значение равно `A`.
2. Поток 1 выполняет некую операцию, основанную на этом значении, например разыменование (если это указатель) или поиск, либо делает что-то еще.
3. Поток 1 приостанавливается операционной системой.

4. Другой поток выполняет операции над переменной x, и ее значение меняется на B.
5. Затем поток изменяет данные, связанные со значением A, и значение, хранящееся в потоке 1, становится недействительным. Результат может быть весьма радикальным, например освобождение памяти по указателю или изменение связанного значения.
6. Затем на основе новых данных поток снова присваивает x значение A. Если имеется в виду указатель, то это может быть новый объект, который случайным образом использует тот же самый адрес, что и старый.
7. Поток 1 возобновляется и выполняет в отношении x операцию сравнения — обмена со значением A. Операция сравнения — обмена завершается успешно, поскольку значение действительно равно A, но это уже не то значение A. Данные, изначально считанные на шаге 2, уже потеряли достоверность, но сообщить об этом потоку 1 невозможно, и его действия приведут к повреждению структуры данных.

От этой проблемы не страдает ни один из представленных ранее алгоритмов, но вероятность создания свободного от блокировок алгоритма, подверженного этой напасти, весьма высока. Чаще всего избежать данной проблемы удается за счет включения в переменную x счетчика ABA. Тогда операция сравнения — обмена будет выполняться в отношении объединенной структуры x плюс счетчик, представляющей собой единое целое. При каждой замене значения счетчик увеличивается на единицу, следовательно, даже если x имеет то же самое значение, операция сравнения — обмена провалится, если переменную x обновлял другой поток.

Чаще всего проблема ABA встречается в алгоритмах, задействующих списки свободных узлов или иные приемы организации повторного использования узлов без возвращения распределителю занимаемой ими памяти.

7.3.4. Выявляйте циклы активного ожидания и организуйте помощь другому потоку

В последнем примере с созданием очереди было показано, как потоку, добавляющему узел, приходилось ждать завершения такого же добавления другим потоком, прежде чем продолжить выполнение своей операции. Если оставить все как есть, получится цикл активного ожидания и поток будет впустую тратить время центрального процессора, ожидая возможности продолжить операцию. Если смириться с циклом активного ожидания, то, по сути, получится заблокированная операция, для чего можно было использовать мьютексы и блокировки. Изменив алгоритм, чтобы ожидающий поток, которому диспетчер выделил процессорное время, выполнял незавершенные шаги до того, как исходный поток завершит свою операцию, можно избавиться от активного ожидания, и блокировка исчезнет. В примере с очередью это потребует изменения компонента data с превращением его из неатомарной переменной в атомарную, а также применения для установки его значения операций сравнения — обмена, но в более сложных структурах данных могут потребоваться изменения серьезнее.

Резюме

В этой главе мы перешли от структур данных, основанных на использовании блокировок (см. главу 6), к простым реализациям различных структур данных, свободных от блокировок, начав, как и прежде, со стека и закончив очередью. Вы увидели, как важно организовать порядок доступа к памяти в используемых атомарных операциях, чтобы гарантировать отсутствие состояния гонки за данными и единое видение структуры всеми потоками. Вы также убедились, насколько усложняется управление памятью для структур данных, свободных от блокировок, по сравнению со структурами на основе блокировок. Мы исследовали несколько механизмов, позволяющих преодолеть эти сложности. Кроме этого, были продемонстрированы способы избавления от циклов ожидания за счет помощи тому потоку, завершение которого ожидается.

Разработка структур данных, свободных от блокировок, — задача непростая, при ее решении легко наделать ошибок. Но такие структуры данных масштабируемы, что в определенных ситуациях играет весьма важную роль. Надеюсь, что предстоящий анализ примеров из этой главы и изучение рекомендаций позволят вам лучше разобраться в структурах данных, свободных от блокировок. Вы без труда сможете реализовать структуру, следуя описаниям в научных статьях, или обнаружить ошибку в структуре, разработанной бывшим коллегой до того, как он ушел из компании.

Когда данные совместно используются несколькими потоками, необходимо как следует продумывать структуры данных и способы синхронизации доступа к данным между потоками. Разрабатывая конкурентные структуры данных, можно заложить соответствующие функции в сами структуры, чтобы остальной код был сконцентрирован на решении поставленной задачи в отношении данных, а не на синхронизации доступа к ним. Практическое воплощение такого подхода будет показано в главе 8 по мере перехода от конкурентных структур данных к конкурентному коду в целом. В параллельных алгоритмах для повышения их производительности используется сразу несколько потоков, и выбор конкурентной структуры данных играет решающую роль.

8 Разработка конкурентного кода

В этой главе

- Способы распределения данных между потоками.
- Факторы, влияющие на производительность конкурентного кода.
- Влияние факторов производительности на дизайн структуры данных.
- Безопасность исключений в многопоточном коде.
- Масштабируемость.
- Примеры реализации параллельных алгоритмов.

В предыдущих главах основное внимание уделялось инструментальным средствам для создания конкурентного кода, имеющимся в арсенале C++. В главах 6 и 7 рассматривались способы применения этих средств для разработки простых конкурентных структур данных, допускающих безопасный конкурентный доступ со стороны нескольких потоков. Но как столяру для создания шкафа или стола недостаточно знать, как создаются шарниры или соединения, так и для разработки конкурентного кода недостаточно уметь разрабатывать и использовать простые структуры данных. Теперь нужно рассматривать проблему в более широком контексте, чтобы создавать крупные структуры, способные выполнять полезную работу. В качестве примеров возьмем многопоточные реализации некоторых алгоритмов стандартной библиотеки C++, но те же принципы применимы и при разработке приложений разного уровня.

Как и в любом программном проекте, здесь первостепенную роль играет тщательное продумывание структуры конкурентного кода. Но, работая с многопоточным кодом, приходится учитывать еще больше факторов, чем при написании последовательного. Нужно принимать во внимание не только инкапсуляцию, связанность и сцепленность (эти понятия подробно рассматриваются во многих книгах по проектированию программного обеспечения), но и то, какие данные разделять, как организовывать доступ к ним, каким потокам придется ждать других потоков для завершения операции и т. д.

В этой главе будут рассмотрены все эти вопросы, начиная с высокоуровневых (но фундаментальных) соображений о количестве применяемых потоков, о том, какой код выполнять в каждом потоке и как многопоточность влияет на понятность программы, и заканчивая низкоуровневыми подробностями того, как организовать разделяемые данные для достижения оптимальной производительности.

Начнем с рассмотрения способов распределения работы между потоками.

8.1. Способы распределения работы между потоками

Представьте, что вам нужно построить дом. Для этого придется вырыть котлован под фундамент, залить сам фундамент, возвести стены, проложить трубы и электропроводку и т. д. Теоретически при достаточных навыках все можно сделать самостоятельно, но, скорее всего, на это уйдет много времени и придется переключаться с одной работы на другую. Но можно нанять помощников. Тогда нужно будет выбирать, сколько помощников нанимать, и решать, что они должны уметь. Можно, к примеру, нанять двух разнорабочих и трудиться вместе с ними. Тогда по-прежнему придется переключаться с одной работы на другую, но теперь дела пойдут быстрее, так как исполнителей станет больше.

Можно выбрать другой вариант — нанять бригаду специалистов, например каменщика, плотника, электрика и водопроводчика. Каждый будет работать по своей специальности, следовательно, пока у водопроводчика не появится фронт работ, он будет сидеть без дела. И все же дела пойдут быстрее, чем прежде, поскольку работников стало больше, и, пока электрик будет вести проводку в кухне, водопроводчик может заняться санузлом. Но когда нет работы для конкретного специалиста, простоев получается больше. Однако можно заметить, что даже с учетом простоев работа движется скорее, когда за дело берутся специалисты, а не команда разнорабочих. Специалистам не нужно постоянно менять инструменты, и наверняка каждый из них будет выполнять свою задачу быстрее разнорабочего. Будет ли так на самом деле, зависит от конкретных обстоятельств: все познается на практике.

Даже если задействовать специалистов, нужно все же выбирать разное количество работников различных специальностей. Возможно, есть смысл нанять, к примеру, больше каменщиков, чем электриков. К тому же состав вашей команды и общая эффективность ее работы могут меняться, если придется строить сразу несколько домов. Даже если для водопроводчика мало работы в отдельно взятом

доме, то при строительстве сразу нескольких домов его можно занять на весь день. Притом, если вы не должны платить специалистам за простой, можно набрать команду побольше, даже при условии, что количество одновременно работающих людей не изменится.

Но хватит рассуждать о строительстве. Какое все это имеет отношение к потокам? А к ним можно применить аналогичные соображения. Следует решить, сколько потоков задействовать и какие задачи они должны выполнять. Нужны ли универсальные потоки, делающие ту работу, которая необходима в конкретный момент, или потоки-специалисты, хорошо приспособленные к чему-то одному? А может стоит сочетать те и другие? Эти решения необходимо принимать вне зависимости от причин распараллеливания программы, и от того, насколько они будут удачны, значительно зависит производительность и ясность кода. Поэтому так важно представлять, какие имеются варианты, чтобы при разработке структуры приложения принять грамотное решение. В этом разделе будет рассмотрен ряд приемов распределения задач, начиная с распределения данных между потоками до выполнения любой другой работы.

8.1.1. Распределение данных между потоками до начала обработки

Проще всего поддаются распараллеливанию простые алгоритмы, например `std::for_each`, выполняющие операции над каждым элементом набора данных. Чтобы распараллелить этот алгоритм, можно назначить каждый элемент одному из обрабатывающих потоков. В дальнейшем, при рассмотрении вопросов производительности, станет понятно, что наилучший вариант распределения для достижения оптимальной производительности зависит от особенностей структуры данных.

При распределении данных простейшим считается вариант, когда первые N элементов назначаются одному потоку, следующие N элементов — другому и т. д. (рис. 8.1), но можно использовать и другие схемы. Независимо от способа распределения данных, каждый поток обрабатывает только назначенные ему элементы, никак не взаимодействуя с другими потоками до тех пор, пока не завершит обработку.

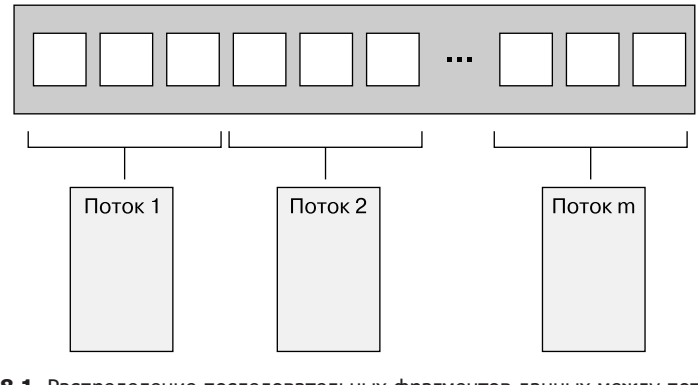


Рис. 8.1. Распределение последовательных фрагментов данных между потоками

Структура должна быть знакома каждому, кто имел дело с программированием в среде Message Passing Interface (MPI, http://www.mpi-forum.org/) или OpenMP (http://www.openmp.org/): задача разбивается на множество параллельно выполняемых задач, рабочие потоки запускают их независимо друг от друга, а результаты собираются на финальной стадии *сведения*. Такой подход использовался в примере с функцией `accumulate` из раздела 2.4: обе параллельные задачи и этап сведения представляют собой аккумулирование. Для простого алгоритма `for_each` финальный этап отсутствует, так как нечего сводить.

То, что в качестве сути финального этапа определено сведение, играет весьма важную роль: элементарная реализация, подобная показанной в листинге 2.9, выполнит это сведение в качестве итогового последовательного этапа. Но зачастую этот этап также распараллеливается: аккумулирование является операцией сведения, поэтому код листинга 2.9 можно изменить, чтобы получить рекурсивный вызов этого же кода, когда, к примеру, количество потоков больше минимального количества обрабатываемых потоком элементов. Также можно заставлять рабочие потоки выполнять этапы сведения, как только каждый из них завершит свою задачу, а не запускать всякий раз новые потоки.

При всей своей эффективности этот прием не отличается универсальностью. Иногда данные не удается аккуратно разделить заранее, поскольку состав каждой части становится известен только при обработке. В частности, это оказывается очевидным при использовании рекурсивных алгоритмов, например Quicksort, поэтому для них требуется иной подход.

8.1.2. Рекурсивное распределение данных

У алгоритма Quicksort два основных этапа: разбиение данных на две части — все, что до одного из элементов (опорного), и все, что после него в окончательном порядке сортировки, и рекурсивная сортировка этих двух половин. Это невозможно распараллелить предварительным разбиением данных, так как определить, в какую «половину» они попадут, можно только в ходе обработки элементов. Намереваясь распараллелить этот алгоритм, нужно воспользоваться самой сутью рекурсии. На каждом уровне рекурсии выполняется *все больше* вызовов функции `quick_sort`, так как приходится сортировать как те элементы, что больше опорного, так и те, что меньше его. Эти рекурсивные вызовы независимы друг от друга, поскольку обращаются к отдельным наборам элементов. Из-за этого они первые кандидаты на конкурентность. Данное рекурсивное распределение показано на рис. 8.2.

Эта реализация уже встречалась в главе 4. Вместо выполнения двух рекурсивных вызовов для большей и меньшей половин использовалась функция `std::async()`, запускающая на каждом шаге асинхронные задачи для меньшей половины. Из-за применения `std::async()` C++ Thread Library должна была решать, когда запускать задачу в новом потоке, а когда — в синхронном режиме.

Есть одно важное обстоятельство: при сортировке большого набора данных запуск нового потока для каждой рекурсии приведет к быстрому росту числа потоков. При изучении вопросов производительности будет показано, что слишком большое

количество потоков может замедлить работу приложения. Кроме того, при крупном наборе данных потоков может просто не хватить. Сам по себе замысел разбиения всей задачи в подобном рекурсивном режиме представляется весьма удачным, нужно только тщательнее отслеживать число потоков. В простых случаях с этим справляется функция `std::async()`, но есть и другие варианты.

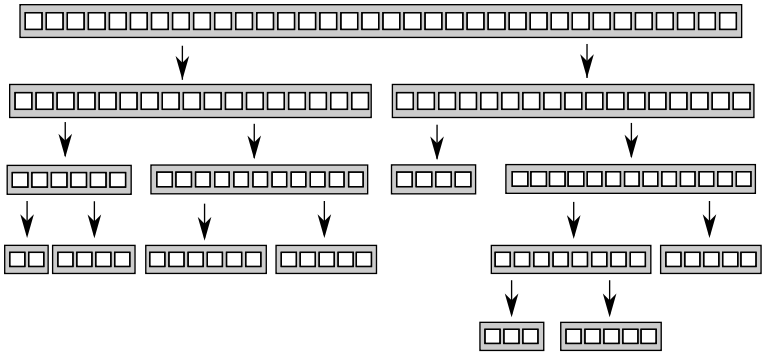


Рис. 8.2. Рекурсивное распределение данных

Один из них заключается в применении для выбора количества потоков функции `std::thread::hardware_concurrency()`, как это делалось в параллельной версии функции `accumulate()` из листинга 2.9. Тогда вместо запуска нового потока для каждого рекурсивного вызова можно помещать подлежащий сортировке фрагмент в потокобезопасный стек, например такой, как рассматривался в главах 6 и 7. Если потоку больше нечем заняться либо он завершил обработку всех своих фрагментов или находится в ожидании сортируемого фрагмента, он может взять фрагмент из стека и отсортировать его.

В листинге 8.1 показана простая реализация этой технологии. Как и в большинстве других примеров, она лишь демонстрирует замысел, а не является кодом, готовым к практическому применению. Если используется компилятор C++17 и ваша библиотека его поддерживает, стоит воспользоваться параллельными алгоритмами, предоставляемыми стандартной библиотекой в соответствии с описаниями, приведенными в главе 10.

Листинг 8.1. Параллельный алгоритм Quicksort, в котором используется стек фрагментов, 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)
(3) --> std::vector<std::thread> threads;
```

```
    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()  ←❹
    {
❺→      end_of_data=true;
        for(unsigned i=0;i<threads.size();++i)
        {
            threads[i].join();  ←❻
        }
    }
    void try_sort_chunk()
    {
        boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();  ←❼
        if(chunk)
        {
            sort_chunk(chunk);    ←❽
        }
    }
    std::list<T> do_sort(std::list<T>& chunk_data)  ←❾
    {
        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();
❿→      typename std::list<T>::iterator divide_point=
            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));    ←⓫
⓬→      if(threads.size()<max_thread_count)
        {
            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)) !=
⓭→          std::future_status::ready)
        {
            try_sort_chunk();                              ←⓮
        }
        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));   ←⓯
    }
    void sort_thread()
    {
        while(!end_of_data)      ←⓰
        {
    ⓱→      try_sort_chunk();
            std::this_thread::yield();   ←⓲
        }
    }
};
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)   ←⓳
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);  ←⓴
}
```

Здесь функция `parallel_quick_sort` ⓳ возлагает основную часть функциональных обязанностей на класс `sorter` ❶, предоставляющий простой способ группировки стека неотсортированных фрагментов ❷ и множества потоков ❸. Основная работа выполняется в компонентной функции `do_sort` ❾, занятой обычным разбиением данных ❿. На этот раз вместо запуска нового потока для каждого фрагмента она помещает этот фрагмент в стек ⓫ и запускает новый поток только при наличии свободного процессорного ресурса ⓬. Поскольку фрагмент со значениями меньшими, чем у опорного, может обрабатываться другим потоком, следовало бы дождаться его готовности ⓭. Чтобы время не растрачивалось зря (в том случае, если мы располагаем единственным потоком или все остальные потоки уже заняты), на период ожидания в этом потоке предпринимается попытка обработки фрагментов из стека ⓮. Функция `try_sort_chunk` извлекает фрагмент из стека ❼, сортирует его ❽ и сохраняет результаты в промисе `promise`, чтобы их смог получить поток, который поместил этот фрагмент в стек ⓯.

Теперь только что запущенные потоки находятся в цикле и пытаются сортировать фрагменты из стека ⓱, если не установлен флаг `end_of_data` ⓰. Между проверками они уступают вычислительный ресурс другим потокам ⓲, чтобы те могли поместить в стек дополнительную работу. Работа кода в плане приведения в порядок этих потоков зависит от деструктора вашего класса `sorter` ❹. Когда отсортированы все данные, функция `do_sort` вернет управление (даже при сохранении активности рабочих потоков), основной поток возвратится из `parallel_quick_sort` ⓴ и уничтожит объект `sorter`. Он установит флаг `end_of_data` ❺ и станет ждать завершения работы потоков ❻. Установка флага приведет к остановке цикла в функции потоков ⓰.

С таким подходом исчезнет проблема неограниченного числа потоков, присущая функции `spawn_task`, запускавшей новый поток, и пропадет зависимость от библиотеки потоков C++, выбиравшей за вас число потоков, как она делает при использовании `std::async()`. Вместо этого, чтобы не допустить слишком частого переключения задач, количество потоков ограничивается значением, возвращаемым функцией `std::thread::hardware_concurrency()`. Но возникает другая проблема: управление этими потоками и обмен данными между ними сильно усложняют код. Вдобавок, несмотря на то что потоки обрабатывают отдельные элементы данных, все они обращаются к стеку, добавляя в него новые фрагменты и забирая фрагменты для обработки. Такая острая конкуренция может снизить производительность, даже если используется свободный от блокировок (следовательно, неблокируемый) стек, и причины этого вскоре будут рассмотрены.

Данный подход является особой версией *пула потоков* — набора потоков, каждый из которых получает работу из списка отложенных работ, выполняет ее, а затем обращается к списку за новой. Некоторые потенциальные проблемы, присущие пулу потоков (включая конкуренцию при обращении к списку работ), и пути их решения рассматриваются в главе 9. О масштабировании создаваемого приложения таким образом, чтобы оно выполнялось на нескольких процессорах, более подробно поговорим в этой главе чуть позже (см. подраздел 8.2.1).

При распределении данных как до обработки, так и в рекурсивном режиме предполагается, что они заранее зафиксированы, и ведется поиск способов их распределения. Но так бывает не всегда: если данные создаются в динамическом режиме или поступают от внешнего источника, этот подход не работает. В таком случае, возможно, разумнее распределить работу по типам задач, а не на основе самих данных.

8.1.3. Распределение работы по типам задач

Распределение работы между потоками путем назначения каждому из них (заранее или рекурсивно в ходе обработки данных) различных фрагментов данных в любом случае основывается на предположении, что потоки собираются выполнять над каждым фрагментом одну и ту же работу. Альтернативный вариант распределения работы заключается в специализации потоков, где каждый выполняет отдельную задачу, как водопроводчики и электрики выполняют разные задачи при строительстве дома. Потоки могут работать с разными или одними и теми же данными, но в последнем случае они делают это с разными целями.

Это своеобразное разделение труда возникает в результате разделения решаемых задач с помощью конкурентности: у каждого потока имеется отдельная задача, которую он выполняет независимо от других потоков. Иногда другие потоки могут поставлять потоку данные или выдавать события, на которые он должен реагировать, но в целом каждый поток концентрируется на качественном выполнении какой-то одной задачи. Это неплохая базовая конструкция, где каждый фрагмент кода должен отвечать за что-то одно.

Распределение работы по типам задач с целью разделения ответственности

Однопоточному приложению приходится справляться с конфликтами, связанными с принципом единой ответственности, когда имеется несколько задач, которые должны выполняться непрерывно в течение определенного времени, или приложение должно своевременно справляться с обработкой поступающих событий (к примеру, пользователь нажимает клавишу или по сети приходят данные) даже при наличии других незавершенных задач. В среде однопоточных вычислений приходится самостоятельно создавать код, который выполняет часть задачи A, часть задачи B, проверяет, не нажата ли клавиша и нет ли поступивших сетевых пакетов, а затем циклично возвращается к выполнению следующей части задачи A. Это приводит к усложнению кода для выполнения задачи A из-за необходимости сохранения его состояния и периодического возвращения управления основному циклу. Если в цикл добавить слишком много задач, работа может существенно замедлиться и пользователь, вероятно, заметит замедленную реакцию на нажатие клавиш. Уверен, что крайние проявления подобной ситуации в тех или иных приложениях наблюдал каждый: вы ставите перед приложением какую-то задачу, и интерфейс ни на что не реагирует, пока она не завершится.

Тут на сцену выходят потоки. Если каждую задачу запускать в отдельном потоке, то этим вместо вас может заняться операционная система. В коде для задачи A можно сосредоточиться на выполнении задачи, не беспокоясь о сохранении состояния и возвращении в основной цикл или о том, сколько времени пройдет, прежде чем это случится. То есть операционная система автоматически сохранит состояние и в нужный момент переключится на задачу B или C, и, если система, на которой будет выполняться программа, обладает несколькими ядрами или процессорами, появится возможность одновременно выполнять задачи A и B. Код обработки нажатий клавиш или поступления сетевых пакетов теперь может выполняться своевременно, и в выигрыше останутся все: пользователь получит адекватную реакцию программы, а вы, как разработчик, получите упрощенный код, поскольку каждый поток можно будет направить на выполнение операций, имеющих прямое отношение к его обязанностям, не смешивая их с потоком управления и взаимодействием с пользователем.

Вырисовывается какая-то идеальная картина. Но может ли все складываться именно таким образом? Как всегда, все зависит от конкретных обстоятельств. Если соблюдается полная независимость и потокам не нужно обмениваться друг с другом данными, то именно так все и произойдет. К сожалению, подобная ситуация наблюдается довольно редко. Зачастую необходимые пользователю действия имеют вид удобных фоновых задач, и им нужно оповестить пользователя о выполнении задачи, обновив каким-то образом пользовательский интерфейс. Или же пользователю может потребоваться прекратить выполнение задачи, поэтому пользовательскому интерфейсу нужно будет каким-то образом отправить сообщение фоновой задаче, заставив ее прекратить выполнение. В обоих случаях требуется тщательно продумать конструкцию и надлежащую синхронизацию, но выполняемые задачи так и останутся разобщенными. Поток пользовательского интерфейса по-прежнему

управляет этим интерфейсом, но на него может возлагаться выполнение обновления по запросу других потоков. Поток, реализующий фоновую задачу, по-прежнему концентрируется на операциях, требующихся для ее выполнения, бывает и так, что один из фоновых потоков допускает остановку задачи другим потоком. В обоих случаях потокам все равно, откуда поступает запрос, им важно лишь то, что он предназначен для них и напрямую связан с их обязанностями.

При распределении ответственности между несколькими потоками возникают две серьезные опасности. Во-первых, может оказаться, что распределены *ненадлежащие* обязанности. Признаком этого является слишком большой объем данных, совместно используемых потоками, или же то, что разным потокам приходится дожидаться друг друга. В обоих случаях наблюдается слишком интенсивный обмен данными между потоками. Нужно разобраться с причинами такого обмена. Если он вызван одной и той же причиной, возможно, все, чем эти потоки занимаются, должно стать основной обязанностью одного потока и следует освободить все занятые прежде потоки. В том случае, когда между двумя потоками ведется интенсивный обмен данными, а между другими потоками — гораздо менее напряженный, возможно, эти два потока должны быть сведены в один.

При разделении работы между потоками на основании типа задач не следует ограничиваться только полностью изолированными вариантами. Если несколько наборов входящих данных требуют одной и той же *последовательности* применяемых операций, работу можно распределить таким образом, чтобы каждый поток выполнял один этап из общей последовательности.

Распределение последовательности задач между потоками

Если суть задачи заключается в применении одной и той же последовательности операций ко многим независимым элементам данных, то для использования доступных в вашей системе средств конкурентности можно задействовать *конвейер*. Здесь прослеживается аналогия с настоящим конвейером: данные, поступив на один его конец, проходят через серию операций и выходят с другого конца.

При таком распределении работы отдельный поток создается для каждого конвейерного этапа — по одному потоку для каждой операции в последовательности. Когда операция завершена, элемент данных помещается в очередь, из которой его забирает следующий поток. Это позволяет потоку, выполняющему первую операцию в последовательности, приступить к работе со следующим элементом данных, в то время как второй поток конвейера работает с первым элементом.

Альтернативный вариант распределения данных между потоками, рассмотренный в подразделе 8.1.1, подходит для случаев, когда к началу операции о самих входных данных ничего не известно. Например, данные могут поступить по сети или первой операцией последовательности может быть сканирование файловой системы для идентификации обрабатываемых файлов.

Конвейеры хорошо справляются с ситуациями, когда каждая операция в последовательности требует много времени: распределение между потоками задач, а не данных позволяет изменить профиль производительности. Предположим, что для обработки на четырех ядрах имеется 20 элементов данных и каждому элементу

данных требуется четыре этапа обработки, на каждый из которых уходит 3 с. Если распределить данные по четырем потокам, то у каждого из них будет пять обрабатываемых элементов. Если предположить, что других обработок, способных повлиять на хронометраж, не имеется, то через 12 с у вас будет четыре обработанных элемента, через 24 с — восемь и т. д. Все 20 элементов будут обработаны через минуту. А с конвейером все происходит иначе. Каждый из имеющихся четырех этапов можно назначить ядру обработки. Теперь первый элемент должен обрабатываться каждым ядром, следовательно, на это, как и прежде, уйдет целых 12 с. Получается, что через 12 с будет обработан всего лишь один элемент, что отстает по показателям от варианта распределения данных. Но как только конвейер будет *заполнен*, ситуация изменится: первое ядро, обработав первый элемент, переходит к обработке второго элемента, таким образом, как только последнее ядро обработало первый элемент, оно может выполнить свой этап обработки второго элемента. Теперь каждый элемент обрабатывается каждые 3 с вместо обработки элементов пакетами по четыре каждые 12 с.

Общее время обработки всего пакета увеличится, поскольку должно пройти 9 с, прежде чем последнее ядро приступит к обработке первого элемента. Но при определенных обстоятельствах может оказаться предпочтительнее более плавная и размеренная обработка. Рассмотрим, к примеру, систему просмотра цифрового видео высокого разрешения. Для комфортного просмотра видео обычно нужно демонстрировать не менее 25 кадров в секунду, а в идеале — гораздо больше. Кроме того, чтобы у пользователя возникала иллюзия непрерывного движения, все должно происходить равномерно: приложение, способное декодировать 100 кадров в секунду, будет бесполезным, если продержит паузу 1 с, после чего покажет 100 кадров, остановится еще на 1 с и покажет следующие 100 кадров. В то же время зрители, скорее всего, будут не против двухсекундной паузы перед просмотром видео. В таком случае распараллеливание с использованием конвейера, выдающего кадры в постоянном темпе, конечно же, предпочтительнее.

Рассмотрев различные способы распределения работы между потоками, давайте поговорим о факторах, влияющих на производительность многопоточной системы, и о том, как их учитывать при выборе способов.

8.2. Факторы, влияющие на производительность конкурентного кода

Если конкурентность применяется в системах с несколькими процессорами для повышения производительности кода, следует знать, какие факторы могут повлиять на производительность. Даже если несколько потоков используются для распределения ответственности, нужно убедиться, что это не ухудшит производительность. Потребители не скажут вам спасибо, если на их великолепной машине с 16 ядрами ваше приложение будет работать медленнее, чем на старом одноядерном компьютере.

Как вскоре будет показано, на производительность многопоточного кода влияет множество факторов — даже такие простые, как перестановка элементов данных, об-

рабатываемых каждым потоком (притом что все остальное остается без изменений), могут вылиться в существенное снижение производительности. А теперь без пространных предисловий рассмотрим некоторые из этих факторов, начиная с наиболее очевидного: сколько процессоров имеется на целевой системе?

8.2.1. А сколько у нас процессоров?

Количество (и структура) процессоров является первым существенным фактором, влияющим на производительность многопоточного приложения и играющим весьма важную роль. Иногда точно известно, каким будет целевое оборудование, и разработку можно вести с учетом этих сведений, получая реальные измерения на целевой системе или ее точной копии. В таком случае вам повезло, но обычно удача проходит мимо. Разработка может вестись на подобной системе, но отличия могут быть существенными. Например, программа разрабатывается на системе с двумя или четырьмя ядрами, но системы потребителей могут иметь один многоядерный процессор (с любым количеством ядер) либо несколько одноядерных или даже многоядерных процессоров. Характеристики поведения и производительности программы, рассчитанной на конкурентность, при различных обстоятельствах способны существенно варьироваться, поэтому нужно серьезно задуматься о возможном влиянии и по возможности все протестировать.

В первом приближении один 16-ядерный процессор аналогичен четырем четырехъядерным процессорам или 16 одноядерным процессорам: в любом случае в системе одновременно могут быть запущены 16 потоков. Если нужно этим воспользоваться, у вашего приложения должно иметься не менее 16 потоков. Когда их меньше, вычислительная мощность останется невостребованной, если только система не запустит одновременно другое приложение (сейчас проигнорируем эту возможность). В то же время, если имеется свыше 16 готовых к работе потоков, не заблокированных в ожидании развития событий, ваше приложение будет впустую тратить процессорное время на переключение между ними, о чем говорилось в главе 1. Подобная ситуация называется *переоценкой вычислительных возможностей* (oversubscription).

Чтобы приложение могло подгонять количество действующих потоков под число потоков, доступных для одновременного запуска на имеющемся оборудовании, стандартная библиотека потоков C++11 (Standard Thread Library) предоставляет функцию `std::thread::hardware_concurrency()`. Здесь уже демонстрировалась возможность задействовать ее для масштабирования количества потоков под возможности оборудования.

Использование функции `std::thread::hardware_concurrency()` непосредственным образом требует осторожности: ваш код не принимают в расчет какие-либо другие потоки, запущенные в системе, пока вы не поделитесь этой информацией явным образом. При наихудшем сценарии, если несколько потоков одновременно вызывают для масштабирования функцию, применяющую `std::thread::hardware_concurrency()`, может возникнуть переоценка вычислительных возможностей. Функции `std::async()` удается избежать этой проблемы, поскольку библиотека

осведомлена обо всех вызовах и может справиться с диспетчеризацией. От проблемы можно избавиться также за счет грамотного использования пула потоков.

Но если даже учесть все потоки, запущенные в вашем приложении, останется влияние других приложений, действующих в то же самое время. Хотя одновременное применение на однопользовательских системах нескольких интенсивно потребляющих ресурс центрального процессора приложений — случай довольно редкий, есть ряд областей, где такое случается чаще обычного. Системы, разработанные с учетом такого сценария, обычно предлагают механизмы, позволяющие каждому приложению выбрать подходящее число потоков, хотя такие механизмы не вписываются в область действия стандарта C++. Один из вариантов, предназначенных для функции наподобие `std::async()`, заключается в учете общего количества асинхронных задач, запущенных всеми приложениями, при выборе числа потоков. Другой вариант предусматривает ограничение количества вычислительных ядер, которые может задействовать конкретное приложение. Следовало бы ожидать, что это ограничение отразится в значении, возвращаемом на таких платформах функцией `std::thread::hardware_concurrency()`, но подобный исход не гарантирован. Если нужно справиться с таким сценарием, обратитесь к документации по вашей системе, чтобы выяснить, доступны ли эти варианты.

Еще один нюанс ситуации заключается в том, что идеальный алгоритм решения данной проблемы может зависеть от ее масштаба в сравнении с количеством вычислительных единиц. При наличии мощной параллельной системы со множеством вычислительных единиц алгоритм, выполняющий в целом больше операций, может справиться с задачей быстрее того, который реализует меньше операций, поскольку каждый процессор выполняет всего лишь несколько операций.

По мере роста числа процессоров растет и вероятность того, что на производительность повлияет еще одна проблема — попытка нескольких процессоров обратиться к одним и тем же данным.

8.2.2. Конкуренция при обращении к данным и перебрасывание данных в кэш-памяти процессоров

Если два потока выполняются одновременно на разных процессорах и *читают* одни и те же данные, то обычно это обходится без проблем: данные будут скопированы в соответствующий им кэш и оба процесса смогут продолжить работу. Но если один из потоков *изменяет* данные, это изменение должно распространиться на кэш другого ядра, на что нужно время. В зависимости от характера операций, выполняемых двумя потоками, и упорядочения доступа к памяти, используемой для операций, это изменение может вызвать остановку второго процессора для того, чтобы отследить изменения и подождать, пока они распространятся по физической памяти. В понятиях инструкций центрального процессора это может быть *крайне* медленная операция, эквивалентная многим сотням отдельных инструкций, хотя

точный хронометраж зависит в первую очередь от физической структуры оборудования.

Рассмотрим 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 любые потоки, вызывающие функцию `processing_loop()`, изменяют одну и ту же переменную. Следовательно, для каждого увеличения ее значения на единицу процессор должен убедиться в актуальности копии `counter` в своем кэше, изменить значение и опубликовать его для других процессоров. Даже притом, что используется семантика `std::memory_order_relaxed`, не вынуждающая компилятор заниматься синхронизацией доступа к каким-либо другим данным, `fetch_add` является операцией чтения — изменения — записи и поэтому нуждается в извлечении самого последнего значения переменной. Если другой поток с другого процессора выполняет тот же самый код, данные для `counter` должны передаваться между двумя процессорами и соответствующей им кэш-памятью, чтобы у каждого процессора, когда он увеличивает счетчик на единицу, было самое последнее значение `counter`. Если функция `do_something()` не очень длинная или этот код запущен на слишком большом количестве процессоров, последние могут войти в режим ожидания друг друга: один процессор готов обновить значение, но другой сейчас занят тем же самым и приходится ждать, пока второй процессор завершит обновление и изменение распространится. Такая ситуация называется *высокой конкуренцией*. Если же процессорам редко приходится дожидаться друг друга, складывается *низкая конкуренция*.

В подобном цикле данные для счетчика будут передаваться между кэшами множество раз. Это называется *перебрасыванием данных в кэш-памяти процессоров*, такая ситуация может существенно повлиять на производительность приложения. Если процессор застопоривается на вынужденном ожидании переброски данных из кэша в кэш, он в это время не может заниматься ничем иным, даже если есть другие ожидающие потоки, способные выполнять полезную работу, что не сулит ничего хорошего всему приложению.

Возможно, вы думаете, что вас это обойдет стороной, ведь у вас нет подобных циклов. Вы уверены? А как насчет блокировок мьютексов? Если мьютекс блокируется в цикле, то с позиции обращения к данным ваш код становится похож на предыдущий фрагмент. Чтобы заблокировать мьютекс, другой поток должен перенести данные, составляющие мьютекс, в свой процессор и изменить их. Как только это будет сделано, он снова изменит мьютекс, чтобы разблокировать его, и данные мьютекса должны быть перенесены к следующему потоку, чтобы тот мог его заблокировать.

Время, затраченное на перенос, будет прибавлено к времени вынужденного ожидания другим потоком момента освобождения мьютекса первым потоком:

```
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 неприятное: если к данным и к мьютексу обращается более одного потока, то по мере добавления к системе ядер и процессоров возрастает вероятность возникновения высокой конкуренции и одному процессору придется дожидаться другого. Если для ускорения обработки одних и тех же данных используются несколько потоков, они конкурируют за доступ к данным, а следовательно, и за блокировку одного и того же мьютекса. Чем их больше, тем более вероятны попытки одновременного блокирования одного и того же мьютекса или одновременного обращения к атомарной переменной и т. д.

Последствия конкуренции за мьютексы обычно отличаются от последствий конкуренции при проведении атомарных операций по той простой причине, что использование мьютексов приводит к естественному упорядочению потоков на уровне операционной системы, а не процессора. Если достаточное количество потоков готово к запуску, операционная система может спланировать запуск другого потока, пока какой-то поток ожидает освобождения мьютекса, а то, что процессор застопорился, не дает запуститься на нем никаким другим потокам. Но это все равно повлияет на производительность тех потоков, которые конкурируют за блокировку мьютекса, так как они могут работать только по отдельности.

В главе 3 было показано, как редко обновляемая структура данных может быть защищена мьютексом, допускающим запись только из одного потока, а чтение — из нескольких потоков (см. подраздел 3.3.2). Эффект перебрасывания данных в кэш-памяти процессоров может свести на нет преимущества от использования этих мьютексов при неблагоприятной рабочей нагрузке, поскольку всем потокам, обращающимся к данным (даже выполняющим чтение), все же придется изменять сам мьютекс. По мере роста числа процессоров, обращающихся к данным, конкуренция за мьютекс возрастет и строка кэша, хранящая его, должна будет передаваться между ядрами, создавая предпосылки для неприемлемого увеличения времени на установку и снятие блокировок. Существуют приемы, позволяющие исправить ситуацию за счет распространения мьютекса среди нескольких строк кэша, но пока не будет реализован собственный мьютекс, придется довольствоваться тем, что предлагает система.

Если перебрасывание данных в кэш-памяти процессоров ни к чему хорошему не приводит, нельзя ли обойтись без него? Далее будет показано, что ответ согласуется с основными рекомендациями по раскрытию потенциала конкурентности:

делайте все возможное для снижения вероятности конкуренции двух потоков за доступ к одному и тому же участку в памяти.

Но, как обычно, все совсем непросто. Даже если к конкретному месту в памяти всегда обращается только один поток, все равно может возникнуть перебрасывание данных в кэш-памяти, причиной которого становится так называемый эффект *ложного совместного использования памяти*.

8.2.3. Ложное совместное использование памяти

Кэш-память процессоров обычно не работает с отдельными участками памяти — она манипулирует блоками памяти, называемыми *строками кэша*. Чаще всего это блоки памяти 32 или 64 байта, но этот параметр зависит от модели используемого процессора. Поскольку аппаратура кэш-памяти работает только с блоками памяти размером со строку кэша, небольшие элементы данных на смежных участках памяти будут находиться в одной строке кэша. Иногда это дает положительный эффект: если набор данных, к которому обращается поток, находится в одной и той же строке кэша, это повышает производительность приложения по сравнению с тем, когда тот же набор данных разбросан по нескольким строкам кэша. Но если элементы данных в строке кэша не имеют друг к другу никакого отношения и обращение к ним должно исходить из разных потоков, это может стать основной причиной проблем с производительностью.

Предположим, что имеется массив целочисленных значений типа `int` и набор потоков, каждый из которых обращается к собственному элементу массива и делает это многократно, иногда обновляя этот элемент. Поскольку обычно объекты типа `int` намного меньше строки кэша, в одной строке может оказаться сразу несколько таких элементов массива. Следовательно, даже если каждый поток обращается только к собственному элементу массива, аппаратуре кэш-памяти все равно приходится перебрасывать данные кэша. Всякий раз, когда потоку, обратившемуся к элементу с индексом 0, нужно обновить его значение, права владения строкой кэша должны быть переданы процессору, на котором выполняется этот поток, только для того, чтобы она была передана кэшу для процессора, выполняющего поток, работающий с элементом с индексом 1, когда этому потоку понадобится обновить свой элемент данных. Получается, что строка кэша применяется совместно, хотя к самим данным это никак не относится, откуда и возникает понятие *ложного совместного использования*. Проблема решается за счет структурирования данных таким образом, чтобы элементы данных, к которым обращается один и тот же поток, находились в памяти как можно ближе друг к другу, тогда они, скорее всего, окажутся в одной строке кэша, а те элементы данных, к которым обращаются разные потоки, отстояли в памяти далеко друг от друга, чтобы с большей вероятностью попасть в разные строки кэша. Как это может повлиять на конструкцию кода и данных, будет показано чуть позже. В стандарте C++17 в заголовке `<new>` определяется константа `std::hardware_destructive_interference_size`, в которой указывается максимальное число последовательных байтов, которые могут подвергнуться ложному совместному использованию для текущего целевого объекта компиляции. Если разнести ваши

данные как минимум на это количество байтов, то никакого ложного совместного использования не будет.

Если в обращении нескольких потоков к данным из одной и той же строки кэша нет ничего хорошего, то как повлияет на ситуацию расположение в памяти данных, к которым обращается один и тот же поток?

8.2.4. Насколько близко расположены по отношению друг к другу ваши данные?

Хотя ложное совместное использование памяти возникает из-за того, что данные, к которым обращаются разные потоки, находятся слишком близко друг от друга, есть еще одна неприятность, связанная с размещением данных в памяти и непосредственно влияющая на производительность отдельно взятого потока. Если данные, к которым обращается один и тот же поток, разбросаны по памяти, высока вероятность их попадания в разные строки кэша. В то же время, если они находятся в памяти близко друг к другу, высока вероятность их попадания в одну и ту же строку кэша. Следовательно, если данные рассредоточены, из памяти в кэш процессора должно быть загружено больше строк кэша, что может увеличить задержку доступа к памяти и снизить производительность по сравнению с той ситуацией, когда данные располагаются близко друг к другу.

Кроме того, если данные разбросаны, возрастает вероятность того, что в строке кэша, содержащей данные для текущего потока, окажутся данные, предназначенные *не* для него. В экстремальной ситуации в кэше может оказаться больше ненужных, чем нужных данных. Впустую будет расходоваться драгоценное пространство кэш-памяти, и увеличится вероятность того, что процессор не обнаружит в кэше нужных данных и ему придется извлекать элемент данных из основной памяти, даже если он когда-то уже сохранял их в кэше. А все из-за того, что ему пришлось удалить элемент из кэша, освобождая место для другого элемента.

В данной ситуации это важно для однопоточного кода, так зачем все это рассматривать именно здесь? Все дело в *переключении задач*. Если количество потоков превышает количество ядер в системе, то каждому ядру приходится запускать по несколько потоков. Это увеличивает нагрузку на кэш-память, когда с целью избежать ложного совместного использования памяти предпринимаются попытки обеспечить обращение разных потоков к разным строкам кэша. Следовательно, когда процессор переключает потоки, притом что каждый из потоков задействует данные, разбросанные по нескольким строкам кэша, ему скорее придется перезагрузить строки кэша, чем когда данные каждого потока располагаются близко друг к другу в одной и той же строке кэша. В стандарте C++17, также в заголовке `<new>`, определена константа `std::hardware_constructive_interference_size`, в которой указывается максимальное количество последовательных байтов, гарантированно находящихся в одной строке кэша (при приемлемом выравнивании). Если совместно востребуемые данные уложатся в это количество байтов, возможно, это сократит число промахов кэша.

Когда потоков больше, чем ядер или процессоров, операционная система может выбрать диспетчеризацию потока в один отрезок времени на одном ядре, а затем

в другой отрезок времени — на другом. По этой причине потребуется переносить строки кэша для данных этого потока из кэш-памяти первого ядра в кэш-память второго ядра: чем больше строк кэша нужно переносить, тем больше времени это займет. Хотя операционная система старается избегать подобного развития событий, такое все же случается и отрицательно влияет на производительность.

Проблемы переключения задач возникают особенно часто, когда много потоков готовы к запуску, а не к ожиданию. Мы уже сталкивались с данной проблемой, определив ее как переоценку вычислительных возможностей (oversubscription).

8.2.5. Переоценка вычислительных возможностей и чрезмерное количество переключений задач

В многопоточных системах, если только они не запущены на базе *мощного параллельно* работающего оборудования, в превышении количества потоков над количеством процессоров нет ничего необычного. Но потоки зачастую тратят время на ожидание завершения внешнего ввода-вывода, блокируются на мьютексах, ждут изменений условных переменных, и т. д., так что никакой проблемы это не создает. Наличие дополнительных потоков позволяет приложению выполнять полезную работу, не давая процессорам бездельничать, пока потоки находятся в режиме ожидания.

Но обилие потоков — это не всегда хорошо. Если дополнительных потоков слишком много, число готовых к запуску потоков превысит число доступных процессоров и операционной системе придется начать интенсивно переключать задачи, чтобы обеспечить им справедливое выделение отрезков времени. В главе 1 говорилось, что это может привести к увеличению издержек на переключение задач, а также возникновению проблем с кэш-памятью, связанных с недостаточной сближенностью данных. Переоценка вычислительных возможностей может возникать при наличии задачи, систематически порождающей все новые и новые потоки, как происходит при рекурсивной быстрой сортировке из главы 4 или когда естественное число потоков, порожденное при распределении задач по типам, превышает число процессоров, а рабочая нагрузка связана со счетными задачами, а не с вводом-выводом.

Если слишком много потоков порождается из-за распределения данных, количество рабочих потоков может быть ограничено, что и было показано в подразделе 8.1.2. Если переоценка вычислительных возможностей стала следствием естественного распределения работы, то смягчить проблему иначе, кроме как выбрав другой вариант распределения, вряд ли получится. В таком случае выбор подходящего распределения может потребовать лучшей осведомленности о целевой платформе, превышающей доступную, и добиваться ее стоит только при недопустимо низком уровне производительности и возможности продемонстрировать, что смена способа распределения работы действительно повысит производительность.

На производительность многопоточного кода способны повлиять и другие факторы. Например, затраты на перебрасывание данных в кэш-памяти между двумя одноядерными и одним двухъядерным процессорами могут существенно варьироваться даже при одинаковом типе процессоров и одной и той же тактовой частоте, но мы рассмотрели наиболее важные факторы, заметно влияющие на производительность. А теперь посмотрим, как они воздействуют на конструкцию кода и структур данных.

8.3. Разработка структур данных для высокопроизводительных многопоточных приложений

В разделе 8.1 рассматривались различные способы распределения работы между потоками, а в разделе 8.2 — факторы, способные повлиять на производительность вашего кода. Как воспользоваться этой информацией при разработке структур данных для многопоточного выполнения программ? Этот вопрос отличается от того, чему были посвящены главы 6 и 7, где речь шла о разработке структур данных, безопасных с точки зрения конкурентного доступа к данным. В разделе 8.2 было показано, что размещение в памяти данных, используемых одним потоком, может воздействовать на производительность, даже если данные не используются совместно с другими потоками.

Главное, что нужно учитывать при разработке структур данных для многопоточного выполнения программ, — это *конкуренция*, *ложное совместное использование данных* и *сближенность данных*. Все три фактора могут существенно повлиять на производительность, и зачастую ситуацию можно улучшить, изменив расположение данных в памяти или порядок назначения элементов данных конкретным потокам. Сначала рассмотрим самое простое — распределение элементов массива между потоками.

8.3.1. Распределение элементов массива для сложных операций

Предположим, проводятся интенсивные математические вычисления и нужно перемножить две большие квадратные матрицы. Чтобы получить верхний левый элемент результата, каждый элемент первой *строки* первой матрицы перемножается с соответствующим элементом первого *столбца* второй матрицы и полученные произведения складываются. Затем все это повторяется со второй строкой и первым столбцом, чтобы получить второй элемент первого столбца результата, и с первой строкой и вторым столбцом, чтобы получить первый элемент второго столбца результата, и т. д. Этот процесс показан на рис. 8.3, где выделением продемонстрировано, что вторая строка первой матрицы соотносится с третьим столбцом второй матрицы, чтобы получить элемент во второй строке третьего столбца результата.

Теперь, чтобы был смысл воспользоваться для оптимизации умножения несколькими потоками, предположим, что имеются *большие* матрицы из нескольких тысяч строк и столбцов. Обычно, если матрица не является разреженной, она представляется в памяти большим массивом, где за всеми элементами первой строки следуют элементы второй строки и т. д. Для перемножения матриц есть три огромных массива. Чтобы достичь оптимальной производительности, нужно обратить пристальное внимание на схемы обращения к данным, в особенности записи в третий массив.

Способов разделения работы между потоками довольно много. Если предположить, что строк и столбцов больше, чем доступных процессоров, то каждый поток

можно назначить для вычисления значений нескольких столбцов в матрице результата, или же вычисления результатов для нескольких строк, или даже вычисления результатов прямоугольной части матрицы.

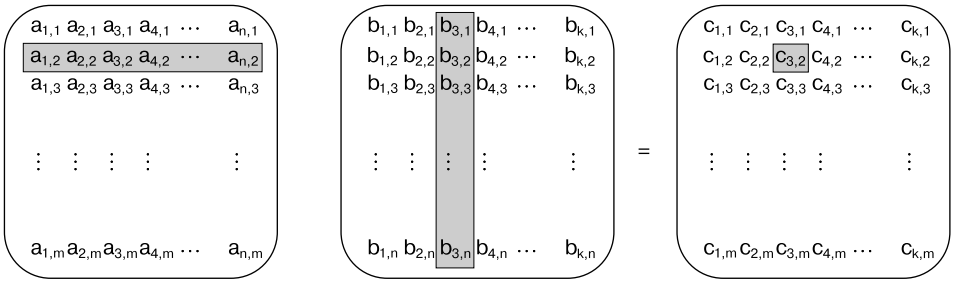


Рис. 8.3. Перемножение матриц

В подразделах 8.2.3 и 8.2.4 было показано, что лучше обращаться к смежным элементам массива, а не к разбросанным повсюду значениям, тогда снизятся интенсивность задействования кэш-памяти и вероятность возникновения ложного совместного использования памяти. Если назначить каждому потоку вычисление набора столбцов, ему придется считывать каждое значение из первой матрицы и значения из соответствующих столбцов второй матрицы, но записывать придется только значения столбцов. С учетом того что матрицы хранятся со смежными строками, это означает, что происходит обращение к N элементам из первой строки, N элементам из второй и т. д., где N — количество обрабатываемых столбцов. Поскольку другие потоки будут обращаться к другим элементам каждой строки, понятно, что вам следует обращаться к смежным столбцам, следовательно, N элементов из каждой строки являются смежными и вероятность ложного совместного использования памяти сводится к минимуму. Если пространство, занимаемое N элементами, точно вписывается в несколько строк кэша, то ложного совместного использования памяти не будет, поскольку потоки будут работать с отдельными строками кэша.

В то же время, если каждый поток будет вычислять набор *строк*, ему нужно будет считывать каждое значение из *второй* матрицы и значения из соответствующих строк *первой* матрицы, а записывать придется только значения строки. Поскольку матрицы хранятся со смежными строками, теперь обращение выполняется ко всем элементам N строк. Если опять выбрать смежные строки, значит, теперь поток будет единственным выполняющим запись в эти N строк, он располагает непрерывным блоком памяти, к которому не обращаются другие потоки. Наверное, так будет лучше, чем когда каждый поток вычисляет набор столбцов, потому что единственная вероятность ложного совместного использования памяти относится к последним нескольким элементам одного блока и первым нескольким элементам следующего блока, но для подтверждения этого обстоятельства стоит измерить расход времени на целевой архитектуре.

А как насчет третьего варианта — распределения работы по прямоугольным блокам? Его можно рассматривать как распределение по столбцам, а затем распределение

по строкам. Таким образом, у него такая же вероятность ложного совместного использования памяти, как и у распределения по столбцам. Если, чтобы избежать этого, выбрать число столбцов в блоке, то распределение по прямоугольным блокам получит преимущество при чтении: не нужно будет читать всю исходную матрицу. Придется только прочитать значения, соответствующие строкам и столбцам целевого прямоугольника. Чтобы конкретизировать представление, рассмотрим перемножение двух матриц, имеющих 1000 строк и 1000 столбцов. Получается 1 млн элементов. При наличии 100 процессоров каждый сможет вычислять по 10 строк, что составит круглое число 10 000 элементов. Но для вычисления результатов этих 10 000 элементов ко второй матрице придется обращаться целиком (к 1 млн элементов) плюс к 10 000 элементов — из соответствующих строк первой матрицы, что в целом составит 1 010 000 элементов. В то же время, если каждый из процессоров вычисляет блок 100 × 100, то есть всего 10 000 элементов, придется обращаться к значениям из 100 строк первой матрицы ($100 \cdot 1\,000 = 100\,000$ элементов) и из 100 столбцов второй матрицы (еще 100 000). Это всего 200 000 элементов, что уменьшает количество считываемых элементов в пять раз. Чем меньше считывается элементов, тем ниже вероятность промаха при обращении к кэш-памяти и выше возможность достижения более высокой производительности.

Поэтому, возможно, лучше распределить матрицу результата на небольшие квадратные или почти квадратные блоки, чем вычислять в каждом потоке совокупность небольшого количества строк. Размер каждого блока можно подогнать в ходе выполнения программы в зависимости от размера матриц и доступного числа процессоров. Как обычно, если производительность играет значимую роль, важно предоставить различные варианты целевой архитектуры и изучить литературу в этой области (я не утверждаю, что при перемножении матриц это единственно возможные или наилучшие варианты).

Скорее всего, вам не приходится перемножать матрицы, тогда какое отношение все это имеет к вам? Дело в том, что те же принципы применимы к любой ситуации, где имеются большие блоки данных, распределяемые между потоками. Следует внимательно изучить все аспекты схем доступа к данным и определить возможные причины снижения производительности. В проблемной области, с которой придется столкнуться, могут сложиться аналогичные обстоятельства, при которых изменения в распределении работы могут повысить производительность, не требуя для этого каких-либо изменений основного алгоритма.

Итак, нами изучено влияние на производительность схем обращения к массивам. А как насчет других типов структур данных?

8.3.2. Схемы доступа к данным в других структурах данных

В принципе, при оптимизации схем обращения к элементам других структур данных применяются те же соображения, что и при оптимизации обращения к массивам.

- Настройте распределение данных между потоками таким образом, чтобы смежные данные обрабатывались одним и тем же потоком.

- Сведите к минимуму объем данных, востребованный любым отдельно взятым потоком.
- Руководствуясь значением константы `std::hardware_destructive_interference_size`, обеспечьте достаточную удаленность друг от друга данных, к которым выполняется обращение из разных потоков, чтобы избежать ложного совместного использования данных.

Применение этих рекомендаций к другим структурам данных не обойдется без трудностей. Например, сама природа двоичного дерева создает проблемы при распределении по частям, отличным от поддерева, что в зависимости от сбалансированности дерева и количества разделов, на которые нужно его разделить, может иметь как положительный, так и отрицательный эффект. К тому же особенности деревьев диктуют преимущественно динамическое распределение памяти под узлы, приводящее к выделению для них различных мест в куче.

Разбросанность данных по куче сама по себе не проблема, но это означает, что процессору приходится хранить в кэш-памяти больше данных. Из этого можно извлечь пользу. Если проход по дереву нужен нескольким потокам, значит, всем им требуется доступ к узлам дерева. Но если эти узлы содержат только указатели на реально хранящиеся в узле данные, то, если они востребованы, процессору придется лишь загрузить их из памяти. Если данные изменяются потоками, которым нужны эти изменения, сложившиеся обстоятельства могут позволить обойтись без снижения производительности, возникающего из-за ложного совместного использования памяти между данными узлов и данными, предоставляющими структуру дерева.

Похожая проблема возникает и с данными, защищенными мьютексом. Предположим, имеется простой класс, состоящий из нескольких элементов данных, и мьютекс для защиты доступа со стороны нескольких потоков. Если мьютекс и элементы данных располагаются в памяти близко друг к другу, это будет идеальный случай для потока, удерживающего мьютекс: нужные ему данные уже могут находиться в кэш-памяти процессора, потому что были загружены для изменения мьютекса. Но есть и проблема: если другие потоки пытаются заблокировать мьютекс, удерживаемый первым потоком, им придется обращаться к той же памяти. Блокировки мьютекса в попытках завладеть им обычно реализуются в форме атомарной операции чтения — изменения — записи в отношении места в памяти внутри мьютекса, за которой следует вызов ядра операционной системы, если мьютекс уже заблокирован. Эта операция чтения — изменения — записи может привести к тому, что станут неактуальны данные, сохраненные в кэше тем потоком, который владеет мьютексом. Пока мьютекс удерживается, проблемы не возникает: этот поток не собирается обращаться к мьютексу, пока не разблокирует его. Но если мьютекс использует строку кэша совместно с данными, с которыми работает поток, то поток, владеющий мьютексом, может снизить производительность из-за того, что мьютекс пытается заблокировать другой поток!

Один из способов проверки того, создает ли подобное ложное совместное использование памяти проблему, заключается в добавлении огромных блоков заполнения между элементами данных, к которым могут конкурентно обращаться разные

потоки. Например, чтобы проверить, есть ли конкуренция за владение мьютексом, можно воспользоваться следующей структурой:

```
struct protected_data
{
    std::mutex m;
    char padding[std::hardware_destructive_interference_size];
    my_data data_to_protect;
};
```

Если константа std::hardware_destructive_interference_size недоступна вашему компилятору, можно, к примеру, воспользоваться чем-то вроде 65 536 байт, что будет, наверное, на несколько порядков больше строки кэша

А для проверки ложного совместного использования данных массива — вот этой структурой:

```
struct my_data
{
    data_item1 d1;
    data_item2 d2;
    char padding[std::hardware_destructive_interference_size];
};
my_data some_array[256];
```

Если производительность повысится, то станет понятно, что проблема заключалась в ложном совместном использовании памяти и можно либо оставить заполнение, либо поработать над исключением ложного совместного использования памяти иным способом за счет перестановки обращения к данным.

Но при разработке конкурентных программ нужно уделять внимание не только схемам обращения к данным, поэтому рассмотрим дополнительные факторы.

8.4. Дополнительные факторы, учитываемые при разработке конкурентных программ

Итак, уже рассмотрены способы распределения работы между потоками, факторы, воздействующие на производительность, и то, как они способны повлиять на выбор схем обращения к данным и структур данных. Но для разработки конкурентных программ этого недостаточно. Нужно также учитывать безопасность выдачи исключений и *масштабируемость*. Код называют масштабируемым, если производительность (в понятиях снижения скорости выполнения или увеличения пропускной способности) растет с добавлением к системе вычислительных ядер. В идеале производительность растет прямо пропорционально, и система, имеющая 100 процессоров, выполняет код в 100 раз быстрее системы с одним процессором.

Хотя код может работать, даже не обладая масштабируемостью (к примеру, однопоточное приложение лишено этой возможности по определению), безопасность исключений — это вопрос приличия. Если код не обладает безопасностью исключений, можно столкнуться с нарушенными инвариантами или состоянием гонки либо приложение может неожиданно прекратить работу из-за того, что операция выдала исключение. Исходя из этого, сначала рассмотрим безопасность исключений.

8.4.1. Безопасность исключений в параллельных алгоритмах

Безопасность исключений — важный аспект качественного кода C++, и код, использующий конкурентность, здесь не исключение. Фактически параллельные алгоритмы зачастую требуют более пристального внимания к исключениям, чем обычные последовательные алгоритмы. Если операция в последовательном алгоритме выдает исключение, алгоритму приходится беспокоиться только о ликвидации последствий своей деятельности, чтобы избежать утечки ресурсов и нарушения инвариантов. Он может просто позволить исключению распространиться на вызывающий код, чтобы тот занялся его обработкой. В отличие от этого в параллельном алгоритме несколько операций будут запущены в отдельных потоках. В данном случае исключение не способно распространяться, так как находится не в том стеке вызовов. Если из функции, породившей новый поток, происходит выход с выдачей исключения, приложение прекращает работу.

В качестве конкретного примера еще раз обратимся к воспроизведенной далее функции `parallel_accumulate` из листинга 2.9 (листинг 8.2).

Листинг 8.2. Простейшая параллельная версия std::accumulate (из листинга 2.9)

```
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first,Iterator last,T& result)
    {
        result=std::accumulate(first,last,result);    ←❶
    }
};
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);   ←❷
    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;
❸→  std::vector<T> results(num_threads);
    std::vector<std::thread> threads(num_threads-1);    ←❹
❺→  Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;          ←❻
        std::advance(block_end,block_size);
        threads[i]=std::thread(                  ←❼
```

```
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
❽ →    block_start=block_end;
    }
    accumulate_block<Iterator,T>()(
        block_start,last,results[num_threads-1]);    ←❾
    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));
    return std::accumulate(results.begin(),results.end(),init);  ←❿
}
```

Теперь пройдемся по коду и выявим участки, где возможны исключения: известно, что это может произойти в местах вызова функции или выполнения операций над типом, определенным пользователем.

В начале имеется вызов функции `distance` ❷, где выполняются операции над определенным пользователем типом итератора. Пока никакой работы не выполнено и все происходит в рамках вызывающего потока, поэтому нас все устраивает. Далее распределяется память под векторы `results` ❸ и `threads` ❹. Здесь тоже все делается в рамках вызывающего потока и не выполняется никакой работы, а также не порождается ни одного потока, и нас опять это устраивает. Если исключение будет выдано при работе конструктора `threads`, то память, распределенную под `results`, нужно будет освободить, но за вас об этом побеспокоится деструктор.

Инициализацию `block_start` ❺ пропускаем, поскольку она безопасна по тем же соображениям, и переходим к операциям в цикле порождения потоков ❻, ❼ и ❽. После прохода через создание первого потока ❼ возникает беспокойство о том, будут ли выданы какие-либо исключения: деструкторы новых объектов `std::thread` вызовут `std::terminate` и прервут выполнение программы. Такое развитие событий нас не устроит.

Вызов `accumulate_block` ❾ также может выдать исключение с такими же последствиями: ваши объекты `thread` будут уничтожены и произойдет вызов `std::terminate`. В то же время завершающий вызов `std::accumulate` ❿ может выдать исключение без каких-либо осложнений, потому что в данном месте кода все потоки уже объединены.

С основным потоком разобрались, но это еще не все: исключение может быть выдано при вызове `accumulate_block` в новых потоках ❶. Здесь нет `catch`-блоков, и исключение останется необработанным и заставит библиотеку вызвать `std::terminate()`, чтобы прервать выполнение приложения.

Пусть это и не вполне очевидно, но данный код нельзя считать безопасным при выдаче исключений.

Добавление безопасности исключений

Итак, мы выявили все возможные места выдачи исключений и определили опасные последствия этого процесса. Что же можно сделать? Начнем с ликвидации последствий исключений, выдаваемых в новых потоках.

Средства, позволяющие справиться с этой задачей, встречались в главе 4. Если задуматься о том, чего нужно добиться от новых потоков, станет очевидным, что

требуется вычислить результат для возврата и при этом учесть возможность выдачи исключения. Именно для этого разработана комбинация `std::packaged_task` и `std::future`. Если переделать код для применения `std::packaged_task`, получится следующий результат (листинг 8.3).

Листинг 8.3. Параллельная версия std::accumulate с использованием std::packaged_task

```
template<typename Iterator,typename T>
struct accumulate_block
{
    T operator()(Iterator first,Iterator last)    ←❶
    {
        return std::accumulate(first,last,T());      ←❷
    }
};
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!=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);     ←❸
    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(  ←❹
            accumulate_block<Iterator,T>());
  ❺→    futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task),block_start,block_end); ←❻
        block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);  ←❼
    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));
❽→  T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();    ←❾
    }
    result += last_result;   ←❿
    return result;
}
```

Первое изменение заключается в том, что теперь оператор вызова функции `accumulate_block` возвращает результат напрямую, а не получает ссылку на то место, где его нужно сохранить ❶. Чтобы обеспечить безопасность исключений, используют `std::packaged_task` и `std::future`, поэтому их можно применить и при передаче результата. Для этого потребуется явно передать сконструированное по умолчанию значение `T` в вызов `std::accumulate` ❷, а не повторно задействовать предоставленное значение `result`, но это несущественное изменение.

Следующее изменение заключается в замене вектора результатов вектором `futures` ❸ для хранения `std::future<T>` для каждого порожденного потока. В цикле порождения потоков сначала создается задача для `accumulate_block` ❹. В классе `std::packaged _task<T(Iterator, Iterator)>` объявляется задача, получающая два итератора `Iterator` и возвращающая `T`, чем, собственно, и занимается ваша функция. Затем происходит получение фьючерса для этой задачи ❺, и она запускается в новом потоке с переданными ей началом и концом обрабатываемого блока ❻. Когда задача выполняется, результат, а также любое выданное исключение будут перехвачены и сохранены во фьючерсе.

Из-за использования фьючерсов массива результатов больше нет, поэтому результат из финального блока нужно сохранить в переменной ❼, а не в элементе массива. А так как значения придется извлекать из фьючерсов, то теперь проще будет воспользоваться простым циклом `for`, а не `std::accumulate`, начиная с предоставления исходного значения ❽ и прибавления результата из каждого фьючерса ❾. Если какая-либо задача выдаст исключение, оно будет захвачено фьючерсом и снова выдано при вызове функции `get()`. И наконец, перед возвращением вызывающему коду всего результата прибавляется результат из последнего блока ❿.

Таким образом удалось избавиться от одной из потенциальных проблем — исключения, выданные в рабочих потоках, повторно выдаются в основном потоке. Если исключение будет выдано более чем одним потоком, то распространение получит лишь одно, но это не так уж и важно. Если это произойдет, то для перехвата всех исключений можно будет воспользоваться чем-нибудь вроде класса `std::nested_exception` и выдать вместо них одно исключение.

Остается решить проблему утечки потоков при выдаче исключения между порождением первого потока и объединением со всеми потоками. Простейшее решение — перехватывать любые исключения, объединять со всеми потоками, которые все еще положительно отзываются на вызов функции `joinable()`, а затем выдать исключение повторно:

```
try
{
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        // ... как и раньше
    }
    T last_result=accumulate_block<Iterator,T>()(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;
}
```

Вот теперь все работает. Все потоки будут присоединены независимо от того, каким образом код покинет блок. Но блоки `try-catch` выглядят уродливо, и при этом пришлось продублировать код. Потоки объединятся как при нормальном потоке управления, так и в блоке `catch`. Дублирование кода редко ставится в заслугу разработчику, поскольку из-за него увеличивается количество мест внесения изменений. Вместо этого давайте извлечем данный фрагмент и вставим его в деструктор объекта — все-таки именно такой способ очистки ресурсов присущ языку C++. Тогда класс присоединения потоков приобретет следующий вид:

```
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, за исключением того, что он расширяется на весь вектор потоков. Затем код можно упростить следующим образом (листинг 8.4)

Листинг 8.4. Параллельная версия std::accumulate с безопасными исключениями

```
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!=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);
❶ ──►   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<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<Iterator,T>()(block_start,last);
        T result=init;
        for(unsigned long i=0;i<(num_threads-1);++i)
        {
            result+=futures[i].get();          ◄──❷
        }
        result += last_result;
        return result;
    }
```

После создания контейнера потоков создается экземпляр вашего нового класса ❶, чтобы на выходе он объединился со всеми потоками. Затем явно указанный цикл объединения можно удалить, будучи уверенными, что потоки будут объединены независимо от того, каким образом завершится работа функции. Следует отметить, что вызовы `futures[i].get()` ❷ будут блокироваться до готовности результатов, поэтому здесь явно объединять потоки не требуется. Этим код отличается от исходной версии из листинга 8.2, где требовалось объединение с потоками, чтобы гарантировать правильное заполнение вектора `results`. Но в результате был получен не только код, безопасный при выдаче исключений, но и более короткие функции, поскольку код объединения извлекался в новый, пригодный для повторного использования класс.

Обеспечение безопасности исключений с помощью std::async()

Изучив все, что требуется для обеспечения безопасности исключений при явном управлении потоками, посмотрим, как то же самое достигается с помощью `std::async()`. Как уже было показано, в этом случае всю заботу об управлении потоками берет на себя библиотека и любые порожденные потоки завершаются по готовности фьючерсов. Здесь для безопасности исключений важно отметить, что, если удалить фьючерс, не дожидаясь его готовности, деструктор станет ждать завершения потока. Это позволяет весьма изящно обойти проблему утечки все еще выполняемых

потоков и сохранения ссылок на данные. В листинге 8.5 показан код безопасной при выдаче исключений реализации, в котором используется `std::async()`.

Листинг 8.5. Параллельная, безопасная при выдаче исключений версия std::accumulate, применяющая std::async

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last); ❶
    unsigned long const max_chunk_size=25;
    if(length<=max_chunk_size)
    {
        return std::accumulate(first,last,init); ❷
    }
    else
    {
    Iterator mid_point=first;
        std::advance(mid_point,length/2);  ❸
        std::future<T> first_half_result=
            std::async(parallel_accumulate<Iterator,T>, ❹
                       first,mid_point,init);
        T second_half_result=parallel_accumulate(mid_point,last,T()); ❺
        return first_half_result.get()+second_half_result; ❻
    }
}
```

В этой версии применяется рекурсивное распределение данных, а не предварительно вычисленное разделение данных на части, но в целом код стал проще, чем в предыдущей версии, сохранив при этом безопасность выдачи исключений. Как и прежде, сначала определяется длина последовательности ❶, и, если она меньше максимального размера части, `std::accumulate` вызывается напрямую ❷. Если элементов больше, чем размер части, определяется средняя точка ❸, после чего порождается асинхронная задача обработки первой половины ❹. Вторая половина диапазона обрабатывается с помощью непосредственного рекурсивного вызова ❺, а затем результаты этих двух частей складываются ❻. Библиотека гарантирует, что вызовами `std::async` используются доступные аппаратные потоки и не создается чрезмерное количество потоков. Некоторые асинхронные вызовы будут выполнены в синхронном режиме при вызове функции `get()` ❻.

Достоинствами этого решения являются не только возможность воспользоваться конкурентностью, обеспеченной на аппаратном уровне, но и заведомая безопасность выдачи исключений. Если исключение выдается рекурсивным вызовом ❺, то фьючерс, созданный из вызова `std::async` ❺, будет уничтожен, как только исключение получит распространение. Это приведет к ожиданию завершения асинхронной задачи, позволяя избежать появления «подвисшего» потока. В то же время, если исключение выдаст асинхронный вызов, оно будет перехвачено фьючерсом, а вызов функции `get()` ❻ приведет к повторной выдаче исключения.

Какие еще факторы следует учесть при разработке конкурентного кода? Давайте рассмотрим *масштабируемость*. Насколько вырастет производительность, если переместить код в систему с увеличенным числом процессоров?

8.4.2. Масштабируемость и закон Амдала

Масштабируемость — гарантия того, что приложение способно воспользоваться дополнительными процессорами, имеющимися в системе, в которой оно запущено. В качестве одной крайности у вас может быть однопоточное, совершенно не масштабируемое приложение: даже если к системе добавить еще 100 процессоров, производительность не изменится. В качестве другой крайности у вас может быть проект, подобный SETI@Home (http://setiathome.ssl.berkeley.edu/), разработанный, чтобы воспользоваться преимуществами тысяч дополнительных процессоров (в виде отдельно взятых компьютеров, добавляемых пользователями по сети) по мере того, как они становятся доступными.

При запуске любой отдельно взятой многопоточной программы число потоков, выполняющих полезную работу, будет варьироваться. Даже если каждый поток выполняет полезную работу в течение всего времени своей жизни, приложение изначально может иметь только один поток, который получит задание породить все остальные потоки. Но даже такой сценарий маловероятен. Потоки зачастую проводят время в ожидании друг друга или завершения операций ввода-вывода.

Если одному потоку приходится ждать чего бы то ни было, то, когда нет другого потока, готового занять его место на процессоре, процессор, способный выполнять полезную работу, простаивает впустую.

Упрощенный способ изучения ситуации состоит в разделении программы на «последовательные» части, где любую полезную работу выполняет только один поток, и «параллельные» части, где полезной работой занимаются все доступные процессоры. Если запустить приложение в системе с увеличенным числом процессоров, «параллельные» части теоретически смогут завершать свою работу быстрее, поскольку ее можно распределить между процессорами, количество которых возросло, а «последовательные» части так и останутся последовательными. При таком упрощенном наборе предположений можно оценить потенциальное повышение производительности, достигаемое за счет увеличения числа процессоров следующим образом: если «последовательные» части составляют долю программы, обозначаемую как fs, то повышение производительности P от использования N процессоров можно оценить как:

$$P = \frac{1}{f_s + \frac{1 - f_s}{N}}.$$

Эта формула описывает *закон Амдала* — его часто цитируют, когда речь заходит о производительности конкурентного кода. Если все можно распараллелить, значит, доля последовательно выполняемого кода равна нулю и ускорение равно N. Если же доля последовательно выполняемого кода составляет 1/3, то даже при бесконечном числе процессоров не удастся достичь ускорения больше 3.

Но представленная картина слишком упрощена, поскольку задачи довольно редко делятся до бесконечности, как требует уравнение, также далеко не все задачи являются вычислительными, как здесь предполагается. Уже было показано, что потоки в процессе выполнения могут находиться в состоянии ожидания наступления множества разнообразных событий.

Из закона Амдала можно сделать один вывод: когда конкурентность используется для повышения производительности, стоит присмотреться ко всей конструкции приложения, чтобы максимально расширить возможности конкурентности и обеспечить постоянную загруженность процессоров полезной работой. Если есть возможность сократить размер «последовательных» частей или снизить вероятность ожидания потоков, то можно повысить возможность роста производительности систем с увеличенным числом процессоров. В качестве альтернативы нужно исследовать возможность предоставления системе большего объема обрабатываемых данных, что позволит загрузить «параллельные» части работой, тогда можно будет уменьшить долю «последовательной» части и повысить производительность *P*.

Цель масштабирования заключается в *сокращении времени на выполнение действия или увеличении объема данных, обрабатываемого за определенное время*, по мере добавления процессоров. Иногда одно вытекает из другого (если на каждый элемент затрачивается меньше времени, то можно обработать больше данных), но так бывает не всегда. Прежде чем выбрать способ, применяемый для распределения работы между потоками, важно определить, какие аспекты масштабирования для вас важны.

В начале раздела уже упоминалось, что потоки не всегда бывают загружены полезной работой. Иногда им приходится ожидать результатов работы других потоков, или завершения операций ввода-вывода, или чего-нибудь еще. Если дать системе заняться чем-либо полезным во время ожидания, то можно вполне эффективно скрасить его.

8.4.3. Компенсация потерь на ожидание за счет применения нескольких потоков

В большинстве случаев, когда речь заходит о производительности многопоточного кода, предполагается, что потоки работают на износ и всегда загружены полезной работой, когда запущены на процессоре. Но это не так: в коде приложения потоки зачастую блокируются в ожидании наступления какого-то события. Например, они могут ожидать завершения операции ввода-вывода, возможности овладеть мьютексом, завершения операции в каком-нибудь другом потоке и уведомления с помощью условной переменной или заполнения фьючерса либо даже впасть на какое-то время в спячку.

Какой бы ни была причина ожидания, если число потоков не превышает количества имеющихся в системе вычислительных блоков, то наличие заблокированных потоков означает: процессорное время расходуется впустую. Процессор, который мог бы выполнять заблокированный поток, вместо этого ничего не будет делать. Следовательно, если известно, что один из потоков, скорее всего, существенную часть своего времени станет проводить в ожидании, то можно воспользоваться свободным временем центрального процессора для запуска одного или нескольких дополнительных потоков.

Рассмотрим приложение, сканирующее компьютер на вирусы, где для распределения работы между потоками применяется конвейер. Первый поток ищет в файловой системе файлы для проверки и помещает их в очередь. Тем временем другой поток берет имена файлов из очереди, загружает файлы и сканирует их на вирусы. Известно, что поток, отыскивающий в системе файлы, которые нужно

просканировать, испытывает ограничения, связанные с вводом-выводом, поэтому свободное процессорное время используется для запуска дополнительного потока сканирования. Получается, что есть один поток, выполняющий поиск в файловой системе, и сканирующие потоки по числу имеющихся в системе физических ядер или процессоров. Поскольку на сканирующий поток для выполнения его функции может также возлагаться чтение с диска значительных частей файлов, возможно, имеет смысл обзавестись дополнительными сканирующими потоками. Но в какой-то момент возникнет переизбыток потоков и система снова начнет тормозить, так как все больше времени будет уходить на переключение (см. описание в подразделе 8.2.5).

Как всегда, речь идет об оптимизации, поэтому важно измерять производительность до и после любого изменения числа потоков: их оптимальное количество будет сильно зависеть от характера выполняемой работы и процентного показателя времени, затрачиваемого потоком на ожидание.

В зависимости от характера приложения свободным временем центрального процессора можно воспользоваться и без запуска дополнительных потоков. Например, если поток заблокирован и ожидает завершения операции ввода-вывода, вероятно, есть смысл воспользоваться асинхронным вводом-выводом, и тогда поток сможет выполнять другую полезную работу, пока ввод-вывод выполняется в фоновом режиме. В других случаях поток вместо того, чтобы в заблокированном виде ждать, пока другой поток выполнит свою операцию, мог бы сам выполнить ее, как было с рассмотренной в главе 7 очередью, свободной от блокировок. В крайнем случае, если поток ожидает завершения задачи, еще не запущенной другим потоком, он может выполнить эту задачу самостоятельно или же выполнить другую еще не завершенную задачу. Подобный пример встретился в листинге 8.1, где функция `sort` неоднократно пытается отсортировать остающиеся в очереди фрагменты, пока нужные ей фрагменты еще не отсортированы.

Иногда потоки полезно добавлять не для того, чтобы обеспечить загруженность работой всех доступных процессоров, а для своевременной обработки внешних событий, чтобы тем самым повысить *отзывчивость* системы.

8.4.4. Повышение отзывчивости за счет конкурентности

Большинство современных сред графического пользовательского интерфейса являются *событийно-управляемыми*: пользователь, нажимая клавиши или перемещая указатель мыши, совершает в интерфейсе какие-то действия, генерирующие серию событий или сообщений, которые затем обрабатываются приложением. Система также может самостоятельно формировать сообщения или выдавать события. Чтобы гарантированно правильно обрабатывать все события и сообщения, приложение обычно располагает циклом событий следующего вида:

```
while(true)
{
    event_data event=get_event();
    if(event.type==quit)
        break;
    process(event);
}
```

Разумеется, детали API могут различаться, но структура в целом одна и та же: дождаться события, сделать все необходимое для его обработки, а затем ждать следующего события. Если приложение однопоточное, это может затруднить создание долговременных задач (см. описание, приведенное в подразделе 8.1.3). Чтобы гарантировать своевременную обработку пользовательского ввода, функции `get_event()` и `process()` следует вызывать не реже разумного, что и делает приложение. Это означает, что либо задача должна периодически приостанавливаться и возвращать управление циклу обработки событий, либо код функций `get_event()` и `process()` должен вызываться из кода в подходящих для этого местах. При любом варианте реализация задачи усложняется.

Разделив обязанности с помощью конкурентности, можно поместить выполнение долговременной задачи в новый поток, а обработку событий возложить на специально выделенный GUI-поток. Тогда потоки смогут обмениваться данными через простые механизмы и не нужно будет каким-то образом смешивать код обработки событий с кодом задачи. Простая схема такого разделения показана в листинге 8.6.

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 пользовательский поток будет постоянно готов быстро реагировать на события, даже если выполнение задачи займет много времени. Отзывчивость зачастую определяет пользовательское восприятие приложения: теми приложениями, которые полностью блокируются при выполнении определенной операции, какой бы она ни была, пользоваться крайне неудобно. За счет предоставления специально выделенного потока обработки событий GUI-интерфейс сможет обрабатывать касающиеся его сообщения (например, об изменении размера или перерисовке окна), не прерывая долговременной обработки, но передавая соответствующие сообщения, если они влияют на реализацию задачи с длительным сроком выполнения.

До сих пор в этой главе мы подробно разбирали все, что нужно учитывать при разработке конкурентного кода. Решение всех обозначенных проблем может показаться абсолютно неподъемным, но по мере того как вы привыкнете работать с прицелом на многопоточное программирование, многие решения органично впишутся в ваш стиль. Даже если все эти соображения вам пока непривычны, надеюсь, все станет понятнее по мере изучения их влияния на конкретные примеры многопоточного кода.

8.5. Разработка конкурентного кода на практике

В какой степени следует учитывать каждый из рассмотренных ранее вопросов, будет зависеть от самой задачи. Чтобы продемонстрировать это, разберем реализацию параллельных версий трех функций из стандартной библиотеки C++. Так у нас будет знакомая платформа, на основе которой можно будет изучать новые темы. В качестве бонуса мы получим работоспособные реализации функций, которыми можно воспользоваться при распараллеливании более крупной задачи.

Эти реализации выбраны не потому, что они лучше других, просто они подходят для демонстрации конкретных приемов. Более совершенные реализации, в которых лучше используются доступные аппаратные средства, можно найти в научной литературе по параллельным алгоритмам или в специализированных многопоточных

библиотеках, например в разработанной компанией Intel библиотеке Threading Building Blocks (http://threadingbuildingblocks.org/).

Концептуально самый простой параллельный алгоритм используется в параллельной версии `std::for_each`, с него и начнем.

8.5.1. Параллельная реализация std::for_each

По идее, в функции `std::for_each` нет ничего сложного: она вызывает предоставленную пользователем функцию поочередно для каждого элемента диапазона. Существенным отличием параллельной реализации от последовательной функции `std::for_each` является порядок вызовов функции. В `std::for_each` функция вызывается с первым элементом диапазона, затем со вторым и т. д., а в параллельной реализации гарантии поочередной обработки элементов нет и они могут (и, надеемся, будут) обрабатываться одновременно.

Чтобы реализовать параллельную версию этой функции, диапазон следует разбить на наборы элементов, обрабатываемых каждым потоком. Число элементов известно заранее, поэтому данные можно разделить до начала обработки (см. подраздел 8.1.1). Будем исходить из того, что это единственная запущенная параллельная задача, поэтому для определения количества потоков можно воспользоваться функцией `std::thread::hardware_concurrency()`. Известно также, что элементы можно обрабатывать независимо друг от друга, поэтому, чтобы избежать ложного совместного использования памяти, допустимо воспользоваться смежными блоками (см. подраздел 8.2.3).

По замыслу алгоритм похож на параллельную версию `std::accumulate`, рассмотренную в подразделе 8.4.1, но вместо вычисления суммы элементов он просто должен применить к ним указанную функцию. Конечно, можно подумать, что это сильно упростит код, поскольку не требует возвращения результата, но, если нужно передавать исключения вызывающему коду, придется, как и прежде, использовать механизмы `std::packaged_task` и `std::future` для передачи исключения между потоками. Пример реализации показан в листинге 8.7.

Листинг 8.7. Параллельная версия 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);  ←❶
```

```
    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(  ←❷
            [=]()
            {
                std::for_each(block_start,block_end,f);
            });
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task));  ←❸
        block_start=block_end;
    }
    std::for_each(block_start,last,f);
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        futures[i].get();  ←❹
    }
}
```

Основная структура кода практически идентична показанной в листинге 8.4, что и неудивительно. Ключевое отличие состоит в том, что в векторе `futures` хранятся объекты `std::future<void>` ❶, так как рабочие потоки не возвращают значение, а для задачи используется простая лямбда-функция, вызывающая функцию `f` для диапазона от `block_start` до `block_end` ❷. Это избавляет от необходимости передавать диапазон в конструктор потока ❸. Поскольку рабочие потоки значений не возвращают, вызов `futures[i].get()` ❹ предоставляет средство извлечения любых исключений, выдаваемых в рабочих потоках. Если передача исключений не нужна, без этого вызова можно обойтись.

Как и в параллельной реализации `std::accumulate`, код `parallel_for_each` можно упростить за счет применения `std::async`. Эта версия показана в листинге 8.8.

Листинг 8.8. Параллельная версия std::for_each с использованием 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);  ←❶
    }
    else
    {
        Iterator const mid_point=first+length/2;
        std::future<void> first_half=            ←❷
```

```
            std::async(&parallel_for_each<Iterator,Func>,
                       first,mid_point,f);
        parallel_for_each(mid_point,last,f);  ❸
        first_half.get();                     ❹
    }
}
```

Здесь, как и в построенной на применении `std::async` версии `parallel_accumulate` из листинга 8.5, данные разбиваются не перед обработкой, а рекурсивно, поскольку неизвестно, сколько потоков будет использовать библиотека. Как и прежде, на каждой стадии данные делятся пополам, обработка одной половины запускается в асинхронном режиме ❷, другой — непосредственно ❸, а когда оставшиеся данные из-за их малого объема не станет смысла делить пополам, настанет черед функции `std::for_each` ❶. Здесь также семантика распространения исключений обеспечивается за счет применения `std::async` и компонентной функции `get()`, принадлежащей объекту `std::future` ❹.

Теперь перейдем от алгоритмов, выполняющих одну и ту же операцию над каждым элементом (а их навскидку припоминается несколько, включая `std::count` и `std::replace`), к более сложному примеру в виде `std::find`.

8.5.2. Параллельная реализация std::find

Алгоритм `std::find` подходит для дальнейшего рассмотрения, поскольку это один из нескольких алгоритмов, работа которых может завершаться и без обработки каждого элемента. Например, если первый элемент диапазона соответствует критерию поиска, то другие элементы анализировать уже не нужно. Как вскоре будет показано, это важное свойство существенно влияет на производительность, от него напрямую зависит и конструкция параллельной реализации. Это конкретный пример того, как схема обращения к данным может повлиять на конструкцию кода (см. подраздел 8.3.2). К другим алгоритмам этой категории относятся `std::equal` и `std::any_of`.

Если бы вы с кем-то еще искали старую фотографию в коробках на чердаке, то, обнаружив ее, вряд ли стали бы продолжать эту работу. Вы бы сообщили всем, что фотография найдена (возможно, воскликнув: «Вот она!»), и ваши помощники смогли бы прекратить поиск и заняться чем-то другим. Характер многих алгоритмов требует обработки каждого элемента, то есть у них нет эквивалента возгласу «Вот она!». Для таких алгоритмов, как `std::find`, возможность завершиться преждевременно является важным свойством, поступиться которым нельзя. Поэтому код нужно сконструировать так, чтобы можно было этим свойством воспользоваться — каким-то образом прервать другие задачи, когда ответ уже известен, чтобы программе не пришлось дожидаться, пока другие рабочие потоки не завершат обработку оставшихся элементов.

Если не прервать другие потоки, последовательная версия может обогнать параллельную реализацию, поскольку последовательный алгоритм способен остановить поиск и вернуть управление, как только будет найдено соответствие. Если, к примеру, система в состоянии поддерживать четыре конкурентных потока, то каждому из

них придется проверить четвертую часть элементов диапазона, и ваша примитивная параллельная реализация потратит примерно четвертую часть времени, требующегося одному потоку на проверку каждого элемента. Если искомый элемент находится в первой четверти диапазона, последовательный алгоритм вернет управление первым, так как ему не нужно будет проверять оставшиеся элементы.

Один из способов прерывания работы других потоков заключается в использовании в качестве флага атомарной переменной и проверке флага после обработки каждого элемента. Если флаг установлен, значит, какой-то другой поток уже нашел соответствие и можно прекратить обработку и вернуть управление. Прерывая работу потоков таким образом, вы сохраняете свойство необязательности проверки каждого значения и во многих случаях добиваетесь более высокой производительности, чем у последовательной версии. Недостаток этого способа — невысокая скорость выполнения операции загрузки атомарной переменной, что может снизить скорость работы каждого потока.

Теперь у нас есть два варианта возвращения значений и распространения исключений. Для передачи значений и исключений можно воспользоваться массивом фьючерсов `std::packaged_task`, после чего обработать результаты, вернувшись в основной поток. Или же воспользоваться `std::promise` для установки окончательного результата непосредственно из рабочих потоков. Все зависит от того, как нужно обрабатывать исключения из рабочих потоков. Если требуется остановиться на первом же исключении (даже если еще не обработаны все элементы), можно воспользоваться `std::promise` для установки как значения, так и исключения. В то же время, если нужно разрешить другим рабочим потокам продолжать поиск, можно воспользоваться `std::packaged_task`, сохранить все исключения, а затем заново выдать одно из них, если соответствия не найдено.

В данном случае я предпочел воспользоваться `std::promise` из-за того, что поведение этого варианта точнее соответствует поведению `std::find`. Особое внимание здесь следует уделить отсутствию искомого элемента в указанном диапазоне. Поэтому, чтобы получить результат из фьючерса, нужно дождаться окончания работы всех потоков. Если значения нет, то блокировка фьючерса приведет к бесконечному ожиданию. Результат показан в листинге 8.9.

Листинг 8.9. Реализация параллельного алгоритма find

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    struct find_element   ←❶
    {
        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)   ←❷
                {
```

```
                    if(*begin==match)
                    {
                        result->set_value(begin);    ❸
                        done_flag->store(true);      ❹
                        return;
                    }
                }
            }
            catch(...)    ❺
            {
                try
                {
                    result->set_exception(std::current_exception());  ❻
                    done_flag->store(true);
                }
                catch(...)    ❼
                {}
            }
        }
    };
    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;                          ❽
    std::atomic<bool> done_flag(false);                     ❾
    std::vector<std::thread> threads(num_threads-1);
    {                                                       ❿
        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(),          ⓫
                                   block_start,block_end,match,
                                   &result,&done_flag);
            block_start=block_end;
        }
        find_element()(block_start,last,match,&result,&done_flag);  ⓬
    }
    if(!done_flag.load())                                   ⓭
    {
        return last;
    }
    return result.get_future().get();                       ⓮
}
```

Основная часть кода листинга 8.9 похожа на код предыдущих примеров. На этот раз работа выполняется в операторе вызова функции локального класса `find_element` ❶. Здесь происходит последовательный перебор всех элементов переданного блока с проверкой флага при каждом шаге ❷. Если соответствие найдено, окончательное значение результата устанавливается в промисе ❸, а затем перед возвратом управления устанавливается флаг `done_flag` ❹.

Если выдано исключение, оно перехватывается общим обработчиком ❺, и перед установкой флага `done_flag` предпринимается попытка сохранить исключение в промисе ❻. Если значение промиса уже установлено, то попытка его повторной установки может привести к выдаче исключения, поэтому здесь перехватываются и сбрасываются все исключения, выдаваемые на данном этапе ❼.

Это означает: если поток, вызывающий `find_element`, либо найдет соответствие, либо выдаст исключение, все другие потоки увидят флаг `done_flag` и остановятся. Если в одно и то же время найдут соответствие или выдадут исключение сразу несколько потоков, возникнет состояние гонки за установкой значения в промисе. Но оно опасности не представляет: самый удачливый станет номинально первым, а следовательно, будет получен вполне приемлемый результат.

Если вернуться к основной функции `parallel_find` как таковой, в ней есть промис ❽ и флаг ❾, используемый для остановки поиска, передаваемые в новый поток наряду с диапазоном поиска ❶❶. Основной поток задействует также `find_element` для поиска в оставшихся элементах ❶❷. Как уже упоминалось, прежде чем проверять результат, нужно дождаться завершения работы всех потоков, поскольку соответствующих элементов в диапазоне может и не быть. Для этого код запуска и объединения потоков заключается в блок ❶❿, чтобы к моменту проверки флага на предмет найденного соответствия все потоки уже были объединены ❶❸. Если соответствие найдено, можно получить результат или выдать сохраненное исключение, вызвав функцию `get()` в отношении объекта `std::future<Iterator>`, который возможно получить из промиса ❶❹.

В этой реализации также предполагается, что задействованы все доступные аппаратные потоки или же есть иные механизмы для определения числа потоков, которыми можно воспользоваться, позволяющие заранее распределить работу между потоками. Как и прежде, для упрощения реализации можно воспользоваться автоматическим масштабированием стандартной библиотеки C++ и задействовать `std::async` и разделение данных в рекурсивном режиме. Реализация `parallel_find` с применением `std::async` показана в листинге 8.10.

Листинг 8.10. Реализация параллельного алгоритма find с использованием std::async

```
❶→ template<typename Iterator,typename MatchType>
   Iterator parallel_find_impl(Iterator first,Iterator last,MatchType match,
                               std::atomic<bool>& done)
   {
       try
       {
           unsigned long const length=std::distance(first,last);
     ❷→    unsigned long const min_per_thread=25;
           if(length<(2*min_per_thread))                    ←❸
           {
```

```
            for(;(first!=last) && !done.load();++first)   ←❹
            {
                if(*first==match)
                {
                    done=true;      ←❺
                    return first;
                }
            }
            return last;    ←❻
        }
        else
        {
            Iterator const mid_point=first+(length/2);   ←❼
            std::future<Iterator> async_result=
                std::async(&parallel_find_impl<Iterator,MatchType>,   ←❽
                        mid_point,last,match,std::ref(done));
            Iterator const direct_result=
                    parallel_find_impl(first,mid_point,match,done);   ←❾
            return (direct_result==mid_point)?
                async_result.get():direct_result;   ←❿
        }
    }
    catch(...)
    {
        done=true;   ←⓫
        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);   ←⓬
}
```

Желание завершить поиск на ранней стадии, как только будет найдено соответствие, подразумевает введение флага, совместно используемого всеми потоками и показывающего, что соответствие найдено. При этом возникает необходимость передать флаг всем рекурсивным вызовам. Проще всего выполнить эту задачу, делегировав ее функции реализации ❶, получающей дополнительный параметр — ссылку на флаг `done`, которая передается из главной точки входа ⓬.

После этого основная реализация идет уже знакомым путем. Как и во многих показанных здесь реализациях, одному потоку назначается минимальное число обрабатываемых элементов ❷, и, если разделения на две равные половины не получается, все запускается в текущем потоке ❸. Алгоритм представляет собой простой цикл последовательного прохода через указанный диапазон, не прекращающийся до достижения конца диапазона или установки флага `done` ❹. Если соответствие найдено, то перед возвратом управления устанавливается флаг `done` ❺. Если поиск остановлен либо по достижении конца списка, либо потому, что другой поток установил флаг `done`, возвращается значение `last`, чтобы показать, что соответствие найдено не было ❻.

Если диапазон можно разделить, то сначала, до использования `std::async` для запуска поиска во второй половине диапазона ❽, находится средняя точка ❼, при этом нужно не забыть воспользоваться `std::ref` для передачи ссылки на флаг `done`. В то же время за счет прямого рекурсивного вызова можно выполнить поиск в первой половине диапазона ❾. Как асинхронный вызов, так и прямая рекурсия могут привести к дальнейшему разделению исходного диапазона, если он довольно большого размера.

Если непосредственный поиск вернул среднюю точку `mid_point`, значит, он не смог найти соответствие, поэтому нужно получить результат асинхронного поиска. Если результат не был найден и в этой половине, этим результатом станет `last`, превращаясь во вполне приемлемое возвращаемое значение, показывающее, что искомое значение найдено не было ❿. Если якобы асинхронный вызов на самом деле был не асинхронным, а отложенным, то он будет запущен здесь в вызове функции `get()`. В этих обстоятельствах, если поиск в нижней половине диапазона проведен успешно, поиск в верхней половине диапазона пропускается. Если асинхронный поиск выполняется в другом потоке, то деструктор переменной `async_result` ожидает завершения потока, поэтому никакой утечки потоков не будет.

Как и прежде, применение `std::async` обеспечивает безопасность выдачи и возможность распространения исключений. Если непосредственная рекурсия выдает исключение, деструктор фьючерса обеспечит завершение работы потока, запустившего асинхронный вызов, до возвращения функцией управления, и если асинхронный вызов выдаст исключение, оно будет распространено посредством вызова функции `get()` ❿. Использование вокруг всего этого блока `try-catch` нужно только для того, чтобы установить флаг `done` в случае выдачи исключения и обеспечить в связи с этим быстрое завершение работы всех потоков ⓫. Данная реализация правильно работала бы и без этого, но продолжала бы проверку элементов до завершения работы всех потоков.

Главная особенность обеих реализаций этого алгоритма, характерная и для других рассмотренных параллельных алгоритмов, — то, что теперь нет никакой гарантии, что элементы обрабатываются в той же последовательности, которая получалась при использовании `std::find`. Это важно учесть, приступая к распараллеливанию алгоритма. Одновременная обработка элементов невозможна, если порядок их обработки имеет какое-то значение. Если элементы можно обрабатывать независимо друг от друга, то порядок для таких алгоритмов, как `parallel_for_each`, неважен. Но ситуация складывается иначе, когда алгоритм `parallel_find` возвращает элемент, расположенный ближе к концу диапазона, даже если есть соответствие ближе к его началу, что может стать для вас полной неожиданностью.

Итак, мы смогли распараллелить `std::find`. Как говорилось в начале раздела, есть и другие подобные алгоритмы, работа которых может завершиться без обработки каждого элемента данных, и для них можно использовать точно такие же приемы. Вопрос прерывания потоков будет рассматриваться и в главе 9.

В последнем примере из трех намеченных мы пойдем в обратном направлении и рассмотрим алгоритм `std::partial_sum`. Он не такой распространенный, как предыдущие, но представляет интерес для распараллеливания и позволяет обратить внимание на выбор нескольких дополнительных конструкторских решений.

8.5.3. Параллельная реализация std::partial_sum

Алгоритм `std::partial_sum` позволяет вычислить в диапазоне промежуточные суммы с заменой каждого элемента суммой его самого и всех предшествующих элементов в исходной последовательности. То есть последовательность 1, 2, 3, 4, 5 превращается в 1, (1 + 2) = 3, (1 + 2 + 3) = 6, (1 + 2 + 3 + 4) = 10, (1 + 2 + 3 + 4 + 5) = 15. Этот алгоритм интересен для распараллеливания, поскольку здесь нельзя просто разбить диапазон на блоки и вычислить каждый блок отдельно от других. Ведь исходное значение первого элемента нужно сложить с каждым из остальных элементов.

Один из подходов к вычислению промежуточных сумм диапазона предусматривает определение промежуточных сумм отдельных блоков и последующее прибавление получившегося значения последнего элемента первого блока ко всем элементам следующего блока и т. д. Если имеющиеся элементы 1, 2, 3, 4, 5, 6, 7, 8, 9 разбить на три блока, то сначала получится {1, 3, 6}, {4, 9, 15}, {7, 15, 24}. Если затем прибавить 6 (сумму в последнем элементе первого блока) к элементам второго блока, получится {1, 3, 6}, {10, 15, 21}, {7, 15, 24}. Затем последний элемент второго блока (21) прибавить к каждому элементу третьего, последнего блока, и получится окончательный результат: {1, 3, 6}, {10, 15, 21}, {28, 36, 55}.

Распараллелить можно не только исходное разбиение на блоки, но и прибавление промежуточной суммы из предыдущего блока. Если последний элемент каждого блока обновляется первым, то остальные элементы в блоке могут обновляться одним потоком, при этом второй поток может обновлять следующий блок и т. д. Такой алгоритм работает вполне приемлемо, если элементов в списке намного больше, чем вычислительных ядер в системе, потому что тогда у каждого ядра на каждой стадии будет подходящее число обрабатываемых элементов.

Если же вычислительных ядер очень много (столько же, сколько элементов, или даже больше), эффективной работы не получится. Если распределить работу между процессорами, то на первом этапе каждый из них будет работать всего с двумя элементами. При таких условиях предполагаемое дальнейшее распространение результатов будет означать, что многим процессорам придется ждать и их потребуется загрузить какой-либо работой. Тогда к решению проблемы можно подойти иначе. Вместо полного прямого распространения сумм от одного блока к другому выполняется частичное распространение: сначала, как и раньше, суммируются смежные элементы, но затем эти суммы складываются со значениями двух следующих элементов, после чего следующий набор результатов складывается со значениями следующих четырех элементов и т. д. Если приступить к вычислениям с теми же исходными девятью элементами, то после первого этапа будет получен список 1, 3, 5, 7, 9, 11, 13, 15, 17, в котором отражены правильные результаты для первых двух элементов. После второго этапа получится список 1, 3, 6, 10, 14, 18, 22, 26, 30, содержащий правильные результаты для первых четырех элементов. После третьего этапа получится список 1, 3, 6, 10, 15, 21, 28, 36, 44, правильный для первых восьми элементов. И наконец, после четвертого этапа будет получен окончательный ответ 1, 3, 6, 10, 15, 21, 28, 36, 45. Хотя в целом количество этапов по сравнению с первым вариантом увеличилось, возможности для распараллеливания при наличии множества процессоров расширились: каждый процессор на каждом этапе может обновить одну запись.

Второй подход состоит из log2(N) этапов приблизительно из N операций (по одной на каждый процессор), где N — число элементов в списке. Для сравнения: в первом алгоритме каждому потоку приходилось выполнять N/k операций для исходной промежуточной суммы распределенного ему блока, а затем следующие N/k операций для последующего распространения, где k — число потоков. Если оценивать общее количество операций, то при первом подходе получается $O(N)$, а при втором — $O(N\log(N))$. Но если процессоров столько же, сколько элементов в списке, то при втором подходе потребуется всего $\log(N)$ операция на каждый процессор, а при первом, когда число k увеличивается, операции из-за прямого распространения выполняются последовательно. Для небольшого количества вычислительных блоков при первом подходе работа завершится быстрее, но в гораздо более крупных параллельных системах она будет выполнена быстрее при втором подходе. Это яркий пример, иллюстрирующий вопросы, рассмотренные в подразделе 8.2.1.

Оставим эффективность в покое и рассмотрим сам код. В листинге 8.11 показана реализация первого подхода.

Листинг 8.11. Вычисление промежуточных сумм в параллельном режиме с разделением задачи

```
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)
 (3) -->        if(previous_end_value)
                {
                    value_type& addend=previous_end_value->get();  <-- (4)
 (5) -->            *last+=addend;
                    if(end_value)
                    {
                        end_value->set_value(*last);   <-- (6)
                    }
 (7) -->            std::for_each(begin,last,[addend](value_type& item)
                                  {
                                      item+=addend;
                                  });
                }
                else if(end_value)
                {
                    end_value->set_value(*last);  <-- (8)
                }
            }
```

```
            catch(...)      ←❾
            {
                if(end_value)
                {
          ❿→        end_value->set_exception(std::current_exception());
                }
                else
                {
                    throw;   ←⓫
                }
            }
        }
    };
    unsigned long const length=std::distance(first,last);
    if(!length)
        return;
    unsigned long const min_per_thread=25;     ←⓬
    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);   ←⓭
    std::vector<std::promise<value_type> >
        end_values(num_threads-1);              ←⓮
    std::vector<std::future<value_type> >
⓯→      previous_end_values;
    previous_end_values.reserve(num_threads-1);   ←⓰
    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);     ←⓱
⓲→      threads[i]=std::thread(process_chunk(),
                               block_start,block_last,
                               (i!=0)?&previous_end_values[i-1]:0,
                               &end_values[i]);
        block_start=block_last;
⓳→      ++block_start;
        previous_end_values.push_back(end_values[i].get_future());  ←⓴
    }
    Iterator final_element=block_start;
    std::advance(final_element,std::distance(block_start,last)-1);   ←㉑
㉒→  process_chunk()(block_start,final_element,
                     (num_threads>1)?&previous_end_values.back():0, 0);
}
```

В данном случае общая структура такая же, как и у предыдущих алгоритмов, — разделение задачи на два блока с указанием их минимального размера для каждого потока ⓬. Наряду с вектором потоков ⓭ есть вектор промисов ⓮, используемый для

хранения значения последнего элемента блока, и вектор фьючерсов ⓯, с помощью которого получается последнее значение из предыдущего блока. Поскольку число необходимых фьючерсов известно, место для них можно зарезервировать ⓰, избегая тем самым перераспределения памяти при порождении потоков.

Основной цикл выглядит почти так же, но на этот раз в нем нужен итератор, *указывающий на* последний элемент каждого блока, а не, как обычно, на тот, что следует за последним ⓱, предоставляя возможность прямого распространения последнего элемента каждого диапазона. Обработка выполняется в рассматриваемом далее функциональном объекте `process_chunk`, в качестве аргументов передаются начальный и конечный итераторы блока, а также фьючерс для конечного значения предыдущего диапазона (если таковой имеется) и промис для хранения конечного значения этого диапазона ⓲.

После порождения потока можно обновить начало блока, не забыв продвинуть его за последний элемент ⓳, и сохранить фьючерс для последнего значения в текущем блоке в векторе фьючерсов, чтобы можно было забрать его при следующем проходе цикла ⓴.

Перед обработкой последнего блока следует получить итератор для последнего элемента ㉑, который можно передать в `process_chunk` ㉒. Алгоритм `std::partial_sum` не возвращает значение, поэтому после обработки последнего блока больше ничего делать не надо. Операция завершится, как только закончат работу все потоки.

Итак, настало время рассмотреть функциональный объект `process_chunk`, выполняющий всю работу ❶. Сначала вызывается `std::partial_sum` для всего блока, включая последний элемент ❷, но затем потребуется узнать, первый это блок или нет ❸. Если это не первый блок, значит, был указан параметр `previous_end_value` и его нужно дождаться ❹. Затем, чтобы максимально распараллелить алгоритм, первым обновляется последний элемент ❺, чтобы можно было передать значение следующему блоку (если таковой имеется) ❻. Как только это будет сделано, можно будет воспользоваться `std::for_each` и простой лямбда-функцией, чтобы обновить все оставшиеся элементы диапазона ❼.

Если параметра `previous_end_value` *не было*, значит, это первый блок и можно обновить значение `end_value` для следующего блока (опять-таки если он существует, можно ведь иметь дело только с одним блоком) ❽.

И наконец, если какая-то операция выдаст исключение, оно перехватывается ❾ и сохраняется в промисе ❿, поэтому будет распространено на следующий блок, когда он попытается получить предыдущее конечное значение ❹. Таким образом все исключения дойдут до последнего блока, где и будет заново выдано исключение ⓫, поскольку известно, что он обрабатывается в основном потоке.

Из-за синхронизации между потоками этот код нельзя будет полностью переделать для использования `std::async`. Задачи ждут результатов, которые становятся доступными в процессе выполнения других задач, поэтому все задачи должны выполняться одновременно.

Теперь оставим в стороне подход, основанный на блоках и прямом распространении результатов, и рассмотрим второй подход к вычислению промежуточных сумм диапазона.

Реализация инкрементного попарного алгоритма вычисления промежуточных сумм

Второй подход к вычислению промежуточных сумм, основанный на сложении элементов, удаленных друг от друга на все большее и большее расстояние, хорошо работает, когда процессоры могут выполнять сложение в слаженном ритме. В таком случае никакой дальнейшей синхронизации не требуется, поскольку все промежуточные результаты могут непосредственно распространяться на следующий нуждающийся в них процессор. На практике работать с такими системами приходится довольно редко, за исключением тех случаев, когда один процессор может выполнять одну и ту же команду над небольшим числом элементов данных одновременно, применяя так называемые SIMD-команды (Single-Instruction/Multiple-Data — одиночный поток команд, множественный поток данных). Поэтому нужно разрабатывать код для общего случая и на каждом этапе синхронизировать потоки явным образом.

Один из способов решения этой задачи заключается в использовании *барьера* — механизма синхронизации, заставляющего потоки ждать, пока нужное количество потоков не достигнет его. Как только до барьера дойдут все потоки, они будут разблокированы и смогут продолжать выполнение. В C++11 Thread Library такое средство напрямую не предлагается, поэтому его следует разработать самостоятельно.

Представьте себе американские горки в парке аттракционов. При достаточном количестве ожидающих персонал сначала убедится, что все места заполнены, и только после этого запустит аттракцион. Барьер работает точно так же: заранее указывается число мест и потоки должны ждать, пока все они будут заполнены. При достаточном количестве ожидающих потоков все они могут продолжить выполнение. Барьер перезапускается и ожидает следующей партии потоков. Зачастую такая конструкция используется в цикле, где одни и те же потоки проходят цикл и ждут следующего раза. По идее, нужно поддерживать слаженность работы всех потоков, чтобы ни один из них не обгонял других и не сбивался с ритма. Сбой приводит подобные алгоритмы к полному краху, поскольку забежавший вперед поток сможет изменить данные, все еще применяемые другими потоками, или же воспользоваться данными, еще должным образом не обновленными.

Простая реализация барьера показана в листинге 8.12.

Листинг 8.12. Простой класс барьера

```
class barrier
{
    unsigned const count;
    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
public:
    explicit barrier(unsigned count_):          ←❶
        count(count_),spaces(count),generation(0)
    {}
    void wait()
    {
        unsigned const my_generation=generation;  ←❷
```

```
❸ ──► if(!--spaces)
      {
          spaces=count;    ◄──❹
          ++generation;    ◄─┐
      }                      ❺
      else
      {
          while(generation==my_generation)  ◄──❻
              std::this_thread::yield();    ◄─┐
      }                                       ❼
  }
};
```

В этой реализации барьер создается с количеством мест ❶, которое сохраняется в переменной `count`. Изначально количество свободных мест `spaces` у барьера равно `count`. По мере перехода очередного потока в режим ожидания количество свободных мест в `spaces` уменьшается на единицу ❸. Когда оно достигнет нуля, количество `spaces` возвращается к значению `count` ❹, а значение переменной `generation` увеличивается, чтобы сигнализировать другим потокам: они могут продолжить выполнение ❺. Если число свободных мест не достигло нуля, потоку приходится ждать. В этой реализации используется простая спин-блокировка ❻, сравнивающая значение `generation` со значением, полученным в начале `wait()` ❷. Поскольку значение `generation` обновляется только после достижения барьера всеми потоками ❺, на время ожидания запускается функция `yield()`, позволяющая уступить вычислительный ресурс другим потокам ❼, поэтому ожидающий поток не заставляет центральный процессор находиться в режиме активного ожидания.

Говоря, что это очень простая реализация, я имел в виду следующее: здесь используется ожидание на основе спин-блокировки, поэтому она неидеальна — потокам приходится долго ожидать — и не работает, когда число потоков, потенциально способных в любое время вызвать функцию `wait()`, больше, чем значение переменной `count`. Когда нужно будет реализовать любой из этих сценариев, придется воспользоваться более надежной, но и более сложной реализацией. Я также придерживался применения последовательно согласованных операций над атомарными переменными — так было проще все объяснить. Но вы можете ослабить некоторые ограничения, накладываемые на порядок обращений к памяти. На крупных параллельных архитектурах такая глобальная синхронизация обходится довольно дорого, поскольку строки кэша, содержащие состояние барьера, должны курсировать между всеми вовлеченными в работу процессорами (вспомните рассуждения насчет перебрасывания данных в кэш-памяти в подразделе 8.2.2). Поэтому нужно быть абсолютно уверенными в том, что в конкретном случае это самый лучший вариант. Если используемая вами стандартная библиотека C++ предоставляет средства, входящие в спецификацию Concurrency TS, то здесь можно воспользоваться классом `std::experimental::barrier`. Подробности изложены в главе 4.

Итак, у вас есть все, что нужно, а именно фиксированное число потоков, которые нужно запускать в цикле в слаженном ритме. Но это *почти* фиксированное число потоков. Как вы, наверное, помните, элементы в начале списка получают окончательные значения после двух этапов вычисления. Значит, либо эти потоки вынуждены

будут находиться в цикле, пока не будет обработан весь диапазон, либо нужно, чтобы барьер обрабатывал выпадающие из работы потоки и уменьшал значение `count`. Я выбрал второй вариант: так потоки не занимаются ненужной работой, находясь в цикле до завершения финального этапа.

Поэтому `count` нужно сделать атомарной переменной, чтобы ее можно было обновлять из нескольких потоков без внешней синхронизации:

```
std::atomic<unsigned> count;
```

Инициализация осталась прежней, но теперь при перезапуске числа свободных мест `spaces` из `count` необходим явный вызов функции `load()`:

```
spaces=count.load();
```

Это все, что нужно было изменить в отношении `wait()`, но теперь требуется новая компонентная функция для декремента `count`. Назовем ее `done_waiting()`, поскольку поток объявляет об окончании ожидания:

```
void done_waiting()
{
    --count;                    ❶
    if(!--spaces)               ❷
    {
        spaces=count.load();    ❸
        ++generation;
    }
}
```

Сначала нужно уменьшить на единицу значение `count` ❶, чтобы при следующей переустановке значения `spaces` в этой переменной отображалось новое, более низкое число ожидающих потоков. Затем нужно уменьшить число свободных мест `spaces` ❷. Если этого не сделать, другие потоки будут ждать до бесконечности, поскольку значение `spaces` было инициализировано старым, более высоким значением. Если текущий поток — последний в данной партии, следует переустановить значение `counter` и увеличить на единицу значение `generation` ❸, как это делается в `wait()`. Основное отличие здесь в том, что если текущий поток — последний в партии, ждать уже не нужно.

Теперь все готово к созданию второй реализации алгоритма вычисления промежуточных сумм. На каждом этапе каждый поток вызывает функцию `wait()` в отношении барьера, чтобы гарантировать, что все потоки проходят его вместе, и как только каждый поток выполнит свою работу, он вызывает в отношении барьера функцию `done_waiting()` для уменьшения значения `count` на единицу. Если вместе с исходным диапазоном используется второй буфер, необходимая синхронизация обеспечивается барьером. На каждом этапе потоки либо считывают данные из исходного диапазона и записывают новое значение в соответствующий элемент буфера, либо, наоборот, выполняют чтение из буфера, а новое значение записывают в соответствующий элемент исходного диапазона. Если на одном этапе потоки считывали данные из исходного диапазона, то на следующем они ведут чтение из буфера и наоборот. Тем самым гарантируется отсутствие состояния гонки за данными между операциями чтения и записи в разных потоках. Как только поток завершит цикл,

он должен убедиться, что правильное окончательное значение было записано в исходный диапазон. Все это, собранное воедино, показано в листинге 8.13.

Листинг 8.13. Параллельная реализация partial_sum, выполненная с использованием 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;
❶   struct process_element
    {
        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)?    ←❷
                    buffer[i]:ith_element;
                value_type& dest=(step%2)?
                    ith_element:buffer[i];
                value_type const& addend=(step%2)?     ←❸
                    buffer[i-stride]:*(first+i-stride);
        ❹→      dest=source+addend;
                update_source=!(step%2);
                b.wait();                ←❺
            }
            if(update_source)          ←❻
            {
                ith_element=buffer[i];
            }
            b.done_waiting();        ←❼
        }
    };
    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);  ←❽
    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, ←❾
                               std::ref(buffer),i,std::ref(b));
    }
    process_element()(first,last,buffer,length-1,b);  ←❿
}
```

Общая структура этого кода, наверное, уже знакома. Имеется класс с оператором вызова функции (`process_element`), предназначенной для выполнения работы ❶, которая вызывается из партии потоков ❾, хранящейся в векторе ❽, которая также вызывается из основного потока ❿. Теперь основное отличие заключается в том, что число потоков зависит от количества элементов в списке, а не от значения, возвращенного функцией `std::thread::hardware_concurrency`. Как уже говорилось, если работа ведется не на крупной параллельной системе с низкими издержками на применение множества потоков, удачной эту идею не назовешь, но с ее помощью демонстрируется общая структура. Можно было бы справиться с задачей и меньшим числом потоков, заставив каждый из них обрабатывать несколько значений из исходного диапазона, но так можно дойти до того, что потоков станет слишком мало и их использование будет менее эффективным, чем в алгоритме с прямым распространением результатов.

Основная работа выполняется в операторе вызова функции `process_element`. На каждом этапе берется либо *i*-й элемент из исходного диапазона, либо *i*-й элемент из буфера ❷ и складывается со значением предшествующего элемента, отстоящего от него на `stride` элементов ❸, с сохранением результата в буфере, если начальное

значение бралось из исходного диапазона, или в исходном диапазоне, если начальное значение бралось из буфера ❹. Затем перед запуском нового этапа наступает период ожидания у барьера ❺. Работа завершается, когда значение `stride` выходит за границу начала диапазона. В этом случае необходимо обновить элемент в исходном диапазоне, если окончательный результат был сохранен в буфере ❻. И наконец, барьеру с помощью вызова функции `done_waiting()` сообщается, что ожидание закончилось ❼.

Следует отметить, что данное решение не предусматривает безопасную выдачу исключений. Если исключение выдается в функции `process_element` в одном из рабочих потоков, работа приложения прекращается. Исправить положение можно за счет сохранения исключения с помощью `std::promise`, как это делалось в реализации `parallel_find` из листинга 8.9, или даже за счет использования объекта `std::exception_ptr`, защищенного мьютексом.

На этом рассмотрение трех примеров завершается. Надеюсь, они помогли разобраться в некоторых вопросах разработки, поднятых в разделах 8.1–8.4, и показали, как описанные приемы воплощаются в реальном коде.

Резюме

В этой главе мы рассмотрели много вопросов. Сначала разобрали различные способы распределения работы между потоками, включая предварительное распределение данных или использование нескольких потоков для создания конвейера. Затем обсудили низкоуровневые аспекты достижения высокой производительности многопоточного кода, в том числе ложное совместное использование памяти и конкуренцию при обращении к данным. Далее мы перешли к вопросу о том, как порядок обращения к данным влияет на производительность программного кода. После этого переключились на исследование дополнительных факторов, учитываемых при разработке конкурентных программ, включая безопасность выдачи исключений и масштабируемость. И в завершение разобрали несколько примеров реализации параллельных алгоритмов, в каждом из которых освещены конкретные проблемы, возникающие при разработке многопоточного кода.

Несколько раз в этой главе упоминалась идея создания пула потоков — заранее сконфигурированной группы потоков, выполняющей задачи, назначенные пулу. Разработка удачного пула потоков далеко не простое дело, поэтому в следующей главе мы рассмотрим возможные проблемы, а также различные особенности управления потоками.

9 Усовершенствованное управление потоками

В этой главе

- Пулы потоков.
- Управление зависимостями между задачами пула.
- Перехват работы в пуле потоков.
- Прерывание потоков.

В предыдущих главах все управление потоками сводилось к явному созданию объектов `std::thread` для каждого из них. В нескольких местах было видно, насколько это может быть нежелательно, так как вынуждало управлять продолжительностью жизни объектов потоков, определять число потоков, соответствующее объему решаемой задачи и используемому оборудованию, и т. д. Идеальным был бы сценарий, при котором имелась бы возможность разделить код на самые мелкие части, которые можно было бы выполнять одновременно, передать их компилятору и библиотеке и сказать: «Распараллельте это для достижения оптимальной производительности». В главе 10 будет показано, что бывают случаи, позволяющие именно так и сделать: если код, требующий распараллеливания, может быть выражен в виде вызова стандартного библиотечного алгоритма, то в большинстве случаев можно будет запросить у библиотеки распараллеливание.

В нескольких примерах повторялась еще одна тема — использования для решения поставленной задачи нескольких потоков с возможностью их более раннего завершения при определенных условиях. Причиной может быть определение результата, или

ошибка, или даже явный запрос пользователя на прерывание операции. Какой бы она ни была, потокам нужно отправить запрос: «Пожалуйста, остановитесь», чтобы они могли отказаться от полученного задания, прибрать за собой и как можно быстрее завершиться.

В этой главе мы рассмотрим приемы управления потоками и задачами и вначале поговорим об автоматическом управлении потоками и распределении задач между ними.

9.1. Пулы потоков

Сотрудникам многих компаний, обычно работающим в офисе, время от времени нужно выезжать к клиентам или поставщикам либо присутствовать на выставке или конференции. Поездки могут быть обязательными, и в какой-нибудь день в разъездах могут быть сразу несколько сотрудников, но для любого конкретного сотрудника перерывы между командировками могут составлять месяцы или даже годы. Поскольку компании не могут выделить в распоряжение каждого сотрудника служебный автомобиль, они зачастую предоставляют пул машин, доступный всему персоналу. Когда сотруднику нужно уехать по делам, он заказывает на определенное время одну из машин пула, а возвращаясь в офис, предоставляет возможность коллегам взять ее. Если в какой-то день свободных машин не окажется, придется перенести поездку на другую дату.

Пул потоков построен по схожему принципу, за исключением того, что совместно применяются *потоки*, а не машины. В большинстве систем нецелесообразно использовать отдельный поток для каждой задачи, когда можно выполнять ее параллельно с другими, но при этом нужно стремиться задействовать доступную конкурентность там, где это возможно. Именно это и позволяет вам создать пул потоков: задачи, которые могут выполняться одновременно, передаются пулу, который помещает их в очередь ожидающих работ. Затем *рабочие потоки* поочередно берут из очереди по одной задаче, выполняют ее и возвращаются в цикл за следующей задачей.

При создании пула потоков решается несколько основных проблем, касающихся его конструкции, например: сколько потоков использовать, каким будет наиболее эффективный способ распределения задач по потокам и нужно ли ожидать завершения задачи. В этом разделе рассмотрим несколько реализаций пула потоков, начиная с самого простого, в которых дается ответ на поставленные вопросы.

9.1.1. Простейший пул потоков

В самом простом виде пул потоков представляет собой фиксированное число *рабочих потоков*, обычно совпадающее со значением, возвращаемым функцией `std::thread::hardware_concurrency()`. Когда появляется работа, вызывается функция, помещающая ее в очередь ожидающих работ. Каждый рабочий поток берет работу из очереди, выполняет поставленную в ней задачу и обращается к очереди за другой работой. В простейшем случае возможность дождаться завершения задачи не предусматривается. Если она нужна, то синхронизацией придется заняться самостоятельно. Пример реализации такого пула потоков показан в листинге 9.1.

Листинг 9.1. Простой пул потоков

```
class thread_pool
{
    std::atomic_bool done;
    threadsafe_queue<std::function<void()> > work_queue;  ❶
❷   std::vector<std::thread> threads;
    join_threads joiner;                                  ❸
    void worker_thread()
    {
        while(!done)                                      ❹
        {
            std::function<void()> task;
            if(work_queue.try_pop(task))                  ❺
            {
                task();                                   ❻
            }
            else
            {
                std::this_thread::yield();                ❼
            }
        }
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency(); ❽
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this));  ❾
            }
        }
        catch(...)
        {
            done=true;                                    ❿
            throw;
        }
    }
    ~thread_pool()
    {
        done=true;                                        ⓫
    }
    template<typename FunctionType>
    void submit(FunctionType f)
    {
        work_queue.push(std::function<void()>(f));        ⓬
    }
};
```

В этой реализации есть вектор рабочих потоков ❷, а для управления очередью работ задействуется одна из потокобезопасных очередей из главы 6 ❶. В данном случае пользователи не могут дожидаться завершения задач и возвращать какие-либо

значения, поэтому для инкапсуляции задач можно воспользоваться объектом `std::function<void()>`. Затем функция `submit()` заключает предоставленную ей функцию или вызываемый объект в экземпляр `std::function<void()>` и помещает его в очередь ⓬.

Потоки формируются в конструкторе: из вызова функции `std::thread::hardware_concurrency()` берется число поддерживаемых оборудованием конкурентных потоков ❽ и создается именно столько потоков, выполняющих компонентную функцию `worker_thread()` ❾.

Запуск потока может провалиться из-за выдачи исключения, и на этот случай нужно обеспечить остановку всех уже запущенных потоков и ликвидацию последствий их запуска. Задача решается с помощью блока `try-catch`, который при выдаче исключения устанавливает флаг `done` ❿. Кроме того, для объединения всех потоков используется экземпляр класса `join_threads` из главы 8 ❸. Этот же прием работает с деструктором: можно установить флаг `done` ⓫, а экземпляр класса `join_threads` обеспечит завершение работы всех потоков, прежде чем пул будет уничтожен. Следует отметить важность порядка объявления компонентов: и флаг `done`, и очередь `worker_queue` должны быть объявлены до вектора потоков `threads`, который, в свою очередь, должен быть объявлен до `joiner`. Тем самым будет гарантировано удаление компонентов в правильном порядке: к примеру, невозможно безопасно удалить очередь, пока не будут остановлены все потоки.

В самой функции `worker_thread` нет ничего сложного: она находится в цикле, ожидая установки флага `done` ❹, извлекая между ожиданиями задачи из очереди ❺ и выполняя их ❻. Если задач в очереди нет, функция вызывает `std::this_thread::yield()`, чтобы взять небольшой перерыв ❼ и дать другому потоку возможность поместить какую-нибудь работу в очередь, пока он через некоторое время не возобновит попытки извлечения задачи.

Даже такого простого пула потоков достаточно будет для многих задач, особенно если они полностью независимы друг от друга, не возвращают никаких значений и не выполняют блокирующих операций. Но немало и таких обстоятельств, при которых простой пул потоков не сможет адекватно соответствовать потребностям, и даже таких, при которых он способен вызвать проблемы, связанные с взаимной блокировкой. Кроме того, в простых случаях, возможно, стоит воспользоваться `std::async`, как в многочисленных примерах, приведенных в главе 8. В данной главе будут рассмотрены более сложные реализации пула потоков, имеющие дополнительные свойства, которые либо полнее соответствуют вашим потребностям, либо сужают круг возможных проблем. Начнем с ожидания завершения переданных пулу задач.

9.1.2. Ожидание завершения задач, переданных пулу потоков

В примерах в главе 8, где потоки порождались явным образом, основной поток после распределения работы между потоками всегда дожидался завершения работы только что порожденных потоков, чтобы перед возвращением управления вызывающему коду гарантировать завершение всей задачи. При использовании пула потоков нуж-

но дожидаться завершения не самих рабочих потоков, а переданных пулу потоков задач. Это похоже на ожидание фьючерсов в основанных на применении `std::async` примерах из главы 8. А в простом пуле потоков из листинга 9.1 ожидание придется задавать самостоятельно, задействуя механизмы из главы 4 — условные переменные и фьючерсы. Это усложняет код, и лучше было бы дожидаться завершения задач напрямую.

Непосредственного ожидания завершения задач можно добиться за счет переноса этих усложнений в пул потоков. Можно заставить функцию `submit()` возвращать описатель задачи с информацией, которой затем можно было бы воспользоваться для ожидания завершения задачи. В этот описатель было бы заключено использование условных переменных или фьючерсов, что упростило бы код, задействующий пул потоков.

Частный случай, требующий дождаться завершения порожденной задачи, возникает, когда основной поток нуждается в результате, вычисленном задачей. Он уже встречался в приводимых в этой книге примерах, например в функции `parallel_accumulate()` из главы 2. В данном случае с помощью фьючерсов можно объединить ожидание с передачей результата. В листинге 9.2 показаны изменения, которые нужно внести в код простого пула потоков, чтобы появилась возможность дождаться завершения задачи, а затем передать возвращаемые значения из задачи ожидающему потоку. Поскольку экземпляры класса `std::packaged_task<>` не допускают *копирования* и могут только *перемещаться*, класс `std::function<>` больше не может использоваться для элементов очереди, так как `std::function<>` требует, чтобы сохраненные функциональные объекты были копируемыми. Вместо него нужно воспользоваться специализированной функцией-оболочкой, способной обрабатывать типы, предназначенные только для перемещений. Это простой максирующий тип класс с оператором вызова функции. Нужны лишь функции обработки, не получающие параметров и возвращающие тип `void`, следовательно, в реализации это будет простой виртуальный вызов.

Листинг 9.2. Пул потоков, позволяющий дождаться завершения задач

```
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;
    void worker_thread()
    {
        while(!done)
        {
            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>
        submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type
            result_type;
        std::packaged_task<result_type()> task(std::move(f));
        std::future<result_type> res(task.get_future());
        work_queue.push(std::move(task));
        return res;
    }
    // все остальное не изменилось
};
```

Использование function_wrapper вместо std::function

❶ ❷ ❸ ❹ ❺ ❻

Сначала отметим, что измененная функция `submit()` ❶ возвращает объект `std::future<>` для хранения возвращаемого задачей значения и предоставления вызывающему коду возможности ждать завершения задачи. Для этого требуется знать возвращаемый тип предоставленной функции `f`, и здесь пригодится `std::result_of<>`: типом результата, возвращаемого вызовом экземпляра типа `FunctionType`

(например, `f`) без аргументов, будет `std::result_of<FunctionType()>::type`. То же самое выражение `std::result_of<>` используется для `result_type` `typedef` ❷ внутри функции.

Затем функция `f` помещается в оболочку `std::packaged_task<result_type()>` ❸, поскольку `f` является функцией или вызываемым объектом, не получающим параметры и возвращающим, как выяснилось, экземпляр типа `result_type`. Теперь можно получить фьючерс из `std::packaged_task<>` ❹ перед помещением задачи в очередь ❺ и возвращением фьючерса ❻. Следует отметить, что при помещении задачи в очередь необходимо задействовать функцию `std::move()`, поскольку `std::packaged_task<>` не допускает копирования. Теперь, чтобы можно было справиться с ситуацией, в очереди хранятся объекты `function_wrapper`, а не объекты `std::function<void()>`.

Этот пул позволяет дожидаться завершения задач и получать от них результаты. В листинге 9.3 показано, какой становится функция `parallel_accumulate` с этим пулом потоков.

Листинг 9.3. Функция parallel_accumulate, использующая пул потоков, позволяющий дожидаться завершения 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; ←❶
    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>()(block_start,block_end);
❷→      });
        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;
}
```

Если сравнить этот код с кодом листинга 8.4, можно заметить две особенности. Работа проводится в понятиях количества используемых блоков (`num_blocks`) ❶, а не количества потоков. Чтобы максимально воспользоваться возможностью

масштабирования пула потоков, нужно разбить работу на мельчайшие блоки, с которыми имеет смысл работать в режиме конкурентности. Если количество потоков в пуле невелико, каждый из них будет предназначен для обработки множества блоков, но по мере роста числа потоков, поддерживаемых оборудованием, будет расти и число блоков, обрабатываемых в параллельном режиме.

При выборе мельчайших блоков, с которыми имеет смысл работать в конкурентном режиме, нужно проявлять осмотрительность. Передача задачи пулу потоков, запуск ее в рабочем потоке и передача возвращаемого значения посредством `std::future<>` не обойдутся без издержек, и при мелких задачах они могут обесценить всю эту затею. Если выбрать слишком маленький размер задачи, то с пулом потоков код может работать медленнее, чем с одним потоком.

При правильном размере блока беспокоиться за упаковку задач, получение фьючерсов или сохранение объектов `std::thread`, чтобы впоследствии можно было объединить потоки, уже не придется, обо всем этом позаботится пул потоков. Останется лишь вызвать функцию `submit()`, передав ей задачу ❷.

Пул потоков обеспечит также безопасность выдачи исключений. Любое исключение, выданное задачей, будет распространено посредством фьючерса, возвращаемого из вызова функции `submit()`, и, если выход из функции не обходится без исключения, деструктор пула потоков закроет любые еще не оконченные задачи и дождется завершения потоков, относящихся к пулу.

Для простых случаев, когда задачи не зависят друг от друга, эта конструкция вполне работоспособна. Но для ситуаций, при которых задачи зависят от других задач, также передаваемых пулу потоков, она непригодна.

9.1.3. Задачи, ожидающие завершения других задач

Алгоритм Quicksort многократно приводился в этой книге в качестве примера. В принципе, в нем нет ничего сложного: сортируемые данные разбиваются на элементы, составляющие последовательности до опорного элемента и после него. Эти два набора элементов сортируются в рекурсивном режиме, после чего собираются в полностью отсортированный набор. При распараллеливании алгоритма необходимо, чтобы рекурсивные вызовы пользовались доступными возможностями конкурентности.

При первом представлении этого примера в главе 4 для запуска одного из рекурсивных вызовов на каждом этапе применялась функция `std::async`, что позволяло библиотеке выбирать между запуском этой функции в отношении нового потока и сортировкой в синхронном режиме при вызове соответствующей функции `get()`. Эта реализация работала неплохо, поскольку каждая задача либо запускалась в собственном потоке, либо вызывалась по мере необходимости.

При новом обращении к этой реализации в главе 8 была показана альтернативная структура, использующая фиксированное число потоков, соответствующее возможностям аппаратной конкурентности. В этом случае задействовался стек ожидающих сортировки блоков. Каждый поток, разбив на части сортируемые им данные, добавлял к стеку новый блок, относящийся к одному набору данных, а затем непосредственно сортировал другой набор данных. На этой стадии простое

ожидание завершения сортировки другого блока могло превратиться во взаимную блокировку, поскольку пришлось бы воспользоваться одним из ограниченного числа ожидающих потоков. Все это легко могло вылиться в ситуацию, при которой все потоки ожидали бы завершения сортировки блоков и ни один из них не занимался бы сортировкой. Проблему удалось решить, заставив потоки извлекать блоки из стека и выполнять их сортировку, пока конкретный блок, который они ожидали, был еще не отсортирован.

Если в пример из главы 4 вместо `std::async` подставить простой пул потоков вроде того, который до сих пор рассматривался в этой главе, возникнет точно такая же проблема. Теперь она связана с ограниченным числом потоков, и может оказаться, что все они будут ждать завершения задачи, которая не была запланирована из-за отсутствия свободных потоков. Поэтому следует воспользоваться решением, похожим на одно из использованных в главе 8: обработать остающиеся блоки, пока длится ожидание завершения сортировки нужного блока. Если пул потоков задействуется для управления списком задач и их связью с потоками — в конце концов, в этом и состоит смысл применения пула потоков, — то для этого нам не хватает доступа к списку задач. Нужно изменить пул потоков, чтобы все выполнялось автоматически.

Проще всего добавить к `thread_pool` новую функцию, чтобы запускать задачу из очереди и управлять циклом самостоятельно. Так мы и сделаем. Усовершенствованная реализация пула потоков могла бы включать логику в функцию ожидания или дополнительные функции ожидания, справляющиеся с данным случаем, возможно, за счет расстановки приоритетов задачам, завершения которых ожидаем. В листинге 9.4 показана новая функция `run_pending_task()`, а использующий ее измененный алгоритм Quicksort приведен 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();
    }
}
```

Эта реализация `run_pending_task()` полностью перенесена из основного цикла функции `worker_thread()`, которую теперь можно будет изменить, чтобы она вызывала извлеченную функцию `run_pending_task()`. Эта функция пытается извлечь задачу из очереди и выполнить ее, если она там есть, в противном случае уступает управление, позволяя операционной системе перепланировать поток. Реализация Quicksort, показанная в листинге 9.5, намного проще соответствующей версии из листинга 8.1, поскольку вся логика управления потоками была перенесена в пул потоков.

Листинг 9.5. Реализация Quicksort, основанная на использовании пула потоков

```
template<typename T>
struct sorter          <--❶
{
    thread_pool pool;       <--❷

    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);
  ❸--> std::future<std::list<T> > new_lower=
            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();   <--❹
        }
        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 делегирована компонентной функции `do_sort()`, принадлежащей шаблону класса `sorter` ❶, хотя в данном случае он нужен только для заключения в него экземпляра `thread_pool` ❷.

Теперь управление потоками и задачами свелось к отправке задачи в пул ❸ и запуску ожидающих выполнения задач в период ожидания ❹. Это намного проще, чем в листинге 8.1, где приходилось явным образом управлять потоками и стеком подлежащих сортировке блоков. При отправке задачи пулу используется функция

`std::bind()`, чтобы привязать указатель `this` к `do_sort()` и предоставить блок для сортировки. В данном случае, как только блок будет передан, в отношении `new_lower_chunk` вызывается функция `std::move()`, чтобы гарантировать, что данные перемещаются, а не копируются.

Теперь важная проблема, вызывавшая взаимную блокировку, связанную с задачами, ожидающими результатов выполнения других задач, решена, однако этот пул потоков еще далек от идеала. Начнем с того, что при каждом вызове `submit()` и `run_pending_task()` происходит обращение к одной и той же очереди. В главе 8 было показано, как наличие одного набора данных, изменяемого несколькими потоками, может ухудшить производительность, поэтому с данной проблемой нужно что-то делать.

9.1.4. Предотвращение конкуренции при обращении к очереди работ

При каждом вызове потоком функции `submit()` в отношении конкретного экземпляра пула потоков приходится помещать новый элемент в единственную совместно используемую очередь работ. Кроме того, рабочие потоки постоянно извлекают элементы из очереди, чтобы запускать задачи. Следовательно, по мере роста числа процессоров растет и конкуренция при обращении к очереди. Это может нанести производительности реальный урон: даже если задействуется очередь, свободная от блокировок, и отсутствует явное ожидание, существенным пожирателем времени может стать перебрасывание данных в кэш-памяти процессоров.

Один из способов, позволяющий избежать перебрасывания, заключается в применении отдельной очереди работ для каждого потока. Тогда каждый поток помещает новые элементы в собственную очередь и берет работу из глобальной очереди работ, только если в его отдельной очереди работы нет. В листинге 9.6 показана реализация, использующая переменную `thread_local` для того, чтобы гарантировать, что наряду с глобальной очередью у каждого потока имеется собственная очередь работ.

Листинг 9.6. Пул потоков с локальными относительно потоков очередями работ

```
class thread_pool
{
    threadsafe_queue<function_wrapper> pool_work_queue;
    typedef std::queue<function_wrapper> local_queue_type;    ←❶
    static thread_local std::unique_ptr<local_queue_type>
❷→      local_work_queue;
    void worker_thread()
    {
        local_work_queue.reset(new local_queue_type);    ←❸

        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());
❹       if(local_work_queue)
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));   ❺
        }
        return res;
    }
    void run_pending_task()
    {
        function_wrapper task;
        if(local_work_queue && !local_work_queue->empty())  ❻
        {
            task=std::move(local_work_queue->front());
            local_work_queue->pop();
            task();
        }
        else if(pool_work_queue.try_pop(task))  ❼
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
    // все остальное не изменилось
};
```

Для хранения локальной относительно потока очереди работ используется указатель `std::unique_ptr<>` ❷, потому что не хочется, чтобы такую же очередь имели другие потоки, не входящие в ваш пул потоков. Этот указатель инициализируется в функции `worker_thread()` до запуска цикла обработки ❸. Деструктор `std::unique_ptr<>` гарантирует уничтожение очереди работ, как только произойдет выход из потока.

Затем функция `submit()` проверяет, есть ли у текущего потока очередь работ ❹. Если это так, значит, он входит в пул потоков и задачу можно помещать в локальную очередь. В противном случае задача, как и раньше, должна быть помещена в очередь пула ❺.

Похожая проверка есть и в функции `run_pending_task()` ❻, но на этот раз нужно также выяснить, есть ли в локальной очереди какие-либо элементы. Если есть, можно извлечь крайний элемент из очереди и обработать его. Заметьте: поскольку к локальной очереди обращается всего один поток, она может быть простым объектом `std::queue<>` ❶. Если задач в локальной очереди нет, то, как и раньше, выполняется попытка обращения к очереди пула ❼.

Со снижением конкуренции код справляется, но при неравномерном распределении работ запросто может случиться так, что у одного потока скопится большой объем работы, а другим будет нечем заняться. Например, в примере Quicksort в очередь пула попадает только самый первый блок, поскольку оставшиеся блоки окажутся в локальной очереди того рабочего потока, который обрабатывал самый первый блок. Это лишает использование пула потоков всякого смысла.

К счастью, у данной проблемы есть решение: нужно позволить потокам *перехватывать* работу из очередей друг друга, если ее нет и в их очереди, и в глобальной очереди.

9.1.5. Перехват работы

Чтобы позволить безработному потоку брать работу у потока с полной очередью, последняя должна быть доступна перехватывающему потоку из функции `run_pending_tasks()`. Для этого требуется, чтобы каждый поток зарегистрировал свою очередь в пуле потоков или получил очередь от пула. Кроме того, нужно обеспечить данным в очереди работ удобную синхронизацию и защиту, чтобы ваши инварианты были надежно защищены.

Можно создать очередь, свободную от блокировок, которая позволяет владеющему ею потоку помещать элементы в один конец и извлекать их оттуда, чтобы другие потоки могли перехватывать элементы с другого конца. Реализация такой очереди не относится к темам данной книги. Чтобы продемонстрировать саму идею, обратимся для защиты данных очереди к использованию мьютекса. Мы надеемся, что перехват работы останется редким событием, поэтому конкуренция за овладение мьютексом будет невелика, следовательно, и издержки у данной простой очереди будут сведены к минимуму. Простая реализация на основе блокировок показана в листинге 9.7.

Листинг 9.7. Очередь на основе блокировок, предназначенная для перехвата работы

```
class work_stealing_queue
{
private:
    typedef function_wrapper data_type;
    std::deque<data_type> the_queue;    ←❶
    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)                ←❷
    {
        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)    ←❸
```

```
    {
        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)    ←❹
    {
        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;
    }
};
```

Эта очередь является простой оболочкой вокруг `std::deque<function_wrapper>` ❶, защищающей все обращения блокировкой мьютекса. Функции `push()` ❷ и `try_pop()` ❸ работают в начале очереди, а функция `try_steal()` ❹ — в ее конце.

Следовательно, для владельца эта очередь является стеком, работающим по схеме «последним пришел, первым ушел»: только что помещенная в очередь задача станет первой на выход. Это может помочь поднять производительность за счет удобства кэширования, поскольку вероятность того, что данные, относящиеся к этой задаче, все еще будут оставаться в кэше, намного выше, чем вероятность присутствия там данных задачи, помещенной в очередь чуть раньше. Кроме того, все это хорошо согласуется с такими алгоритмами, как Quicksort. В предыдущей реализации при каждом вызове `do_sort()` в стек помещался один элемент, а затем его приходилось ждать. То, что первым обрабатывается последний помещенный в очередь элемент, гарантирует, что блок, необходимый для завершения текущего вызова, будет обработан до тех блоков, которые нужны в других ветвях. Тем самым уменьшатся количество активных задач и общая интенсивность использования стека. Чтобы снизить конкуренцию, функция `try_steal()`, в отличие от функции `try_pop()`, берет элементы с противоположного конца очереди. В принципе, чтобы разрешить конкурентный вызов функций `try_pop()` и `try_steal()`, можно было бы воспользоваться приемом, рассмотренным в главах 6 и 7.

Итак, в нашем распоряжении появилась весьма привлекательная очередь работ, допускающая перехваты. Как же ею можно воспользоваться в пуле потоков? Одна из возможных реализаций показана в листинге 9.8.

Листинг 9.8. Пул потоков, использующий перехват работы

```
class thread_pool
{
    typedef function_wrapper task_type;
    std::atomic_bool done;
```

```
    threadsafe_queue<task_type> pool_work_queue;
    std::vector<std::unique_ptr<work_stealing_queue> > queues;   ❶
    std::vector<std::thread> threads;
    join_threads joiner;
    static thread_local work_stealing_queue* local_work_queue;  ❷
    static thread_local unsigned my_index;
    void worker_thread(unsigned my_index_)
    {
        my_index=my_index_;
        local_work_queue=queues[my_index].get();  ❸
        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)  ❹
    {
        for(unsigned i=0;i<queues.size();++i)
        {
            unsigned const index=(my_index+i+1)%queues.size();  ❺
            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>(  ❻
                                 new work_stealing_queue));
            }
            for(unsigned i=0;i<thread_count;++i)
            {
                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;                                    ❼
        if(pop_task_from_local_queue(task) ||              ←┘
            pop_task_from_pool_queue(task) ||              ←❽
            pop_task_from_other_thread_queue(task))        ←❾
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
};
```

Это код похож на код листинга 9.6. Первое отличие состоит в том, что у каждого потока есть очередь `work_stealing_queue`, а не простая очередь `std::queue<>` ❷. При создании поток не выделяет память под собственную очередь работ — этим занимается конструктор пула ❻, после чего очередь сохраняется в списке очередей работ для этого пула ❶. Затем индекс очереди в списке передается функции потока и используется для получения указателя на очередь ❸. Это означает, что пул потоков может обращаться к очереди при попытке перехвата задачи для безработного потока. Теперь функция `run_pending_task()` попытается получить задачу из очереди, принадлежащей своему потоку ❼, потом из очереди пула ❽ или из очереди другого потока ❾.

Функция `pop_task_from_other_thread_queue()` ❹ обходит очереди, принадлежащие всем потокам пула, пытаясь перехватить задачу поочередно из каждой. Чтобы избежать попыток перехвата каждым потоком из очереди того потока, который значится первым в списке, все потоки начинают обход с потока, следующего в списке за счет смещения индекса в очереди относительно собственного индекса ❺.

Теперь у нас имеется работоспособный пул потоков, пригодный для множества вариантов использования. И по-прежнему существует много способов его усовершенствования для любого конкретного применения. Но эту возможность я оставлю читателю в качестве упражнения. Один из неисследованных аспектов — динамическое изменение размера пула потоков, гарантирующее оптимальное использование центрального процессора, даже когда потоки заблокированы в ходе ожидания чего-либо вроде завершения операции ввода-вывода или снятия блокировки мьютекса.

Следующим в списке усовершенствованных приемов управления потоками значится прерывание потоков.

9.2. Прерывание потоков

В ряде ситуаций неплохо было бы иметь возможность просигнализировать долгоиграющему потоку, что пора остановиться. Возможной причиной будет принадлежность рабочего потока к пулу, который уже уничтожен, или же явное прерывание пользователем работы, выполняемой потоком, и множество других обстоятельств. Какой бы ни была причина, суть одна: нужен сигнал от одного потока другому, что он должен остановиться до естественного завершения выполняемой им работы. И сделать это следует таким образом, чтобы потоку было позволено прервать ее элегантно, вместо того чтобы резко прерывать процесс.

В принципе, можно было бы разработать отдельный механизм для каждого подходящего случая, но это явно будет излишним. Общий механизм не только упростит написание кода в дальнейшем, но и позволит создавать код, выполнение которого можно будет прерывать, не переживая за то, где конкретно он будет использоваться. Стандарт C++11 не предоставляет такой механизм (хотя предложение добавить поддержку прерывания в будущий стандарт C++ выдвигается весьма активно[1]), но создать его самостоятельно не так уж трудно. Посмотрим, как это можно сделать, и начнем рассматривать вопрос с позиции интерфейса для запуска и прерывания потока, а не с позиции самого прерываемого потока.

9.2.1. Запуск и прерывание другого потока

Для начала рассмотрим внешний интерфейс. Что нам понадобится от потока, допускающего свое прерывание? На самом простом уровне — такой же интерфейс, как у `std::thread`, дополненный функцией `interrupt()`:

```
class interruptible_thread
{
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f);
    void join();
    void detach();
```

[1] P0660: A Cooperatively Interruptible Joining Thread, Rev 3, Nicolai Josuttis, Herb Sutter, Anthony Williams. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0660r3.pdf.

```
    bool joinable() const;
    void interrupt();
};
```

Для управления самим потоком можно в выполняемом в нем коде воспользоваться `std::thread` и задействовать для обработки прерывания какую-нибудь специализированную структуру данных управления. А как это будет выглядеть с позиции самого потока? Все должно сводиться к его возможности сказать: «Я могу быть прерван в этом месте», то есть нужна *точка прерывания*. Чтобы не приходилось передавать дополнительные данные, требуется простая функция `interruption_point()`, которую можно было бы вызывать без каких-либо параметров. Тем самым подразумевается, что специализированная структура данных прерываний должна быть доступна через переменную `thread_local`, значение которой устанавливается при запуске потока, чтобы при вызове потоком функция `interruption_point()` проверяла структуру данных для текущего исполняемого потока. Реализация `interruption_point()` будет рассмотрена чуть позже.

Главной причиной использования флага `thread_local` является невозможность применения для управления потоком только класса `std::thread`: память для него должна быть распределена таким образом, чтобы доступ к флагу имелся как у экземпляра `interruptible_thread`, так и у только что запущенного потока. Для этого, как показано в листинге 9.9, прежде чем передавать предоставляемую функцию классу `std::thread`, чтобы запустить поток в конструкторе, ее можно заключить в оболочку.

Листинг 9.9. Основная реализация interruptible_thread

```
class interrupt_flag
{
public:
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;  ←❶
class interruptible_thread
{
    std::thread internal_thread;
    interrupt_flag* flag;
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f)
    {
        std::promise<interrupt_flag*> p;     ←❷
 ❸→     internal_thread=std::thread([f,&p]{
                p.set_value(&this_thread_interrupt_flag);
   ❹→           f();
            });
        flag=p.get_future().get();    ←❺
    }
    void interrupt()
    {
        if(flag)
        {
```

```
            flag->set();    ←❻
        }
    }
};
```

Предоставляемая функция `f` заключена в лямбда-функцию ❸, в которой хранятся копия `f` и ссылка на локальный промис `p` ❷. Перед вызовом копии предоставленной функции ❹ лямбда-функция устанавливает для нового потока в качестве значения промиса адрес `this_thread_interrupt_flag`, объявленный с модификатором `thread_local` ❶. Затем вызывающий поток дожидается готовности фьючерса, связанного с промисом, и сохраняет результат в компонентной переменной `flag` ❺. Следует отметить, что факт запуска лямбда-функции в новом потоке и наличие в ней висячей ссылки на локальную переменную `p` не должны вас беспокоить, поскольку конструктор `interruptible_thread` перед возвращением управления ждет, пока на `p` не останется ссылок в новом потоке. Замечу, что в этой реализации не берется в расчет управление объединением с потоком или отсоединением от него. Об очистке переменной `flag` во избежание появления висячего указателя при выходе из потока или отсоединения от него придется позаботиться самостоятельно.

Теперь конструкция функции `interrupt()` представляется относительно простой: если имеется действительный указатель на флаг прерывания, значит, есть поток, который требуется прервать, и флаг можно установить ❻. Что затем делать с этим прерыванием, решает сам прерываемый поток. Далее посмотрим, как это происходит.

9.2.2. Обнаружение того, что поток был прерван

Теперь можно установить флаг прерывания, но толку от этого не будет, пока поток не проверит наличия требования о его прерывании. В самом простом варианте это делается с помощью функции `interruption_point()`: можно вызвать ее в тот момент, когда прерывание потока будет безопасным, и если флаг установлен, она выдаст исключение `thread_interrupted`:

```
void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
        throw thread_interrupted();
    }
}
```

Этой функцией можно воспользоваться, вызывая ее в коде там, где это удобно:

```
void foo()
{
    while(!done)
    {
        interruption_point();
        process_next_item();
    }
}
```

Решение работоспособное, но не идеальное. Некоторые наиболее подходящие места для прерывания потока находятся там, где он заблокирован в ожидании чего-то, следовательно, не выполняется и не в состоянии вызвать `interruption_point()`. Здесь необходимо средство, позволяющее находиться в состоянии прерываемого ожидания.

9.2.3. Прерывание ожидания на условной переменной

Итак, обозначить прерывания в тщательно подобранных местах кода можно явным вызовом `interruption_point()`, но этот прием не поможет там, где нужно заблокировать поток в ожидании, к примеру, уведомления от условной переменной. Здесь требуется еще одна функция под названием `interruptible_wait()`, которую затем можно будет переопределить под различные варианты ожидания, и нужно выработать способы прерывания ожидания. Как уже упоминалось, один из вариантов — ожидание уведомления от условной переменной. Поэтому с него и начнем: что нужно сделать, чтобы получить возможность прерывать ожидание на условной переменной? Самым простым работоспособным вариантом могут стать уведомление условной переменной об установке флага прерывания и установка точки прерывания сразу же после ожидания. Но чтобы он сработал и гарантированно пробудил нужный поток, придется уведомлять все потоки, находящиеся в режиме ожидания на условной переменной. Но ожидающие потоки в любом случае должны обрабатывать ложные пробуждения, поэтому другие потоки обработают данное явление точно так же, как и ложное пробуждение, — они не смогут отличить одно от другого. В структуре `interrupt_flag` нужно предусмотреть возможность хранения указателя на условную переменную, чтобы ее можно было уведомить в вызове функции `set()`. Одна из возможных реализаций `interruptible_wait()` для условных переменных может выглядеть так, как показано в листинге 9.10.

Листинг 9.10. Нерабочая версия interruptible_wait для std::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);  <-- (1)
(2) --> cv.wait(lk);
    this_thread_interrupt_flag.clear_condition_variable();  <-- (3)
    interruption_point();
}
```

Если предположить, что есть функции для установки и удаления связи условной переменной с флагом прерывания, все в этом коде покажется простым и понятным. В нем проверяется наличие прерывания, связанного с условной переменной, имеющей флаг прерывания `interrupt_flag` для текущего потока ❶, выполняется ожидание на условной переменной ❷, устраняется связь с условной переменной ❸ и снова проверяется наличие прерывания. Если поток прерывается в процессе ожидания на условной переменной, то прерывающий поток пошлет сигнал этой переменной и выведет нужный поток из ожидания, позволяя ему проверить наличие прерывания. К сожалению, этот код *не будет работать*: в нем есть две проблемы.

Если озаботиться безопасностью выдачи исключений, то распознать первый из них довольно просто: `std::condition_variable::wait()` может выдать исключение, поэтому выйти из функции можно без удаления связи флага прерывания с условной переменной. Эту проблему легко устранить, если обзавестись структурой, удаляющей связь в своем деструкторе.

Второй, менее очевидный проблемный момент связан с возникновением состояния гонки. Если поток прерывается после начального вызова `interruption_point()`, но перед вызовом `wait()`, то связь условной переменной с флагом прерывания не играет роли, поскольку *поток не находится в режиме ожидания, следовательно, его нельзя пробудить уведомлением от условной переменной*. Нужно обеспечить невозможность уведомить поток в период между последней проверкой на наличие прерывания и вызовом `wait()`. Если не внедряться во внутреннее устройство `std::condition_variable`, остается только один способ справиться с этой задачей: воспользоваться для защиты мьютексом, удерживаемым `lk`, для чего потребуется передать его в вызов `set_condition_variable()`. К сожалению, при этом возникают новые проблемы: ссылка на мьютекс, время жизни которого неизвестно, передается другому потоку (тому, который выполняет прерывание) для прерывания нужного потока (в вызове `interrupt()`), и при этом неизвестно, заблокирован он на момент вызова или нет. Тем самым создаются предпосылка возникновения взаимной блокировки и возможность обращения к мьютексу после того, как он был уничтожен, то есть этот вариант никуда не годится. Отсутствие надежного способа прерывания ожидания на условной переменной стало бы фактором, существенно ограничивающим ваши возможности. Но ведь можно обойтись и без специальной функции `interruptible_wait()`, так какими же другими вариантами можно воспользоваться? Одним из вариантов будет установка тайм-аута на ожидание, для чего вместо функции `wait()` можно воспользоваться функцией `wait_for()` с небольшим значением тайм-аута (например, 1 мс). Тем самым для потока будет задан верхний предел продолжительности ожидания, прежде чем он увидит наличие прерывания (с учетом промежутка между тактами часов). Если остановиться на этом варианте, ожидающий поток станет чаще ложно пробуждаться из-за тайм-аутов, но тут уж ничего не поделаешь. Реализация этого варианта, наряду с соответствующей реализацией `interrupt_flag`, показана в листинге 9.11.

Листинг 9.11. Использование тайм-аута в interruptible_wait для std::condition_variable

```
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 имеется некий ожидаемый предикат, то тайм-аут 1 мс может быть полностью скрыт внутри цикла 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();
}
```

При этом предикат будет проверяться чаще обычного, но зато им легко будет воспользоваться вместо обычного вызова функции `wait()`. Несложно реализовать и другие варианты с использованием тайм-аутов: ждать, пока не истечет указанное время или 1 мс в зависимости от того, что короче. Итак, с ожиданием на `std::condition_variable` вопрос решен, а как обстоят дела с `std::condition_variable_any`? Там все точно так же или есть более удачные решения?

9.2.4. Прерывание ожидания на std::condition_variable_any

Класс `std::condition_variable_any` отличается от `std::condition_variable` возможностью работать с любым типом блокировки, а не только с `std::unique_lock<std::mutex>`. Ситуация упрощается, и с `std::condition_variable_any` можно найти более удачные решения, чем с `std::condition_variable`. Поскольку этот класс работает с любым типом блокировки, можно создать собственный тип, выставляющий и снимающий как блокировку при работе с внутренним мьютексом `set_clear_mutex` в вашем `interrupt_flag`, так и блокировку, переданную при вызове функции `wait`, как показано в листинге 9.12.

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();     ❶
                self->thread_cond_any=&cond;      ❷
            }
            void unlock()  ❸
            {
                lk.unlock();
                self->set_clear_mutex.unlock();
            }
            void lock()
            {
                std::lock(self->set_clear_mutex,lk);  ❹
            }
            ~custom_lock()
            {
                self->thread_cond_any=0;      ❺
                self->set_clear_mutex.unlock();
            }
        };
        custom_lock cl(this,cv,lk);
        interruption_point();
        cv.wait(cl);
        interruption_point();
    }
    // все остальное не изменилось
};
template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv, Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}
```

Созданный только что тип блокировки получает блокировку на внутреннем мьютексе `set_clear_mutex` при его создании ❶, а затем устанавливает значение указателя `thread_cond_any` для ссылки на объект `std::condition_variable_any`, переданный в конструктор ❷. Ссылка на объект `Lockable`, доступ к которому уже должен быть заблокирован, сохраняется на будущее. Теперь можно проверить наличие сигнала на прерывания, не опасаясь, что возникнет состояние гонки.

Если в данном месте установлен флаг прерывания, значит, это было сделано до блокировки на мьютексе `set_clear_mutex`. Когда условной переменной внутри функции `wait()` вызывается ваша функция `unlock()`, снимается блокировка с объекта `Lockable` и с внутреннего мьютекса `set_clear_mutex` ❸. Это позволяет потокам, пытающимся прервать текущий поток, заблокировать мьютекс `set_clear_mutex` и проверить указатель `thread_cond_any`, как только управление окажется внутри вызова функции `wait()`, но никак не раньше. Это именно то, к чему мы стремились, но чего не смогли добиться при использовании `std::condition_variable`. Как

только ожидание в функции `wait()` завершится (либо из-за отсутствия уведомления, либо из-за ложного пробуждения), она вызовет функцию `lock()`, которая снова заблокирует внутренний мьютекс `set_clear_mutex` и объект `Lockable` ❹. Теперь опять можно будет проверить наличие прерывания, появившегося в ходе вызова функции `wait()`, прежде чем удалить указатель `thread_cond_any` в деструкторе `custom_lock destructor` ❺, где также снимается блокировка с мьютекса `set_clear_mutex`.

9.2.5. Прерывание других блокирующих вызовов

С прерыванием ожидания на условной переменной вопрос решен, а как насчет других ожиданий на блокировках: на заблокированных мьютексах, на готовности фьючерсов и т. д.? В принципе, нужно обратиться к использованию варианта с тайм-аутом, как уже было с `std::condition_variable`, поскольку нет простого способа прервать ожидание или выполнить ожидаемое условие без обращения к внутренним составляющим мьютекса или фьючерса. Но относительно этих ожиданий заранее известно, что именно ожидается, поэтому можно воспользоваться циклом внутри функции `interruptible_wait()`. В качестве примера рассмотрим переопределение функции `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();
}
```

Ожидание здесь продолжается до тех пор, пока не будет установлен флаг прерывания либо не будет готов фьючерс, но при каждом прохождении цикла блокирующее ожидание на готовности фьючерса продолжается 1 мс. Это означает, что в среднем сигнал на прерывание будет распознан примерно через 0,5 мс (для системных часов с довольно высокой частотой тактов). Ожидание в функции `wait_for` обычно длится по меньшей мере в течение одного такта системных часов, поэтому, если такт составляет 15 мс, ожидать придется не одну, а около 15 мс. Приемлемо это или нет, зависит от конкретных обстоятельств. При необходимости тайм-аут можно уменьшить (если системные часы позволяют), но тогда поток будет чаще пробуждаться для проверки установки флага, что увеличивает издержки на переключение задач.

Итак, мы рассмотрели возможности обнаружения прерывания с помощью функций `interruption_point()` и `interruptible_wait()`. А как обрабатываются прерывания?

9.2.6. Обработка прерываний

С позиции прерываемого потока прерывание представляется исключением `thread_interrupted`, которое в таком случае может быть обработано, как и любое другое исключение. В частности, его можно перехватить в стандартном блоке `catch`:

```
try
{
    do_something();
}
catch(thread_interrupted&)
{
    handle_interruption();
}
```

То есть исключение можно перехватить, неким образом обработать, после чего работу можно продолжить в обычном порядке. Если так и сделано и другой поток снова станет вызывать `interrupt()`, ваш поток опять прервется при следующем вызове точки прерывания. Такое развитие событий может потребоваться при выполнении серии независимых друг от друга задач: прерывание выполнения одной задачи вызовет ее отставку и поток сможет перейти к выполнению следующей задачи, имеющейся в списке.

Поскольку `thread_interrupted` является исключением, то при вызове кода, выполнение которого может быть прервано, во избежание утечки ресурсов и рассогласованности в состоянии вашей структуры данных должны быть предприняты все обычные меры безопасности выдачи исключений. Зачастую желательно дать прерыванию завершить поток, позволив исключению распространиться вверх по цепочке вызовов. Но если разрешить исключениям распространяться за пределы функции потока, переданной конструктору `std::thread`, будет вызвана функция `std::terminate()` и прекращено выполнение всей программы. Чтобы не нужно было помнить о необходимости помещения обработчика `catch(thread_interrupted)` в каждую функцию, передаваемую `interruptible_thread`, можно вместо этого поместить блок `catch` внутрь оболочки, используемой для инициализации флага прерывания `interrupt_flag`. Тем самым будет обеспечена безопасность распространения исключения прерывания в необработанном виде, так как завершение коснется только данного потока. Теперь инициализация потока в конструкторе `interruptible_thread` будет такой:

```
internal_thread=std::thread([f,&p]{
    p.set_value(&this_thread_interrupt_flag);
    try
    {
        f();
    }
    catch(thread_interrupted const&)
    {}
});
```

А теперь рассмотрим конкретный пример, в котором может пригодиться прерывание.

9.2.7. Прерывание фоновых задач при выходе из приложения

Представьте себе приложение поиска на настольном компьютере. Кроме общения с пользователем, ему нужно отслеживать состояние файловой системы, обнаруживая любые изменения и обновляя свой индекс. Обычно, чтобы избежать отрицательного воздействия на отзывчивость графического пользовательского интерфейса, такая обработка возлагается на фоновый поток. Он должен быть запущен в течение всего жизненного цикла приложения, его запуск становится частью инициализации приложения, и он остается в рабочем состоянии до выхода из приложения. Выход из таких приложений обычно выполняется только при выключении машины, поскольку приложение должно работать все время, чтобы иметь возможность поддерживать актуальность индекса. В любом случае при завершении приложения нужно аккуратно закрыть все фоновые потоки. Один из способов — прерывание потоков.

В листинге 9.13 показана примерная реализация той части системы, которая управляет потоками.

Листинг 9.13. Отслеживание состояния файловой системы в фоновом режиме

```
std::mutex config_mutex;
std::vector<interruptible_thread> background_threads;
void background_thread(int disk_id)
{
    while(true)
    {
        interruptible_thread();                          ❶
        fs_change fsc=get_fs_changes(disk_id);           ❷
        if(fsc.has_changes())
        {
            update_index(fsc);                           ❸
        }
    }
}
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();                       ❹
❺   process_gui_until_exit();
    std::unique_lock<std::mutex> lk(config_mutex);
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].interrupt();               ❻
    }
```

```
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].join();   ←❼
    }
}
```

Фоновые потоки запускаются в самом начале ❸. Затем основной поток занимается обработкой графического интерфейса пользователя ❹. После того как пользователь запросит выход из приложения, фоновые потоки прерываются ❺, а основной поток, прежде чем выполнить выход, дожидается завершения каждого фонового потока ❻. В фоновых потоках запускаются циклы, в которых отслеживаются изменения на диске ❼ и выполняется обновление индекса ❷. При каждом проходе цикла путем запуска `interruption_point()` проверяется наличие прерывания ❶.

А зачем прерывать все потоки перед ожиданием завершения какого-либо из них? Почему бы не прерывать работу каждого из них, а затем дожидаться его завершения, перед тем как перейти к следующему потоку? Ответ кроется в *конкурентности*. Вероятнее всего, работа потоков не будет завершена сразу же, как только они получат сигнал на прерывание, поскольку им придется дойти до следующей точки прерывания, а перед выходом вызвать какой-либо деструктор и необходимый код обработки исключений. Немедленное объединение с каждым потоком заставит прерывающий поток входить в режим ожидания, даже если у него еще есть работа, которую он может выполнять (имеется в виду прерывание других потоков). В ожидание можно входить, лишь когда нечем будет заняться (все потоки уже прерваны). Это позволит всем прерываемым потокам обработать свои прерывания в параллельном режиме, что может ускорить завершение работы.

Такой механизм прерывания можно без особого труда расширить, добавив дополнительные прерываемые вызовы или отключив прерывания в определенном блоке кода, но это предлагаю вам выполнить самостоятельно.

Резюме

В этой главе мы рассмотрели продвинутые приемы управления потоками: использование пулов потоков и прерывание потоков. Было продемонстрировано, как использование локальных очередей работ и перехват работ может сократить издержки на обеспечение синхронизации и повысить пропускную способность пула потоков. Кроме того, мы разобрались, как выполнение других задач из очереди в ходе ожидания завершения подзадачи может исключить возможность взаимоблокировки.

Мы также разобрали различные способы, позволяющие одному потоку прервать работу другого, в частности применение конкретных точек прерывания и функций, которые допускают прерывание во время блокирующего ожидания.

10 Алгоритмы параллельных вычислений

В этой главе

- Использование алгоритмов параллельных вычислений C++17.

В предыдущей главе мы рассмотрели усовершенствованное управление потоками и пулы потоков, а глава 8 была посвящена разработке конкурентного кода, где в качестве примеров использовались параллельные версии некоторых алгоритмов. В этой главе мы разберем алгоритмы параллельных вычислений (parallel algorithms), предоставляемые стандартом C++17.

10.1. Перевод стандартных библиотечных алгоритмов в режим параллельных вычислений

В стандарт C++17 к стандартной библиотеке C++ добавлена концепция *алгоритмов параллельных вычислений*. Она представлена в виде дополнительных переопределений многих функций, работающих с диапазонами, в числе которых `std::find`, `std::transform` и `std::reduce`. У параллельных версий такая же сигнатура, что

и у «обычных» однопоточных, за исключением добавления нового первого параметра, определяющего *политику выполнения*. Например:

```
std::vector<int> my_data;
std::sort(std::execution::par,my_data.begin(),my_data.end());
```

Политика выполнения `std::execution::par` указывает стандартной библиотеке, что данный вызов разрешено выполнять в качестве алгоритма параллельных вычислений с задействованием сразу нескольких потоков. Следует отметить, что это *разрешение*, а не *требование* — библиотека вольна по-прежнему выполнять код в одном потоке. Обратите внимание, что с заданием политики выполнения изменяются требования к сложности алгоритма, и они обычно слабее требований, предъявляемых к обычному алгоритму последовательных вычислений. Чтобы можно было воспользоваться преимуществами параллелизма системы, алгоритмы параллельных вычислений зачастую выполняют работу более общего характера. Если работу можно разделить и выполнить на 100 процессорах, то можно достичь чуть ли не 50-кратного ускорения, даже если общий объем работы в реализации удваивается.

Прежде чем перейти к самим алгоритмам, рассмотрим политики выполнения.

10.2. Политики выполнения

Стандартом предусмотрены три политики выполнения:

- `std::execution::sequenced_policy`;
- `std::execution::parallel_policy`;
- `std::execution::parallel_unsequenced_policy`.

Они представляют собой классы, определенные в заголовке `<execution>`. Там же определены три соответствующих объекта политик, которые передаются алгоритмам:

- `std::execution::seq`;
- `std::execution::par`;
- `std::execution::par_unseq`.

Нельзя рассчитывать на самостоятельное создание объектов из этих классов политик, поскольку из-за возможных особых требований по инициализации остается полагаться только на копирование этих трех объектов. В реализациях могут также определяться дополнительные политики выполнения с характерными для них особенностями поведения. Задавать свои собственные политики выполнения невозможно.

О влиянии этих политик на поведение алгоритмов речь пойдет в подразделе 10.2.1. В любой конкретной реализации также может предоставляться дополнительная политика выполнения с любой произвольной семантикой. А теперь рассмотрим последствия использования одной из стандартных политик выполнения, начиная с общих изменений для всех переопределений алгоритма, принимающего политику исключений.

10.2.1. Общие последствия от задания политики выполнения

Если передать политику выполнения одному из стандартных библиотечных алгоритмов, то она станет определять его поведение. Это повлияет на несколько характеристик поведения:

- сложность алгоритма;
- поведение при выдаче исключения;
- место, способ и время выполнения этапов работы алгоритма.

Влияние на сложность алгоритма

Если алгоритму предоставляется политика выполнения, его сложность может измениться: в дополнение к издержкам диспетчеризации выполнения параллельных вычислений, многими параллельными алгоритмами будет выполняться более весомый объем основных операций алгоритма (будь то *обмены, сравнения или использования предоставляемого функционального объекта*). Это предположительно улучшит производительность относительно затраченного времени.

Детали изменения сложности будут варьироваться для каждого алгоритма, но общая политика заключается в том, что если алгоритм определяет, что нечто произойдет за время, точно соответствующее вычислению какого-либо выражения *(some-expression)*, или не превысит время вычисления какого-либо выражения, то переопределение алгоритма с применением политики выполнения ослабит это требование до *O(some-expression)*. Это означает, что переопределение алгоритма с применением политики выполнения может выполнять некоторое кратное число операций, выполняемых его аналогом, не применяющим политику выполнения, где эта кратность будет зависеть от внутренних особенностей библиотеки и платформы, а не от данных, предоставленных алгоритму.

Поведение при выдаче исключения

Если алгоритм выполняется с заданной политикой и при этом выдается исключение, то последствия определяются этой политикой. При выдаче любых неперехваченных исключений всеми предоставляемыми стандартом политиками выполнения предусматривается вызов функции `std::terminate`. Единственным исключением, которое может быть выдано вызовом стандартного библиотечного алгоритма с одной из стандартных политик выполнения, является `std::bad_alloc`. Оно выдается, если библиотека не может получить достаточный объем ресурсов памяти для своих внутренних операций. Например, следующий вызов `std::for_each` без задания политики выполнения распространит исключение:

```
std::for_each(v.begin(),v.end(),[](auto x){ throw my_exception(); });
```

а соответствующий вызов с заданием политики выполнения завершит выполнение программы:

```
std::for_each(
    std::execution::seq,v.begin(),v.end(),
    [](auto x){ throw my_exception(); });
```

Это одно из главных различий между использованием `std::execution::seq` и непредоставлением политики выполнения.

Где и когда выполняются этапы работы алгоритма

Это основной аспект политики выполнения и единственный аспект, определяющий различия между стандартными политиками выполнения. Политика определяет, какие агенты выполнения задействуются для реализации этапов алгоритма, будь то «обычные» потоки, векторные потоки, потоки графического процессора или что-либо еще. Политика выполнения также будет определять ограничения в порядке выполнения работы алгоритма: должны ли они запускаться в определенной последовательности, могут ли отдельные этапы работы алгоритма перемежаться или запускаться параллельно друг с другом и т. д.

Подробности каждой стандартной политики выполнения рассматриваются в подразделах 10.2.2–10.2.4. Сначала мы обсудим основную политику: `std::execution::sequenced_policy`.

10.2.2. std::execution::sequenced_policy

Последовательная политика (sequenced policy) не является политикой параллелизма: ее использование заставляет реализацию выполнять все операции в потоке, где была вызвана функция, то есть без распараллеливания. Но все же это политика выполнения, в силу чего она оказывает такое же влияние на алгоритмическую сложность и на выдачу исключений, как и все другие стандартные политики.

Все операции должны выполняться не только в одном потоке, но и в определенном порядке, то есть они не могут перемежаться. Конкретный порядок не устанавливается и может различаться для разных вызовов функции. В частности, не гарантируется, что порядок выполнения операций будет таким же, как и в соответствующем переопределении без задания политики выполнения. Например, следующий вызов `std::for_each` заполнит вектор числами 1–1000 в неопределенном порядке. Это будет отличаться от выполнения переопределения алгоритма, не предусматривающего задания политики выполнения, который сохранит числа в порядке возрастания:

```
std::vector<int> v(1000);
int count=0;
std::for_each(std::execution::seq,v.begin(),v.end(),
    [&](int& x){ x=++count; });
```

Числа могут быть сохранены и в порядке возрастания, но полагаться на это нельзя.

Это означает, что политика последовательного выполнения не предъявляет особых требований к используемым с алгоритмом итераторам, значениям и вызываемым объектам: они могут свободно использовать механизмы синхронизации и полагаться на все операции, вызываемые в том же потоке, но не могут рассчитывать на порядок выполнения этих операций.

10.2.3. std::execution::parallel_policy

Параллельная политика (parallel policy) предоставляет основное параллельное выполнение сразу в нескольких потоках. Операции можно выполнять либо в потоке, вызвавшем алгоритм, либо в потоках, созданных библиотекой. Операции, выполняемые в конкретном потоке, должны выполняться в определенном порядке и не перемежаться, но точный порядок не определен и от вызова к вызову может варьироваться. Конкретная операция будет выполняться в фиксированном потоке на протяжении всего времени.

Это накладывает дополнительные требования на итераторы, значения и вызываемые объекты, используемые с алгоритмом при применении последовательной политики: они не должны приводить к состоянию гонки за данными при параллельном вызове и не должны полагаться на выполнение в том же потоке, что и любая другая операция, или же действительно полагаться на то, что не будут выполняться в том же потоке, что и любая другая операция.

Политикой параллельного выполнения можно воспользоваться в подавляющем большинстве случаев, когда используется алгоритм стандартной библиотеки без применения политики выполнения. Обойдется без проблем только при наличии определенного порядка между необходимыми элементами или несинхронизированного обращения к совместно используемым данным. Увеличение на единицу всех значений, имеющихся в векторе, может быть выполнено в режиме параллельных вычислений:

```
std::for_each(std::execution::par,v.begin(),v.end(),[](auto& x){++x;});
```

А для заполнения вектора предыдущий пример с использованием политики параллельного выполнения не подойдет, поскольку возникнет неопределенное поведение:

```
std::for_each(std::execution::par,v.begin(),v.end(),
    [&](int& x){ x=++count; });
```

Здесь переменная `count` изменяется при каждом вызове лямбда-функции, поэтому, если бы библиотеке пришлось выполнять лямбда-функции сразу в нескольких потоках, возникло бы состояние гонки, что привело бы к неопределенному поведению. Это исключается требованиями, предъявляемыми к `std::execution::parallel_policy`: имеется в виду неопределенное поведение при выполнении предыдущего вызова, даже если библиотека не использует для этого вызова несколько потоков. Проявление или непроявление неопределенного поведения является статическим свойством вызова, не зависящим от особенностей реализации библиотеки. Но синхронизация между вызовами функции разрешена, поэтому вполне возможно придать поведению определенность, задав для `count` не обычный тип `int`, а `std::atomic<int>` или же воспользовавшись мьютексом. В таком случае использование политики параллельного выполнения будет бессмысленным, поскольку все вызовы станут последовательными, но в общем случае это позволит синхронизировать доступ к совместно используемому состоянию.

10.2.4. std::execution::parallel_unsequenced_policy

Политика параллельного неупорядоченного выполнения (parallel unsequenced policy) предоставляет библиотеке наибольший масштаб распараллеливания алгоритма в обмен на предъявление строжайших требований к итераторам, значениям и вызываемым объектам, используемым с алгоритмом.

Алгоритм, вызываемый с применением параллельной неупорядоченной политики, может выполнять этапы алгоритма в неопределенных потоках выполнения, неупорядоченных и непоследовательных относительно друг друга. Это означает, что операции теперь могут перемежаться в одном потоке, благодаря чему вторая операция будет запускаться в одном и том же потоке до завершения первой операции и может быть перенесена между потоками. То есть конкретная операция может начаться в одном потоке, продолжиться во втором, а завершиться в третьем.

Операции, предоставляемые алгоритму с использованием политики параллельного неупорядоченного выполнения, при их вызове итераторами, значениями и вызываемыми объектами, не должны использовать какие-либо формы синхронизации или вызовы любых функций, синхронизируемых с другими функциями, или же любых функций, с которыми синхронизируется другой код.

Это означает, что операции должны работать только с соответствующим элементом или с любыми данными, доступными на основе этого элемента, и не должны изменять какое-либо состояние, совместно используемое потоками или элементами.

Подробности будут раскрыты на нескольких примерах чуть позже. А теперь посмотрим на сами параллельные алгоритмы.

10.3. Параллельные алгоритмы из стандартной библиотеки C++

Большинство алгоритмов из заголовков `<algorithm>` и `<numeric>` имеют переопределения, воспринимающие политику выполнения. В их числе: `all_of`, `any_of`, `none_of`, `for_each`, `for_each_n`, `find`, `find_if`, `find_end`, `find_first_of`, `adjacent_find`, `count`, `count_if`, `mismatch`, `equal`, `search`, `search_n`, `copy`, `copy_n`, `copy_if`, `move`, `swap_ranges`, `transform`, `replace`, `replace_if`, `replace_copy`, `replace_copy_if`, `fill`, `fill_n`, `generate`, `generate_n`, `remove`, `remove_if`, `remove_copy`, `remove_copy_if`, `unique`, `unique_copy`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `is_partitioned`, `partition`, `stable_partition`, `partition_copy`, `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`, `is_sorted`, `is_sorted_until`, `nth_element`, `merge`, `inplace_merge`, `includes`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`, `is_heap`, `is_heap_until`, `min_element`, `max_element`, `minmax_element`, `lexicographical_compare`, `reduce`, `transform_reduce`, `exclusive_scan`, `inclusive_scan`, `transform_exclusive_scan`, `transform_inclusive_scan` и `adjacent_difference`.

В этом обширном списке перечислены почти все алгоритмы стандартной библиотеки C++, которые могут быть распараллелены. Исключением является такой алгоритм, как `std::accumulate`, представляющий собой сугубо последовательное накопление. Его обобщенный контрпартнер `std::reduce` все же фигурирует в списке,

но с соответствующей оговоркой в стандарте: операция сокращения не является ни ассоциативной, ни коммутативной, результат из-за неопределенного порядка выполнения операций может быть недетерминированным.

Для каждого указанного в списке алгоритма каждое «обычное» переопределение обладает новым вариантом, принимающим в качестве первого аргумента политику выполнения. Затем за этой политикой выполнения следуют соответствующие аргументы для «обычного» алгоритма. Например, у `std::sort` есть два «обычных» переопределения без задания политики выполнения:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(
    RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Соответственно, у него есть и два переопределения с заданием политики выполнения:

```
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Сигнатуры, используемые в вариантах с заданием политики выполнения и без ее задания, имеют одно весьма важное различие, оказывающее влияние лишь на некоторые алгоритмы: если в «обычном» алгоритме допускаются *итераторы ввода (input iterators)* или *итераторы вывода (output iterators)*, то в переопределениях с заданием политики выполнения вместо них требуются *однонаправленные итераторы (forward iterators)*. Дело в том, что входные итераторы в основном являются однопроходными: доступ можно получить только к текущему элементу и нельзя сохранить итераторы для предыдущих элементов. Аналогично выходные итераторы позволяют вести запись только текущего элемента: их нельзя передвинуть на запись более позднего элемента, а затем вернуться и записать предыдущий элемент.

Категории итераторов в стандартной библиотеке C++

В стандартной библиотеке C++ определено пять категорий итераторов: итераторы ввода (input iterators), итераторы вывода (output iterators), однонаправленные итераторы (forward iterators), двунаправленные итераторы (bidirectional iterators) и итераторы произвольного доступа (random access iterators).

Итераторы ввода являются однопроходными итераторами для извлечения значений. Обычно они используются для ввода с консоли, или из сети, или из сгенерированных

последовательностей. Продвижение итератора ввода делает любые его копии недействительными.

Итераторы вывода являются однопроходными итераторами для записи значений. Обычно они используются для вывода данных в файлы или для добавления значений в контейнер. Продвижение итератора вывода делает любые его копии недействительными.

Однонаправленные итераторы являются многопроходными для однонаправленного последовательного обхода существующих данных. Следовательно, итератор нельзя вернуть на предыдущий элемент, можно сохранить копии и воспользоваться ими для ссылок на предыдущие элементы. Однонаправленные итераторы возвращают ссылки на элементы, следовательно, ими можно воспользоваться как для чтения, так и для записи (если целевой элемент не относится к типу `const`).

Двунаправленные итераторы, как и однонаправленные, являются многопроходными, но их также можно заставить повернуться вспять для доступа к предыдущим элементам.

Итераторы произвольного доступа являются многопроходными, способными к переходам вперед-назад, как и двунаправленные итераторы, но переходы вперед-назад осуществляются шагами шире одного элемента. Можно напрямую получить доступ к элементам со смещением, используя индексный оператор массива.

Итак, исходя из «обычной» сигнатуры для `std::copy`:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator first, InputIterator last, OutputIterator result);
```

получаем переопределение с заданием политики выполнения:

```
template<class ExecutionPolicy,
    class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(
    ExecutionPolicy&& policy,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);
```

Хотя с точки зрения компилятора наименование параметров шаблона не влечет за собой никаких прямых последствий, такие последствия имеются с точки зрения стандартов C++: имена параметров шаблона для алгоритмов стандартной библиотеки обозначают семантические ограничения на типы и алгоритмы с указанной семантикой будут зависеть от операций, подразумеваемых имеющимися ограничениями. Если сравнить итератор ввода и однонаправленный итератор, то первый позволяет разыменовать итератор для возврата прокси-типа, который можно преобразовать в тип значения итератора, а второй требует, чтобы при разыменовании итератора возвращалась реальная ссылка на значение и все одинаковые итераторы возвращали ссылку на одно и то же значение.

Для параллельных вычислений это играет важную роль: это значит, что итераторы можно свободно и повсеместно копировать и использовать эквивалентным

образом. Кроме того, важно и требование о том, что приращение однонаправленного итератора не делает недействительными другие копии, поскольку это означает, что отдельно взятые потоки могут работать со своими собственными копиями итераторов, задавая им приращение при необходимости, не беспокоясь об аннулировании итераторов других потоков. Если переопределение с помощью политики выполнения позволяет использовать итераторы ввода, это заставит любые потоки применять последовательную форму доступа к единственному итератору, который был задействован для считывания исходной последовательности. Это, конечно же, приведет к ограничению возможностей по распараллеливанию.

Рассмотрим несколько конкретных примеров.

10.3.1. Примеры использования параллельных алгоритмов

Самый простой пример — параллельный цикл: выполнение каких-либо действий над каждым элементом контейнера. Это классический пример сугубо параллельного сценария: каждый элемент независим от всех остальных, следовательно, предоставляется максимальная возможность для распараллеливания. При использовании компилятора, поддерживающего OpenMP, можно воспользоваться следующим кодом:

```
#pragma omp parallel for
for(unsigned i=0;i<v.size();++i){
    do_stuff(v[i]);
}
```

А используя алгоритмы стандартной библиотеки C++, вместо него можно написать код:

```
std::for_each(std::execution::par,v.begin(),v.end(),do_stuff);
```

Он приведет к разбиению элементов на диапазоны между внутренними потоками, созданными библиотекой, и к вызову функции `do_stuff(x)` в отношении каждого элемента `x`, входящего в диапазон. Способ разбиения этих элементов по потокам — особенность реализации.

Выбор политики выполнения

Наиболее часто приходится иметь дело с политикой `std::execution::par`, если только в вашей реализации не предоставляется нестандартная политика, более полно отвечающая вашим потребностям. Если ваш код подходит для распараллеливания, то он должен работать с `std::execution::par`. В некоторых случаях вы можете использовать `std::execution::par_unseq`. Это может вообще ни к чему не привести (ни одна из стандартных политик выполнения не дает гарантий по достигаемому уровню распараллеливания), но может дать библиотеке дополнительные возможности повышения производительности кода за счет изменения порядка выполнения и чередования задач взамен наложения на код более жестких требований.

Наиболее заметным из таких требований является отсутствие синхронизации при обращении к элементам или при выполнении операций над элементами. Это означает невозможность применения мьютексов, или атомарных переменных, или любого другого механизма, рассмотренного в предыдущих главах, который обеспечивает безопасность обращения из нескольких потоков. Вместо этого нужно полагаться на то, что сам алгоритм не станет обращаться к одному и тому же элементу сразу из нескольких потоков и использовать внешнюю синхронизацию за пределами вызова параллельного алгоритма для предотвращения обращения к данным со стороны других потоков.

В листинге 10.1 показан код, которым можно воспользоваться с политикой выполнения `std::execution::par`, но не с `std::execution::par_unseq`. Использование для синхронизации внутреннего мьютекса означает, что попытка воспользоваться политикой `std::execution::par_unseq` приведет к неопределенному поведению.

Листинг 10.1. Применение параллельных алгоритмов к классу с внутренней синхронизацией

```
class X{
    mutable std::mutex m;
    int data;
public:
    X():data(0){}
    int get_value() const{
        std::lock_guard guard(m);
        return data;
    }
    void increment(){
        std::lock_guard guard(m);
        ++data;
    }
};
void increment_all(std::vector<X>& v){
    std::for_each(std::execution::par,v.begin(),v.end(),
        [](X& x){
            x.increment();
        });
}
```

В листинге 10.2 показан альтернативный вариант, которым можно воспользоваться с политикой выполнения `std::execution::par_unseq`. В этом случае вместо внутреннего поэлементного мьютекса используется мьютекс, действие которого распространяется на весь контейнер.

Листинг 10.2. Применение параллельных алгоритмов к классу без внутренней синхронизации

```
class Y{
    int data;
public:
    Y():data(0){}
    int get_value() const{
        return data;
```

```
    }
    void increment(){
        ++data;
    }
};
class ProtectedY{
    std::mutex m;
    std::vector<Y> v;
public:
    void lock(){
        m.lock();
      }
    void unlock(){
        m.unlock();
      }
      std::vector<Y>& get_vec(){
        return v;
    }
};
void increment_all(ProtectedY& data){
    std::lock_guard guard(data);
    auto& v=data.get_vec();
    std::for_each(std::execution::par_unseq,v.begin(),v.end(),
        [](Y& y){
            y.increment();
        });
}
```

Теперь в листинге 10.2 обращение к элементу происходит без синхронизации, и можно без каких-либо опасений воспользоваться политикой `std::execution::par_unseq`. Недостаток такого решения проявляется в том, что при конкурентном доступе из других потоков, не входящих в сферу вызова параллельного алгоритма, приходится ждать завершения всей операции, в отличие от поэлементной детализации, имеющейся в коде листинга 10.1.

А теперь рассмотрим более реалистичный пример, показывающий порядок использования параллельных алгоритмов: подсчет количества посещений сайта.

10.3.2. Подсчет посещений

Предположим, что у нас есть запущенный деловой сайт, в журнале которого содержатся миллионы записей. Нужно обработать регистрационные записи, чтобы увидеть сводные данные: количество посещений каждой страницы, откуда они были, какие браузеры использовались для обращения к сайту и т. д. Анализ регистрационных записей состоит из двух частей: обработки каждой строки для извлечения нужной информации и получения сводных данных. Это идеальный сценарий для использования параллельных алгоритмов, поскольку обработка каждой отдельно взятой строки проводится независимо от чего бы то ни было, а создание сводки, если конечные итоги верны, может вестись по частям.

Получается та задача, для которой и разработана функция `transform_reduce`. Как ее можно использовать для решения данной задачи, показано в листинге 10.3.

Листинг 10.3. Использование transform_reduce для подсчета посещений страниц на сайте

```
#include <vector>
#include <string>
#include <unordered_map>
#include <numeric>
struct log_info {
    std::string page;
    time_t visit_time;
    std::string browser;
    // любые другие поля
};

extern log_info parse_log_line(std::string const &line);    ❶

using visit_map_type= std::unordered_map<std::string, unsigned long long>;

visit_map_type
count_visits_per_page(std::vector<std::string> const &log_lines) {

    struct combine_visits {
        visit_map_type
        operator()(visit_map_type lhs, visit_map_type rhs) const {  ❸
            if(lhs.size() < rhs.size())
                std::swap(lhs, rhs);
            for(auto const &entry : rhs) {
                lhs[entry.first]+= entry.second;
            }
            return lhs;
        }

  ❹     visit_map_type operator()(log_info log,visit_map_type map) const{
            ++map[log.page];
            return map;
        }
  ❺     visit_map_type operator()(visit_map_type map,log_info log) const{
            ++map[log.page];
            return map;
        }
  ❻     visit_map_type operator()(log_info log1,log_info log2) const{
            visit_map_type map;
            ++map[log1.page];
            ++map[log2.page];
            return map;
        }
    };

    return std::transform_reduce(  ❷
        std::execution::par, log_lines.begin(), log_lines.end(),
        visit_map_type(), combine_visits(), parse_log_line);
}
```

Предположим, что для извлечения нужной информации из регистрационной записи вы располагаете некоей функцией `parse_log_line` ❶, ваша функция `count_visits_per_page` служит простой оболочкой, в которую заключен вызов `std::transform_reduce` ❷. Сложности исходят из операции *предварительной обработки*: нужно скомбинировать две структуры `log_info` для создания отображения, структуру `log_info` и отображение (тем или иным образом) и два отображения. Следовательно, это означает, что ваш функциональный объект `combine_visits` нуждается для оператора функционального вызова в четырех переопределениях, ❸, ❹, ❺ и ❻, что не позволяет обойтись простой лямбда-функцией, несмотря на то что реализация этих четырех переопределений довольно проста.

Поэтому для выполнения данных вычислений в параллельном режиме (поскольку была передана политика `std::execution::par`) реализацией `std::transform_reduce` будет использоваться доступное оборудование. Как уже было показано в предыдущей главе, создание своего собственного подобного алгоритма — задача непростая, следовательно, вам предоставляется возможность переложить всю трудную работу по реализации параллелизма на разработчиков стандартной библиотеки и сосредоточиться на требуемом результате.

Резюме

В данной главе мы рассмотрели параллельные алгоритмы, доступные в стандартной библиотеке C++, и порядок их использования. Уделили внимание различным политикам выполнения, влиянию выбора политики выполнения на поведение алгоритма и на ограничения для вашего кода. Затем разобрали примеры возможного применения этих алгоритмов в реальном коде.

11 Тестирование и отладка многопоточных приложений

В этой главе

- Ошибки, связанные с конкурентностью.
- Поиск ошибок при тестировании и анализе кода.
- Разработка многопоточных тестов.
- Тестирование производительности многопоточного кода.

До сих пор основное внимание мы уделяли всему, что относится к созданию конкурентного кода, — доступному инструментарию, порядку его использования и общим вопросам конструкций и структур кода. Но одну из жизненно важных составляющих разработки такого кода пока еще не рассматривали — речь идет о тестировании и отладке. Если, читая эту главу, вы надеетесь узнать простой способ тестирования конкурентного кода, вы будете разочарованы. Тестирование и отладка конкурентного кода даются нелегко. Я же хочу познакомить вас с приемами, облегчающими решение этой задачи, а также рассмотреть вопросы, о которых стоит призадуматься.

Тестирование и отладка сродни двум сторонам медали — код подвергается тестированию для поиска возможных ошибок, а его отладка ведется с целью устранения этих ошибок. Если повезет, то вам нужно будет только устранить ошибки, найденные вашими собственными тестами, а не обнаруженные конечными пользователями вашего приложения. Прежде чем приступить к вопросам тестирования либо отладки, важно разобраться с возможными проблемами. Этим мы сейчас и займемся.

11.1. Типы ошибок, связанных с конкурентностью

При написании конкурентного кода вы можете столкнуться с любыми разновидностями ошибок, и это неудивительно. Но некоторые типы ошибок напрямую связаны с использованием конкурентности и имеют самое непосредственное отношение к теме данной книги. Обычно ошибки, связанные с конкурентностью, разбиваются на две категории, такие как:

- нежелательная блокировка;
- состояние гонки.

Это обобщенные категории, поэтому давайте разобьем их на части. Начнем с нежелательной блокировки.

11.1.1. Нежелательная блокировка

Что подразумевается под нежелательной блокировкой? Поток *блокируется* при отсутствии возможности продолжать выполнение, поскольку приходится чего-то ждать. Обычно это связано с чем-то вроде мьютекса, условной переменной или фьючерса, но причиной может быть и ожидание результатов операции ввода-вывода. Все это вполне естественные составляющие многопоточного кода, которые не всегда признаются желательными, отсюда и формулируется проблема нежелательной блокировки. В связи с этим возникает следующий вопрос: почему такая блокировка становится нежелательной? Как правило, это происходит из-за того, что некий другой поток также ждет освобождения заблокированного потока для выполнения какого-либо действия и оказывается также заблокированным. Существует несколько вариаций на эту тему.

- *Взаимная блокировка* — как следует из материала главы 3, в случае взаимной блокировки один поток чего-то ждет от другого потока, который, в свою очередь, чего-то ждет от первого потока. В наиболее наглядных проявлениях один из потоков, участвующих в этом, отвечает за пользовательский интерфейс, который в таком случае перестает отзываться. В иных случаях интерфейс сохранит отзывчивость, но ряд необходимых задач останется незавершенным, например, не произойдет возвращения из поиска или документ не пойдет на печать.
- *Активная блокировка* — явление, похожее на взаимную блокировку тем, что один поток ждет другого, который, в свою очередь, ждет его самого. Основное отличие здесь в том, что ожидание связано не с блокировкой, а с активным проверочным циклом, например со спин-блокировкой. В тяжелых случаях симптомы такие же, как и у взаимной блокировки (приложение зависает), за исключением высокого потребления ресурсов центрального процессора из-за непрекращающегося выполнения потоков, блокирующих друг друга. В менее тяжких случаях активная блокировка когда-либо завершается по причине произвольного порядка диспетчеризации, но при выполнении задачи, попавшей в ситуацию активной блокировки, произойдет существенная задержка, в ходе которой будет высокий уровень потребления ресурсов центрального процессора.

- *Блокирование на операции ввода-вывода или на ожидании поступления внешних данных* — если поток заблокирован на ожидании поступления внешних данных, он не может продолжить выполнение, даже если данные вообще никогда не поступят. Поэтому нежелательно блокировать внешние данные потока, также выполняющего задачу, завершения которой могут дожидаться другие потоки.

Вот так вкратце можно охарактеризовать нежелательную блокировку. А как насчет состояния гонки?

11.1.2. Состояния гонки

Состояния гонки относятся к наиболее распространенным проблемам, связанным с многопоточным кодом, — многие взаимные и активные блокировки проявляются только из-за состояний гонки. Но проблемными являются не все состояния гонки. Состояние гонки возникает всякий раз, когда поведение зависит от относительной диспетчеризации операций в отдельно взятых потоках. Есть множество абсолютно безвредных состояний гонки. Например, по большому счету совершенно безразлично, какой именно рабочий поток станет работать с очередной задачей из очереди задач. Но с состояниями гонки связаны многие ошибки конкурентности. В частности, состояниями гонки зачастую вызываются следующие разновидности проблем.

- *Гонка за данными*. Это особый тип состояния гонки, приводящий к неопределенному поведению из-за несинхронизированного конкурентного доступа к совместно используемым областям памяти. Гонка за данными была представлена в главе 5 при рассмотрении модели памяти C++. Гонка за данными обычно возникает из-за некорректного использования атомарных операций для синхронизации потоков или из-за обращения к совместно используемым данным без блокировки соответствующего мьютекса.
- *Нарушение инвариантов*. Может проявляться в виде висячих указателей (из-за того, что другой поток 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, когда два потока извлекают из очереди одно и то же значение и затем оба удаляют связанные с ним данными). Нарушения инвариантов могут быть связаны как с несоблюдением временных соотношений, так и с неверными значениями. Если операции в отдельно взятых потоках должны выполняться в определенном порядке, неправильная синхронизация может привести к состоянию гонки, из-за которой требуемый порядок иногда нарушается.
- *Проблемы времени жизни*. Несмотря на то что такие проблемы можно увязать в одно целое с нарушением инвариантов, они все же составляют отдельную категорию. Основная проблема, вызываемая ошибками в данной категории, заключается в том, что поток живет дольше тех данных, к которым обращается, и поэтому обращается к удаленным или же каким-либо образом разрушенным данным и, возможно, к хранилищу, которое даже повторно используется для другого объекта. Обычно проблемы времени жизни возникают, когда поток ссылается на локальные переменные, которые вышли из области видимости еще до завершения

выполнения функции потока, но только этим сценарием они не ограничиваются. Если время жизни потока и время жизни данных, с которыми он работает, каким-то образом не связаны друг с другом, появляется возможность разрушения данных еще до завершения работы потока. Тем самым выбивается почва из-под ног у функции потока. Если самостоятельно вызвать функцию `join()`, чтобы дождаться завершения работы потока, нужно обеспечить невозможность обхода вызова `join()` при выдаче исключения. Применительно к потокам это составляет основу безопасности выдачи исключений.

Проблемные состояния гонки могут иметь самые печальные последствия. При возникновении взаимных и активных блокировок возможно следующее: приложение подвисает и ни на что не реагирует или же тратит слишком много времени на завершение задачи. Зачастую к запущенным процессам можно прикрепить отладчик, чтобы определить потоки, участвующие во взаимной или активной блокировке, и те объекты синхронизации, за которые ведется битва. В случаях гонки за данными, нарушений инвариантов и возникновения проблем времени жизни видимые симптомы (например, внезапные сбои или некорректные выходные данные) могут обнаруживаться в любом месте кода — программа может переписать содержимое памяти, используемой другой частью системы и востребуемой намного позже текущего момента. Сбой в таком случае проявится в той области кода, которая совершенно не связана с местом, содержащим ошибку, и вполне возможно, что он произойдет в ходе выполнения программы спустя значительное время после ошибочно выполненного действия. Это весьма актуально для систем с совместно используемой памятью — сколько бы ни прикладывать усилий по ограничению доступности данных из того или иного потока и по применению действенных механизмов синхронизации, любой поток может переписать данные, используемые любым другим потоком в этом же приложении.

Очертив круг интересующих нас проблем, посмотрим, что можно сделать для обнаружения и исправления имеющихся в коде ошибок.

11.2. Приемы обнаружения ошибок, связанных с конкурентностью

В предыдущем разделе мы рассмотрели типы ошибок, с которыми можно столкнуться при применении конкурентности, и с их возможными проявлениями в вашем коде. Усвоив эту информацию, можно проанализировать свой код на предмет тех мест, где вероятны такие ошибки, и продумать действия по обнаружению этих мест в конкретном разделе кода.

Пожалуй, самый простой способ — *визуальный анализ кода*. Но при всей своей кажущейся простоте провести такой анализ в должной мере весьма непросто. Читая свой же ранее написанный код, проще всего разглядеть свои же намерения при его написании, а не то, что получилось на самом деле. Аналогично этому при изучении кода, написанного кем-то другим, мы стремимся быстро просмотреть его, проверяя на соответствие местным стандартам программирования. Отмечаем при этом наиболее очевидные проблемные места. Нужно не пожалеть времени на прочесывание

кода мелким гребнем, чтобы обнаружить проблемы, связанные с конкурентностью, а заодно и те, что не связаны с ними. (Все это можно совместить, поскольку устранению подлежат любые ошибки.) На что конкретно нужно будет обратить внимание, станет понятно из предстоящего вскоре анализа кода.

Но даже после тщательного исследования кода некоторые ошибки можно и проглядеть. В любом случае для своего же спокойствия не остается ничего иного, как подтвердить работоспособность этого кода. Следовательно, продолжая рассматривать вопросы визуального исследования кода, мы перейдем к изучению приемов, применяемых при тестировании многопоточных программ.

11.2.1. Просмотр кода с целью поиска возможных ошибок

Как уже упоминалось, поиск ошибок, связанных с конкурентностью, при визуальном просмотре многопоточного кода требует особой тщательности. По возможности нужно привлечь к этой работе и других специалистов. Поскольку они не имеют отношения к созданию этого кода, станут разбираться в том, как он работает, что поможет обнаружить любые ошибки в коде. Важно, чтобы у них было достаточно времени для качественного изучения кода, а не пара минут, чтобы взглянуть вскользь. Здесь нужен серьезный, вдумчивый анализ. Многие ошибки, связанные с конкурентностью, не бросаются в глаза при беглом просмотре, их выявление требует кропотливого изучения кода.

У привлекаемых к просмотру кода коллег будет свежий взгляд. Они будут на все смотреть с другой позиции и смогут заметить то, что ускользает от вашего взгляда. Если таких коллег у вас нет, попросите друзей или даже просто опубликуйте код в Интернете (постаравшись при этом не расстраивать юристов вашей компании). Если привлечь к просмотру кода так никого и не удастся или привлеченные специалисты ничего не найдут, не стоит огорчаться, еще не все потеряно. Для начала было бы полезно на некоторое время оставить код в покое и поработать над другой частью приложения, почитать книгу или отправиться на прогулку. Если сделать перерыв, то подсознание может работать над решением проблем в фоновом режиме, пока будет сосредоточено на чем-то другом. Кроме того, код, когда вы к нему вернетесь, утратит узнаваемость и вы посмотрите на него свежим взглядом.

Можно и не привлекать для просмотра сторонних специалистов, обойдясь собственными силами. Один из эффективных приемов — попробовать подробно объяснить принципы его работы каким-нибудь благодарным слушателям. Ими могут быть даже неодушевленные предметы, у многих команд для этого есть игрушечные медвежата или цыплята. Лично я считаю чрезвычайно полезным написание подробных пояснений к коду. Объясняя предназначение кода, следует уделять внимание каждой строке, тому, что должно произойти, к каким данным происходит обращение и т. д. Задайте самим себе вопросы по коду и дайте на них развернутые ответы. Я считаю такое занятие весьма эффективным приемом — задавая самому себе вопросы и тщательно обдумывая ответы, зачастую удается выявить наличие той или иной проблемы. Польза от таких вопросов проявится не только при просмотре собственного кода, но и при изучении любого другого кода.

Вопросы, требующие осмысления при просмотре многопоточного кода

Как уже упоминалось, рецензенту (автору кода или стороннему специалисту) следует задуматься о конкретных вопросах применительно к просматриваемому коду. Эти вопросы помогут сконцентрировать внимание рецензента на соответствующих особенностях кода и помочь выявлению потенциальных проблем. Ниже приведу перечень вопросов, которые я привык задавать, хотя он, скорее всего, не исчерпывающий. Вы можете поднять другие вопросы, которые помогут лучше сконцентрироваться на поиске ошибок. А вот так выглядят мои вопросы.

- Какие данные нуждаются в защите от конкурентного доступа?
- Как можно обеспечить защищенность данных?
- Какие фрагменты кода будут выполняться в это же время в других потоках?
- Какие мьютексы будут удерживаться анализируемым потоком?
- Какие мьютексы могут удерживаться другими потоками?
- Существуют ли какие-нибудь требования к порядку выполнения операций в анализируемом потоке, а также в других потоках? Как эти требования выполняются?
- Являются ли данные, загружаемые анализируемым потоком, по-прежнему действительными? Могли ли они быть изменены другими потоками?
- Если предположить, что другой поток мог изменить данные, то какое бы это имело значение и как можно было бы гарантировать, что такое никогда не случится?

Последний вопрос — мой любимый, поскольку он заставляет меня задуматься о взаимоотношениях между потоками. Предполагая наличие ошибки, связанной с конкретной строкой кода, можно далее выступить в роли сыщика и отследить причину. Чтобы убедиться в отсутствии ошибок, нужно рассмотреть каждый крайний случай и возможный порядок выполнения операций. Это особенно актуально при защите данных за время их жизни более чем одним мьютексом, как в случае с потокобезопасной очередью из главы 6, где для головы и хвоста очереди имелись отдельные мьютексы: чтобы обеспечить безопасность обращения при удержании одного мьютекса, нужно быть уверенными, что поток, удерживающий *другой* мьютекс, не может обращаться к тому же элементу. Также вполне очевидно, что особому анализу должны подвергаться общедоступные данные или те данные, на которые другой код может легко получить указатель или ссылку.

Важен и предпоследний вопрос приведенного выше перечня, поскольку он касается весьма распространенной и часто допускаемой ошибки: если освободить, а затем снова получить мьютекс, следует предполагать, что совместно используемые данные могли претерпеть изменения со стороны других потоков. При всей очевидности такого предположения, если блокировки мьютексов не лежат на поверхности (возможно, из-за того, что они выполняются внутри объекта), неосознанно допускается именно такая ошибка. В главе 6 мы продемонстрировали, как излишняя детализация обращения к данным в функциях, предоставляемых потокобезопасной структуре данных, может приводить к состоянию гонки. Если для стека, не являющегося потокобезопасным, наличие отдельных операций `top()` и `pop()` имеет

вполне определенный смысл, то для стека, допускающего конкурентное обращение со стороны сразу нескольких потоков, такой вариант уже не подходит, поскольку блокировка на внутреннем мьютексе между двумя вызовами освобождается, позволяя вносить изменения со стороны другого потока. Как вы видели в главе 6, решение заключается в объединении двух операций, чтобы обе они выполнялись под защитой блокировки одного и того же мьютекса, не позволяя возникать состоянию гонки.

Итак, вы (или кто-то другой) просмотрели код. Вы уверены, что ошибок нет. Но, как говорится, распробовать пудинг можно только в еде, так же нужно и протестировать код, чтобы убедиться в отсутствии ошибок.

11.2.2. Обнаружение ошибок, связанных с конкурентностью, путем тестирования

При разработке однопоточных приложений процедура их тестирования при наличии достаточного времени относительно проста. В принципе, можно определить все возможные наборы входных данных (или по крайней мере те, что представляют интерес) и прогнать их через приложение. Если приложение демонстрирует правильное поведение и производит верные выходные данные, то возникает уверенность в его работоспособности с этим набором входной информации. Тестирование на выход из состояний ошибок, например на обработку ошибок переполнения диска, дается сложнее, но замысел прежний: установка исходных условий и предоставление приложению возможности справиться с ними.

Тестирование многопоточного кода на порядок сложнее, поскольку точный порядок диспетчеризации потоков определению не поддается и от запуска к запуску может быть разным. Следовательно, даже если запускать приложение с одними и теми же входными данными, то порой оно может работать правильно, а порой, при наличии скрытых в коде предпосылок к состояниям гонки, давать сбои. Наличие таких предпосылок означает, что сбой в коде будет появляться не всегда, а только время от времени.

Учитывая сложность воспроизводства ошибок, связанных с конкурентностью, тесты требуют тщательной проработки. Нужно, чтобы в каждом тесте запускался наименьший объем кода, в котором может проявиться ошибка. Так можно будет наилучшим образом изолировать сбойный код, если тест не будет пройден — лучше провести непосредственное тестирование одновременно используемой очереди, чтобы проверить, что одновременное помещение в нее данных и их одновременное извлечение работают, а не тестировать ее в контексте всего кода, где она применяется. При разработке кода полезно задуматься и о том, как он будет тестироваться, разработку с прицелом на тестируемость рассмотрим в этой главе чуть позже.

Чтобы проверить, связана ли проблема именно с конкурентностью, стоит убрать из теста все, что с ними связано. Если проблема проявляется при запуске всего кода в одном потоке, значит, причина кроется в самой банальной ошибке, а не в ошибке, связанной с конкурентностью. Это особенно важно при попытке отслеживания той ошибки, что проявляется при обычных прогонах программы, а не в хитросплетениях используемых тестов. Лишь то, что ошибка проявляется в многопоточной части вашего приложения, еще не означает, что она автоматически становится связанной

с конкурентностью. Если для управления уровнями конкурентности используются пулы потоков, то обычно для указания количества рабочих потоков можно воспользоваться параметром конфигурации. Если потоки управляются не в автоматическом режиме, то для использования в тесте одного потока придется вносить изменения в код. В любом случае, если сократить фронт выполнения приложения до одного потока, появится возможность устранить конкурентность как причину возникновения ошибки. С другой стороны, проблема исчезает в *одноядерной системе* (даже при запуске нескольких потоков), но появляется в *многоядерных* или *многопроцессорных системах*. В этом случае речь может идти о состоянии гонки или же проблемах синхронизации либо порядка доступа к памяти.

При тестировании конкурентного кода важна не только структура тестируемого кода, но и структура теста и среды тестирования. Если продолжить разбор примера тестирования конкурентной очереди, то следует продумать целый ряд различных сценариев.

- Один поток автономно вызывает `push()` или `pop()` с целью проверки работоспособности очереди на базовом уровне.
- Один поток вызывает `push()` в отношении пустой очереди, в то время как другой поток вызывает `pop()`.
- Несколько потоков вызывают `push()` в отношении пустой очереди.
- Несколько потоков вызывают `push()` в отношении заполненной очереди.
- Несколько потоков вызывают `pop()` в отношении пустой очереди.
- Несколько потоков вызывают `pop()` в отношении заполненной очереди.
- Несколько потоков вызывают `pop()` в отношении частично заполненной очереди с недостаточным количеством элементов для всех потоков.
- Несколько потоков вызывают `push()`, в то время как один поток вызывает `pop()` в отношении пустой очереди.
- Несколько потоков вызывают `push()`, в то время как один поток вызывает `pop()` в отношении заполненной очереди.
- Несколько потоков вызывают `push()`, в то время как несколько потоков вызывают `pop()` в отношении пустой очереди.
- Несколько потоков вызывают `push()`, в то время как несколько потоков вызывают `pop()` в отношении заполненной очереди.

Обдумав все эти сценарии (и не только их), нужно рассмотреть дополнительные факторы, касающиеся среды тестирования.

- Что подразумевается под «несколькими потоками» в каждом случае (это 3, 4, 1024 потока)?
- Достаточно ли вычислительных ядер в системе, чтобы запустить каждый поток в своем собственном ядре?
- На каких процессорных архитектурах должны запускаться тесты?
- Как обеспечить подходящий режим диспетчеризации для той части тестов, которая обозначена фразой «в то время как»?

Применительно к вашей ситуации следует обдумать и дополнительные факторы. Из четырех приведенных выше аспектов среды первый и последний относятся к структуре самого теста (и рассматриваются в подразделе 11.2.5), а другие два относятся к той физической системе, которая используется для тестирования. Количество используемых потоков зависит от конкретного тестируемого кода, но существуют различные способы структурирования теста для получения подходящего порядка диспетчеризации. Прежде чем приступить к рассмотрению этих способов, давайте посмотрим, как можно сконструировать код приложения, чтобы его легче было тестировать.

11.2.3. Разработка кода с прицелом на удобство тестирования

Тестирование многопоточного кода дается нелегко, поэтому возникает желание облегчить эту задачу. Самое важное из возможных действий в этом направлении — разработка кода с прицелом на удобство его тестирования. Существует множество публикаций применительно к разработке такого кода для однопоточной среды, и многие приводимые там советы не потеряли своей актуальности и в нашем случае. В общем, код проще поддается тестированию, если соблюдены следующие факторы:

- четкое определение сфер ответственности каждой функции и класса;
- краткость и конкретность определения функций;
- возможность установить со стороны тестов полный контроль над средой окружения тестируемого кода;
- сосредоточенность в одном месте, а не разбросанность по всей системе кода операции, подвергаемой тестированию;
- продумывание порядка тестирования кода до его написания.

Все эти факторы актуальны и для многопоточного кода. В сущности, можно сказать, что обращать внимание на пригодность к тестированию многопоточного кода еще важнее, чем однопоточного, поскольку его по определению гораздо сложнее тестировать. Также важен и последний фактор: даже если вы еще не дошли до того, чтобы создавать тесты до написания самого тестируемого кода, то, прежде чем приступать к написанию кода, стоит подумать о том, как можно будет его протестировать: какие входные данные использовать, при каких условиях, скорее всего, станут возникать проблемы, как можно спровоцировать код на проявление проблем и т. д.

Суть одного из лучших способов создания конкурентного кода с прицелом на тестирование заключается в исключении конкурентности. Если код можно разбить на части, которые отвечают за пути передачи данных между потоками, и на части, которые работают с передаваемыми данными в рамках одного потока, то задача существенно упростится. Те части приложения, что работают с данными, обращение к которым бывает только из одного потока, теперь можно протестировать с использованием обычных однопоточных приемов тестирования. Сложный для тестирования конкурентный код, занимающийся обменом данными между потоками

и обеспечивающий доступность к конкретному блоку данных в любой момент со стороны только одного потока, теперь стал намного меньше и легче поддающимся тестированию.

Например, если приложение разработано в качестве многопоточного конечного автомата, его можно разбить на несколько частей. Логику состояния для любого потока, обеспечивающего для каждого набора входных событий корректность переходов и операций, можно протестировать независимо и с применением однопоточных приемов, с помощью тестового набора, предоставляющего входные события, которые будут поступать от других потоков. Затем можно независимо протестировать основной код конечного автомата и код маршрутизации сообщений, обеспечивающий правильную доставку событий нужным потокам и в нужном порядке, но с несколькими конкурентными потоками и с простой логикой состояния, специально разработанной для тестов.

Или же, если есть возможность разбить код на несколько блоков — *чтения совместно используемых данных/преобразования данных/обновления совместно используемых данных*, — можно будет протестировать ту часть кода, которая занимается преобразованием данных, воспользовавшись всеми обычными однопоточными приемами, поскольку теперь это однопоточный код. Сложная задача тестирования многопоточных преобразований теперь сведется к тестированию чтения и обновлению совместно используемых данных, что представляется куда более простой задачей.

Обратите внимание, что в библиотечных вызовах для хранения состояния допускается применение внутренних переменных, которые затем станут использоваться совместно, если один и тот же набор библиотечных вызовов будет задействован сразу несколькими потоками. Это может стать проблемой, потому что факт обращения кода к совместно используемым данным вскроется не сразу. Но со временем станет понятно, какие это библиотечные вызовы, они будут сразу бросаться в глаза. Затем можно либо добавить соответствующую защиту и синхронизацию, либо воспользоваться альтернативной функцией, не представляющей опасности при конкурентном доступе из нескольких потоков.

Разработка легко поддающегося тестированию многопоточного кода не ограничивается структурированием кода с целью минимизации объема кода, необходимого для решения вопросов, связанных с конкурентностью, и контролем использования библиотечных вызовов, не обладающих потокобезопасностью. Полезно помнить о вопросах из подраздела 11.2.1, которые следует задавать самим себе при просмотре кода. Хотя эти вопросы и не относятся напрямую к тестированию и готовности кода к тестированию, но, если переосмыслить вопросы с прицелом на тестирование и подумать о способах тестирования кода, ответы на них окажут благоприятное влияние на решения в выборе конструкции, что облегчит тестирование.

Теперь, когда рассмотрены вопросы разработки кода с прицелом на облегчение тестирования и потенциально модифицированный код для отделения «конкурентных» частей (например, потокобезопасных контейнеров или логики событий конечного автомата) от «однопоточных» частей (которые по-прежнему могут взаимодействовать с другими потоками при содействии конкурентных частей кода), давайте посмотрим на приемы тестирования конкурентного кода.

11.2.4. Приемы тестирования многопоточного кода

Итак, сценарий тестирования продуман, создан небольшой фрагмент кода для испытания тестируемых функций. И как теперь добиться выполнения любых потенциально проблематичных последовательностей диспетчеризации с целью избавления от ошибок?

Способов здесь несколько, и для начала рассмотрим принудительное тестирование, или стресс-тестирование.

Стресс-тестирование

Принудительное тестирование предусматривает создание для кода стрессовой обстановки с целью его испытания на прочность. Обычно под этим подразумевается многократный запуск кода, возможно, с многочисленными одновременно запускаемыми потоками. Если есть ошибка, проявляющаяся только при определенном порядке диспетчеризации потоков, то чем больше запусков кода, тем выше вероятность проявления ошибки. Когда тест запускается один раз и заканчивается успешным проходом, может появиться уверенность в работоспособности кода. Если его прогнать подряд десять раз и всякий раз успешно, уверенность возрастет. При успешном прогоне теста миллиард раз уверенность станет еще крепче.

Возникающая в результате тестирования уверенность не зависит от объема кода, прогоняемого через каждый тест. Если детализация тестирования достаточно высока, вроде тех тестов, которые упоминались ранее в отношении потокобезопасной очереди, такое стресс-тестирование может дать вам высокую степень уверенности в работоспособности кода. С другой стороны, если тестируемый код значительно больше по объему, количество возможных перестановок при диспетчеризации потоков настолько велико, что даже миллиардное число тестовых прогонов не поможет избавиться от низкого уровня уверенности в работоспособности кода.

Недостаток стресс-тестирования заключается в том, что оно может дать ложную уверенность в работоспособности кода. Если при создании теста исходить из того, что проблемные обстоятельства никогда не возникнут, можно выполнять прогон теста вплоть до бесконечности и он не будет давать сбои, даже если неизменно будет сбоить при слегка измененных обстоятельствах. Наихудший вариант такого развития событий возникает, когда система, на которой производится тестирование, настроена так, что проблематичные условия в принципе невозможны. Так 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 системе. Поскольку каждый поток приходится запускать на одном и том же процессоре, весь код подвергается автоматической сериализации, и многие проблемы, связанные с состояниями гонки и перебрасыванием данных в кэш-памяти, с которыми можно столкнуться в многопроцессорной системе, просто исчезают. Но это еще не все, процессоры с разной архитектурой предоставляют различные средства диспетчеризации и упорядочения доступа к памяти. Например, на архитектурах x86 и x86-64 атомарные операции загрузки всегда одинако-

вы, независимо от того, какая семантика применена — `memory_order_relaxed` или `memory_order_seq_cst` (см. подраздел 5.3.3). Это означает, что код, написанный с расслабленной семантикой упорядочения доступа к памяти, может работать на архитектуре x86, а вот в системах с более детализированным набором инструкций упорядочения доступа к памяти, например на SPARC, он даст сбой.

Если от приложения требуется переносимость в широком диапазоне целевых систем, важно будет протестировать его на показательных экземплярах таких систем. Именно поэтому в подразделе 11.2.2 в перечень учитываемых факторов включена архитектура процессора, используемого для тестирования.

Для успешного стресс-тестирования очень важно избавиться от ложной уверенности. Для этого следует тщательно продумать конструкцию теста не только в плане выбора тестируемого блока кода, но и в отношении конструкции средства тестирования и выбора среды тестирования. Нужно обеспечить тестирование максимально возможного количества путей выполнения кода и моментов взаимодействия потоков. И это еще не все. Следует знать, какие варианты уже охвачены тестированием, а какие — еще нет.

Стресс-тестирование придает некую степень уверенности в работоспособности кода, но оно не гарантирует выявления всех проблем. Если есть время и соответствующие программные средства, то для поиска проблем к коду можно применить еще один прием. Я называю его *комбинационным имитационным тестированием*.

Комбинационное имитационное тестирование

Название воспринимается с трудом, поэтому поясню, что имеется в виду. Идея в том, чтобы запускать тестируемый код со специальным фрагментом программы, *имитирующим* прогон в реальных условиях среды применения. Возможно, вам известны программные средства, позволяющие запускать на одном физическом компьютере сразу несколько виртуальных машин, где характеристики виртуальной машины и ее оборудования имитируются управляющей программой. Здесь замысел тот же, только вместо эмуляции системы программа имитации записывает последовательность обращения к данным, блокировок и атомарных операций из каждого потока. Затем она применяет правила модели памяти C++ для повторения прогона с каждой допустимой *комбинацией* операций и выявления состояний гонки и взаимных блокировок.

Хотя такое исчерпывающее комбинационное тестирование гарантирует обнаружение всех проблем, на которые нацелена конструкция системы, для любых, кроме самых простых, программ оно займет уйму времени, поскольку количество комбинаций с увеличением числа потоков и числа операций, выполняемых каждым потоком, растет в геометрической прогрессии. Этот прием лучше приберечь для более детализированного тестирования отдельных частей кода, а не для всего приложения. Еще один явный недостаток — зависимость этого приема от доступности программы имитации, способной управлять операциями, используемыми в вашем коде.

Итак, есть прием, предполагающий многократный прогон теста при обычных условиях, но способный проглядеть проблему, и есть прием, предполагающий многократный прогон теста при особых условиях, но с высокой вероятностью обнаружения всех имеющихся проблем. А существуют ли другие варианты?

Третий вариант предполагает использование библиотеки, обнаруживающей проблемы по мере их проявления при прогоне теста.

Обнаружение проблем, проявляющихся при прогоне теста с использованием специализированной библиотеки

Хотя этот вариант не предоставляет исчерпывающей проверки, присущей комбинационному имитационному тестированию, специализированная реализация библиотечных элементов синхронизации, включая мьютексы, блокировки и условные переменные, позволяет выявить многие проблемы. Например, зачастую требуется, чтобы все обращения к области совместно используемых данных осуществлялись с блокировкой конкретного мьютекса. Если можно проверить, какие мьютексы были заблокированы при обращении к данным, появится возможность проконтролировать факт блокировки соответствующего мьютекса путем вызова потока при обращении к данным и отправки сообщения о сбое, если этот факт не подтвердится. Помечая каким-то образом совместно используемые данные, можно позволить библиотеке провести для вас такую проверку.

Эта реализация библиотеки может также записывать последовательность блокировок, если конкретным потоком одновременно удерживается сразу несколько мьютексов. Если другой поток блокирует те же самые мьютексы в ином порядке, это может быть зарегистрировано как *потенциальная* взаимная блокировка, даже если тест при прогоне с ней не столкнулся.

Еще в одном типе специализированной библиотеки, применяемой при тестировании многопоточного кода, реализация элементов потока, таких как мьютексы и условные переменные, предоставляет создателю тестов возможность брать контроль над тем, какому потоку получать блокировку, когда в режиме ожидания находятся сразу несколько потоков или какой поток уведомляется путем вызова функции `notify_one()` в отношении условной переменной. Это позволит задавать конкретные сценарии и проверять, оправдывает ли работа кода по таким сценариям возложенные на него ожидания.

Некоторые из рассмотренных средств тестирования следовало бы поставлять как часть стандартной библиотеки C++, а остальные можно создать в качестве надстройки над стандартной библиотекой в виде части вашей конструкции средства тестирования.

Рассмотрев различные приемы выполнения тестового кода, перейдем к способам структурирования кода для получения желаемого порядка диспетчеризации потоков.

11.2.5. Структурирование многопоточного тестового кода

В подразделе 11.2.2 говорилось о необходимости поиска способов предоставить подходящую диспетчеризацию потоков для той части тестов, которая шла за фразой «в то время как». Настало время рассмотреть вопросы, связанные с решением этой задачи.

Прежде всего нужно решить вопрос с организацией набора потоков, каждый из которых будет выполнять выбранный фрагмент кода в указанный момент времени. В общем случае речь идет о двух потоках, но все можно легко расширить

и на большее число потоков. На первом этапе следует разбить каждый тест на отдельные части:

- код общей настройки, который должен выполняться в самом начале;
- потоковый код настройки, который должен исполняться в каждом потоке;
- код, запускаемый в конкурентных потоках;
- код, предназначенный для запуска после завершения конкурентного выполнения, который может включать утверждения о состоянии кода.

В целях пояснения рассмотрим конкретный пример из перечня сценариев тестирования, приведенного в подразделе 11.2.2: один поток вызывает функцию `push()` в отношении пустой очереди, в то время как другой поток вызывает функцию `pop()`.

Общий код настройки прост: нужно создать очередь. У потока, выполняющего функцию `pop()` кода, настройки применительно к потоку нет. Код настройки применительно к потоку, выполняющему функцию `push()`, зависит от интерфейса очереди и типа сохраняемого в ней объекта. Если сохраняемый объект трудоемок в создании или требует размещения в куче, то всю предварительную подготовку желательно провести в виде части настройки применительно к потоку, чтобы не оказывать негативного влияния на тест. С другой стороны, если в очереди просто сохраняются обычные `int`-объекты, то от создания `int`-значений в коде настройки мы ничего не выиграем. Тестируемый код относительно прост — вызов `push()` из одного потока и вызов `pop()` из другого потока, ну а как насчет кода «после завершения»?

В данном случае все зависит от того, что требуется получить от функции `pop()`. Если предполагается блокировка до появления в очереди данных, то абсолютно ясно: хочется увидеть, что возвращаются те самые данные, которые были предоставлены вызову функции `push()`, после чего очередь становится пустой. Если функция `pop()` не блокируется и может завершиться, даже если очередь пуста, нужно протестировать две возможности: либо `pop()` возвращает элемент данных, предоставленный функции `push()`, и очередь становится пустой, либо `pop()` сигнализирует об отсутствии данных в очереди, а в очереди имеется один элемент. Должно произойти либо одно, либо другое; хотелось бы избежать сценария, при котором `pop()` сигнализирует, что «нет данных», а очередь пуста, либо сценария, при котором `pop()` возвращает значение, а очередь все же не пуста. Чтобы упростить тест, предположим, что мы имеем дело с блокирующей функцией `pop()`. В таком случае в завершающем коде должно быть утверждение, что извлеченное значение является тем же самым, что и помещенное значение, и очередь пуста.

После определения различных частей кода нужно приложить все усилия для обеспечения их запуска по плану. Для этого можно воспользоваться набором промисов `std::promise`, показывающих момент готовности всех частей. Каждый поток выдает промис, чтобы показать, что он готов, а затем ждет на фьючерсе `std::shared_future` (вернее, на его копии), полученном из третьего промиса `std::promise`, а основной поток ждет выдачи промисов от всех потоков и затем запускает потоки на выполнение с помощью сигнала `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 промиса `go`. Наконец, главный поток ждет завершения других потоков

и проверяет получившееся состояние. И наконец, основной поток ждет, пока потоки завершат свою работу, и проверяет конечное состояние. Следует также не забывать об исключениях и убедиться в отсутствии потоков, ожидающих сигнала go, выдачи которого не будет. Один из путей структурирования этого теста показан в листинге 11.1.

Листинг 11.1. Пример теста для конкурентных вызовов push() и pop() в отношении очереди

```
void test_concurrent_push_and_pop_on_empty_queue()
{
    threadsafe_queue<int> q;                                    ❶
❷   std::promise<void> go,push_ready,pop_ready;
    std::shared_future<void> ready(go.get_future());            ❸
    std::future<void> push_done;
    std::future<int> pop_done;                                  ❹
    try
    {
        push_done=std::async(std::launch::async,                ❺
                             [&q,ready,&push_ready]()
                             {
                                 push_ready.set_value();
                                 ready.wait();
                                 q.push(42);
                             }
            );
        pop_done=std::async(std::launch::async,                 ❻
                             [&q,ready,&pop_ready]()
                             {
                                 pop_ready.set_value();
                                 ready.wait();
                                 return q.pop();                ❼
                             }
            );
        push_ready.get_future().wait();                         ❽
        pop_ready.get_future().wait();
❾       go.set_value();
        push_done.get();                                        ❿
⓫       assert(pop_done.get()==42);
        assert(q.empty());
    }
    catch(...)
    {
        go.set_value();                                         ⓬
        throw;
    }
}
```

Структура в значительной степени совпадает с ранее рассмотренной. Сначала как часть общей настройки создается пустая очередь ❶. Затем создаются все промисы для сигналов «готовности» ❷ и берется фьючерс `std::shared_future` для сигнала `go` ❸. После этого создается фьючерс, который будет использоваться для обозначения завершения работы потоков ❹. Это нужно сделать вне блока `try`, чтобы можно было установить сигнал `go` на исключение без ожидания завершения работы тестовых потоков (что привело бы к взаимной блокировке, которая в коде теста весьма нежелательна).

Затем внутри блока `try` можно запустить потоки (❺ и ❻) — использование `std::launch::async` гарантирует, что каждая задача выполняется в своем собственном потоке. Заметьте, что применение `std::async` облегчает задачу обеспечения безопасности выдачи исключений по сравнению с использованием простого объекта `std::thread`, поскольку деструктор для фьючерса присоединится к потоку. Захваты в лямбда-функциях указывают на то, что каждая задача для подачи сигнала о готовности будет ссылаться на очередь и соответствующий промис, забирая копию фьючерса `ready`, полученную из промиса `go`.

В соответствии с предыдущим описанием каждой задачей устанавливается свой собственный сигнал `ready`, после чего перед запуском тестового кода ожидается поступление общего сигнала `ready`. Основной поток делает все наоборот — ждет сигналов от обоих потоков ❽, прежде чем подать им сигнал на запуск настоящего теста ❾.

И наконец, основной поток вызывает функцию `get()` в отношении фьючерсов из асинхронных вызовов, чтобы дождаться завершения задач, ❿ и ⓫, и проверяет результаты. Следует отметить, что операция `pop` извлекает возвращаемое значение из фьючерса ❼, чем можно воспользоваться для получения результата для утверждения ⓫.

Если выдается исключение, устанавливается сигнал `go`, чтобы исключить любую возможность получения висячего потока, и происходит повторная выдача исключения ⓬. Фьючерсы, соответствующие задачам ❹, были объявлены последними, поэтому они будут уничтожены первыми, а их деструкторы будут ждать завершения задач, если этого еще не произошло.

Несмотря на создающееся впечатление, что для тестирования двух простых вызовов здесь слишком много шаблонного кода, чтобы рассчитывать на успешное тестирование нужного вам кода, необходимо воспользоваться чем-то подобным. Например, запуск потока может занять слишком много времени, поэтому если не заставить потоки ждать сигнала `go`, то поток, помещающий данные, может завершиться еще до запуска потока, извлекающего данные, что полностью лишит тест всякого смысла. Данный способ использования фьючерсов гарантирует, что оба потока запускаются и блокируются на одном и том же фьючерсе. Затем разблокировка фьючерса позволит потокам приступить к выполнению. После того как эта структура будет освоена, задача создания новых тестов по той же схеме не составит особого труда. Для тестов, требующих использования более двух потоков, эта схема допускает расширение и на дополнительные потоки.

До сих пор мы говорили лишь о *корректности* многопоточного кода. Это, конечно, наиболее важный, но не единственный вопрос, ради которого проводится тестирование. Важно также протестировать *производительность* многопоточного кода, о чем пойдет речь далее.

11.2.6. Тестирование производительности многопоточного кода

Одной из основных причин, по которым для приложения делается выбор в пользу применения конкурентности, является возможность повышения производительности приложений в условиях роста распространенности многоядерных процессоров. Поэтому важно протестировать создаваемый вами код на подтверждение факта роста

его производительности, как это делалось при любых других попытках проведения оптимизации кода.

Особый интерес при использовании конкурентности для повышения производительности представляет вопрос *масштабируемости* — вам нужен код, который при прочих равных условиях на 24-ядерной машине работает примерно в 24 раза быстрее или обрабатывает в 24 раза больше данных, чем на одноядерной машине. Вам не нужен код, работающий вдвое быстрее на двухъядерной машине, но медленнее на 24-ядерной машине. В подразделе 8.4.2 было показано, что если значительная часть кода запускается только в одном потоке, то возможный прирост производительности будет ограничен. Поэтому перед началом тестирования стоит изучить общую конструкцию кода, чтобы понять, стоит ли надеяться на 24-кратное увеличение производительности, или последовательно выполняемая часть этого кода предопределяет ограничение увеличения максимум до трехкратного.

В предыдущих главах уже говорилось, что конкуренция процессоров за доступ к структуре данных может серьезно повлиять на производительность. То, что прекрасно масштабируется на малое количество процессоров, может утратить производительность при существенном росте их числа из-за резкого повышения уровня конкуренции.

Соответственно, при тестировании многопоточного кода на предмет его производительности лучше проверить производительность в как можно большем количестве систем с разной конфигурацией. Это позволит получить целостное представление о масштабируемости. По крайней мере следует прогнать тесты на однопроцессорном компьютере и на доступной вам машине с максимальным количеством процессоров.

Резюме

В этой главе мы рассмотрели различные типы ошибок, с которыми можно столкнуться при использовании конкурентности, — от взаимных и активных блокировок до состояний гонки за данными и другими проблематичными состояниями гонки. Были описаны приемы обнаружения ошибок, а также факторы, требующие осмысления при визуальном анализе кода. Я дал рекомендации по разработке кода с прицелом на удобство тестирования и предложил несколько приемов структуризации тестов для конкурентного кода. И в заключение мы рассмотрели некоторые полезные компоненты, способные помочь в процессе тестирования.

Приложения

A Краткий справочник по некоторым функциям языка C++11

В новый стандарт C++ привнесена не только поддержка конкурентности; в нем также есть множество других функций языка и новых библиотек. В этом приложении дадим краткий обзор новых функций языка, используемых в Thread Library и в остальной части книги. Кроме модификатора `thread_local` (рассматриваемого в разделе A.8), ни одна из них не имеет непосредственной связи с конкурентностью. Тем не менее они важны и (или) полезны для создания многопоточного кода. Их перечень я ограничил теми функциями, которые либо необходимы (как, например, ссылки на *r*-значения), либо упрощают код или делают его понятнее. Поначалу возможны затруднения с пониманием кода, использующего эти функции, из-за его новизны, но по мере ознакомления с ним можно будет согласиться с тем, что в целом эти функции делают код понятнее. По мере роста востребованности C++11 код, использующий эти функции, будет встречаться все чаще.

А теперь без лишних предисловий приступим к рассмотрению ссылок на *r*-значения, которые широко применяются Thread Library для облегчения передачи прав владения (потоками, блокировками или чем-либо еще) между объектами.

A.1. Ссылки на r-значения

Каждый, кто когда-либо сталкивался с программированием на C++, знаком со ссылками. Ссылки C++ позволяют создавать новые имена для существующих объектов. Все обращения или изменения, совершаемые по новой ссылке, воздействуют на исходный объект, например:

```
int var=42;
int& ref=var;
ref=99;
assert(var==99);
```

Создание ссылки на var

Исходное значение обновлено в результате присваивания нового значения по ссылке

Единственными ссылками до появления C++11 были *ссылки на l-значения*. Понятие *l*-значения пришло из языка C и относится к тому, что может быть слева от знака присваивания, к поименованным объектам, объектам, размещаемым в стеке или в куче, или к компонентам других объектов, то есть ко всему, имеющему вполне определенное место хранения. Понятие *r*-значение также пришло из языка C и относится к тому, что может появляться только справа от знака присваивания — к примеру, к литералам и временным значениям. Ссылки на *l*-значения могут быть привязаны только к *l*-, но не к *r*-значениям. К примеру, нельзя написать

```
int& i=42;
```
← Не пройдет компиляцию

потому что `42` является *r*-значением. Это верно лишь отчасти, поскольку всегда можно привязать *r*-значение к константной ссылке на *l*-значение:

```
int const& i=42;
```

Но это исключение в часть стандарта было введено преднамеренно до появления ссылок на *r*-значения, чтобы дать возможность передавать временные значения функциям, принимающим ссылки. Тем самым разрешаются подразумеваемые преобразования, позволяющие написать следующий код:

```
void print(std::string const& s);
print("hello");
```
← Создание временного объекта std::string

В стандарте C++11 появились *ссылки на r-значения*, которые привязываются *только* к *r*-, но не к *l*-значениям и объявляются с использованием не одного, а двух амперсандов:

```
int&& i=42;
int j=42;
int&& k=j;
```
← Не пройдет компиляцию

Чтобы определить, являются ли параметры функции *l*- или *r*-значениями, можно воспользоваться переопределением функции, и одно ее определение будет принимать ссылку на *l*-значение, а другое — ссылку на *r*-значение. Это станет краеугольным камнем *семантики перемещений*.

A.1.1. Семантика перемещений

r-значения обычно представлены временными объектами, поэтому в них свободно можно вносить изменения. Если известно, что параметром вашей функции является *r*-значение, им можно воспользоваться в качестве временного хранилища или «украсть» его содержимое без ущерба для корректности программы. Это означает, что вместо *копирования* содержимого параметра, являющегося *r*-значением, можно *переместить* его содержимое. Для крупных динамических структур это позволит сэкономить на многократном распределении памяти и предоставит широкие возможности для их оптимизации. Рассмотрим функцию, получающую в качестве параметра `std::vector<int>` и нуждающуюся во внутренней копии для изменения,

не затрагивая при этом оригинал. Раньше для этого нужно было принять параметр как const-ссылку на *l*-значение и сделать внутреннюю копию:

```
void process_copy(std::vector<int> const& vec_)
{
    std::vector<int> vec(vec_);
    vec.push_back(42);
}
```

Это позволяет функции принимать как *l*-, так и *r*-значения, но в любом случае вынуждает выполнять копирование. Если переопределить функцию с версией, принимающей ссылку на *r*-значение, то в случае с *r*-значением можно избежать копирования, поскольку известно, что в оригинал разрешается свободно вносить изменения:

```
void process_copy(std::vector<int> && vec)
{
    vec.push_back(42);
}
```

Теперь, если рассматриваемая функция является конструктором вашего класса, можно будет «украсть» внутреннее содержимое *r*-значения и воспользоваться им для ваших новых экземпляров. Рассмотрим класс, показанный в листинге А.1. В конструкторе по умолчанию он распределяет весьма крупный блок памяти, высвобождаемый в деструкторе.

Листинг А.1. Класс с конструктором перемещения

```
class X
{
private:
    int* data;
public:
    X():
        data(new int[1000000])
    {}
    ~X()
    {
        delete [] data;
    }
    X(const X& other):                    ❶
        data(new int[1000000])
    {
        std::copy(other.data,other.data+1000000,data);
    }
    X(X&& other):                         ❷
        data(other.data)
    {
        other.data=nullptr;
    }
};
```

Копирующий конструктор ❶ определяется вполне ожидаемо: распределение памяти под новый блок и копирование в него данных. Но есть также и новый кон-

структор, получающий прежнее значение по ссылке на *r*-значение ❷. Это *перемещающий конструктор*. В этом случае копируется *указатель* на данные, а в экземпляре объекта `other` остается нулевой указатель, благодаря чему экономится крупный блок памяти и время на создание переменных из *r*-значений.

Для класса `X` перемещающий конструктор является оптимизацией, но в некоторых случаях есть смысл предоставить такой конструктор, даже когда совершенно бессмысленно предоставлять копирующий конструктор. Например, вся суть `std::unique_ptr<>` состоит в том, что каждый ненулевой экземпляр является единственным указателем на свой объект, поэтому в копирующем конструкторе нет никакого смысла. А вот перемещающий конструктор позволяет передавать владение указателем между экземплярами и использовать `std::unique_ptr<>` в качестве возвращаемого значения функции — указатель *не копируется*, а *перемещается*.

Если требуется явное перемещение из поименованного объекта, о котором известно, что больше он не будет использоваться, его можно привести к типу *r*-значения либо воспользовавшись `static_cast<X&&>`, либо вызвав `std::move()`:

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

Это может пригодиться, когда нужно переместить значение параметра в локальную или компонентную переменную без копирования, поскольку, несмотря на то что параметр, представляющий собой ссылку на *r*-значение, может быть привязан к *r*-значениям, внутри функции он рассматривается как *l*-значение:

```
void do_stuff(X&& x_)
{
    X a(x_);                    ← Копируется
    X b(std::move(x_));         ← Перемещается
}
do_stuff(X());                  ← Все верно; r-значение привязывается
                                  к ссылке на r-значение
X x;
do_stuff(x);                    ← Ошибка; l-значение не может
                                  привязываться к ссылке на r-значение
```

Семантика перемещения широко используется в Thread Library, как в тех случаях, когда копирование не имеет смысла. Но ресурсы можно передавать и в качестве оптимизации для исключения весьма затратного копирования, когда источник все равно подлежит уничтожению. Пример я уже приводил в разделе 2.2, где функция `std::move()` использовалась для передачи экземпляра `std::unique_ptr<>` в только что созданный поток, а также в разделе 2.3, где показана передача владения потоками между экземплярами `std::thread`.

Экземпляры классов `std::thread`, `std::unique_lock<>`, `std::future<>`, `std::promise<>` и `std::packaged_task<>` нельзя скопировать, но во всех их классах есть перемещающие конструкторы, позволяющие передавать связанный с ними ресурс между экземплярами и поддерживающие их использование в качестве значений, возвращаемых функцией. Экземпляры классов `std::string` и `std::vector<>` всегда можно скопировать, но у них также есть перемещающие конструкторы и связанные

с перемещениями операторы, позволяющие избегать копирования больших объемов данных из *r*-значения.

Стандартная библиотека C++ никогда ничего не делает с объектом, явно перемещенным в другой объект, кроме его уничтожения или присваивания ему значения (либо путем копирования, либо, что более вероятно, путем перемещения). Но правилом хорошего тона будет гарантировать в инварианте класса наличие состояния «перемещен из». Например, экземпляр `std::thread`, использованный в качестве источника для перемещения, эквивалентен созданному по умолчанию экземпляру `std::thread`, а экземпляр `std::string`, использованный в качестве источника для перемещения, по-прежнему будет иметь допустимое состояние, хотя не дается никаких гарантий насчет того, каким именно будет это состояние (в понятиях длины строки или содержащихся в ней символов).

A.1.2. Ссылки на r-значения и шаблоны функций

Существует еще один, последний, нюанс, имеющий отношение к использованию ссылок на *r*-значения в качестве параметров шаблона функции: если параметр функции является ссылкой на *r*-значение, представляющее собой параметр шаблона, то механизм автоматического выведения типа аргумента шаблона делает вывод, что типом будет ссылка на *l*-значение, если предоставлено *l*-значение, или простой обычный тип, если предоставлено *r*-значение. Последнюю фразу непросто понять, поэтому обратимся к примеру. Рассмотрим следующую функцию:

```
template<typename T>
void foo(T&& t)
{}
```

Если ее вызвать, как показано далее, с *r*-значением, то будет сделан вывод, что `T` является типом значения:

```
foo(42);          // Вызывает foo<int>(42)
foo(3.14159);     // Вызывает foo<double>(3.14159)
foo(std::string()); // Вызывает foo<std::string>(std::string())
```

Но если вызвать `foo` с *l*-значением, то будет сделан вывод, что `T` является ссылкой на *l*-значение:

```
int i=42;
foo(i);   // Вызывает foo<int&>(i)
```

Поскольку параметр функции объявлен как `T&&`, получается, что это ссылка на ссылку, что рассматривается как исходный ссылочный тип. Сигнатура `foo<int&>()` такова:

```
void foo<int&>(int& t);
```

Это позволяет одному шаблону функции принимать в качестве параметров как *l*-, так и *r*-значение, что используется конструктором `std::thread` (см. разделы 2.1 и 2.2), и тогда, если параметр является *r*-значением, предоставляемый вызываемый объект можно не скопировать, а переместить.

А.2. Удаленные функции

Иногда разрешение копировать класс не имеет никакого смысла. Ярким примером этого может послужить `std::mutex`. Что бы получилось, если скопировать мьютекс? Еще один пример: `std::unique_lock<>` — экземпляр является единственным владельцем удерживаемой им блокировки. Если по-настоящему его скопировать, это будет означать, что копия также удерживает блокировку, что не имеет никакого смысла. Передача владения, описанная в подразделе А.1.2, имеет смысл, но это не копирование. Уверен, что вы можете привести и другие примеры.

Стандартной манерой предотвращения копирования класса является объявление копирующего конструктора и оператора копирующего присваивания закрытыми с последующим непредоставлением их реализации. Тогда, если код вне рассматриваемого класса попытается скопировать экземпляр, будет выдана ошибка компиляции, а если это же попытается сделать какая-нибудь компонентная функция класса, будет выдана ошибка на этапе компоновки (из-за отсутствия реализации):

```
class no_copies
{
public:
    no_copies(){}
private:
    no_copies(no_copies const&);                    | Нет реализации
    no_copies& operator=(no_copies const&);         |
};
no_copies a;
no_copies b(a);     <-- Не пройдет компиляцию
```

В комитете по разработке стандарта C++11 понимали, что это широко распространенный прием, и осознавали его невысокую надежность. Поэтому был предоставлен более общий механизм, применимый также и в других классах: функцию можно указать удаленной, добавив к ее объявлению признак `= delete`. Это позволяет переписать `no_copies` следующим образом:

```
class no_copies
{
public:
    no_copies(){}
    no_copies(no_copies const&) = delete;
    no_copies& operator=(no_copies const&) = delete;
};
```

Это намного нагляднее по сравнению с исходным кодом и четче выражает намерение, а также позволяет компилятору выдавать более внятные описания ошибок и перенести момент выдачи ошибки из этапа компоновки в этап компиляции в том случае, если вам вздумается выполнить копирование внутри компонентной функции вашего класса.

Если в дополнение к удалению копирующего конструктора и оператора копирующего присваивания будет также явно написан перемещающий конструктор и оператор перемещающего присваивания, ваш класс будет допускать только перемещение по аналогии с классами `std::thread` и `std::unique_lock<>`. Пример такого типа, предназначенного только для перемещений, показан в листинге А.2.

Листинг А.2. Простой тип, предназначенный только для перемещения

```
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;
move_only m2(m1);
move_only m3(std::move(m1));
```

Ошибка; копирующий конструктор объявлен удаленным

Все в порядке; перемещающий конструктор найден

Объекты, допускающие только перемещения, можно передавать в качестве параметров функциям и возвращать из функций. Но если потребуется переместить их из *l*-значения, это намерение нужно явно выразить, воспользовавшись функцией `std::move()` или оператором `static_cast<T&&>`.

Спецификатор `= delete` можно применять к любой функции, не только к копирующим конструкторам и операторам присваивания. Тем самым поясняется, что функция недоступна. Но это еще не все. Удаленная функция, как и любая другая, участвует в разрешении переопределения, и при ее выборе возникает ошибка компиляции. Этим можно воспользоваться для удаления конкретных переопределений. Например, если ваша функция принимает параметр типа `short`, то сужение значений типа `int` можно предотвратить, написав переопределение, принимающее `int`, и объявив его удаленным:

```
void foo(short);
void foo(int) = delete;
```

Теперь любая попытка вызова `foo` с типом `int` будет встречена выдачей ошибки компиляции, и вызывающему коду придется выполнить явное преобразование предоставленного значения в `short`:

```
foo(42);
foo((short)42);
```

Ошибка; переопределение для типа int объявлено удаленным

Все в порядке

A.3. Функции по умолчанию

Удаленные функции позволяют явно указать на то, что функция не реализована, а у *функций по умолчанию* (defaulted) обратное предназначение: указать на то, что компилятор должен написать функцию вместо вас, применив реализацию по умол-

чанию (исходную). Сделать это можно только в отношении тех функций, которые компилятор может автоматически создавать в любом случае: конструкторов по умолчанию, деструкторов, копирующих конструкторов, перемещающих конструкторов, операторов копирующего присваивания и операторов перемещающего присваивания.

Что может побудить к таким действиям? На это есть несколько причин:

- *желание изменить доступность функции* — изначально функции, генерируемые компилятором, являются открытыми. Если нужно сделать их защищенными или даже закрытыми, создавать их придется самостоятельно. Если объявить их функциями по умолчанию, можно заставить компилятор написать функцию и изменить уровень ее доступности;
- *желание воспользоваться функцией в качестве документации* — если сгенерированной компилятором версии вполне достаточно, то, может быть, стоит объявить о ее достаточности явным образом, чтобы вы или кто-то другой, изучая код, понимали, что это сделано намеренно;
- *желание заставить компилятор сгенерировать функцию, если он не стал бы это делать*, — обычно это делается в отношении конструкторов по умолчанию, которые при иных обстоятельствах создаются компилятором только при отсутствии конструкторов, определенных пользователем. Если нужно, к примеру, определить специализированный копирующий конструктор, можно получить конструктор, сгенерированный компилятором, объявив его исходным (по умолчанию);
- *желание сделать деструктор виртуальным и при этом оставить его в числе генерируемых компилятором*;
- *желание принудительно создать конкретное объявление копирующего конструктора, например, чтобы он принимал параметр по умолчанию не по const-ссылке, а по не-const-ссылке*;
- *желание воспользоваться особыми свойствами сгенерированных компилятором функций, утрачиваемых при предоставлении своей реализации*. Подробное разъяснение вскоре последует.

Если удаленные функции объявляются продолжением объявления спецификатором `= delete`, то функции по умолчанию объявляются продолжением объявления спецификатором `= default`, например:

```
class Y
{
private:
    Y() = default;
public:
    Y(Y&) = default;
    T& operator=(const Y&) = default;
protected:
    virtual ~Y() = default;
};
```

Изменение уровня доступности

Прием не-const-ссылки

Объявление в качестве функции по умолчанию с целью документирования

Изменение уровня доступности и добавление виртуальности

Я уже упоминал, что функции, сгенерированные компилятором, могут иметь особые свойства, которые невозможно было бы получить от версии, определенной пользователем. Наиболее существенное отличие состоит в том, что функция, сгенерированная компилятором, может быть *тривиальной*. Это влечет за собой ряд последствий, включая следующие:

- объекты с тривиальными копирующими конструкторами, тривиальными операторами копирующего присваивания и тривиальными деструкторами можно копировать с помощью `memcpy` или `memmove`;
- у литеральных типов, используемых для `constexpr`-функций (см. раздел А.4), должен быть тривиальный конструктор, копирующий конструктор и деструктор;
- классы с тривиальным конструктором по умолчанию, копирующим конструктором, оператором копирующего присваивания и деструктором можно использовать в связке с конструктором и деструктором, которые определены пользователем;
- классы с тривиальным оператором копирующего присваивания можно использовать с шаблоном класса `std::atomic<>` (см. подраздел 5.2.6) с целью предоставления значения этого типа с атомарными операциями.

Но просто объявление функции с использованием спецификатора `= default` не делает ее тривиальной — она станет таковой только в том случае, если класс также поддерживает тривиальность всех остальных критериев для соответствующей функции — но явное написание функции в пользовательском коде *не даст ей возможности* стать тривиальной.

Второе отличие классов с функциями, сгенерированными компилятором, от их предоставленных пользователем эквивалентов заключается в том, что класс без конструкторов, предоставленных пользователем, может быть *агрегатным*, а значит, его можно инициализировать 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` инициализируется значением `3.141`.

Третье отличие функций, сгенерированных компилятором, от их предоставленных пользователем эквивалентов известно немногим и применимо только к конструктору по умолчанию и к конструктору классов по умолчанию, отвечающих конкретному критерию. Рассмотрим следующий класс:

```
struct X
{
    int a;
};
```

Если создается экземпляр класса `X` без инициализатора, содержащееся в нем значение (`a`), относящееся к типу `int`, *инициализируется по умолчанию*. Если у объекта статическая продолжительность хранения, он инициализируется нулевым значением. В противном случае у него получается неопределенное значение, которое потенциально может вызвать неопределенное поведение, если к нему будет обращение, предшествующее присваиванию ему нового значения:

```
X x1;    <-- У x1.a имеется неопределенное значение
```

Если же инициализировать ваш экземпляр `X` путем явного вызова конструктора по умолчанию, то он будет инициализирован нулевым значением:

```
X x2=X();    <-- x2.a==0
```

Эта весьма странная возможность также распространяется на базовые классы и компоненты. Если у вашего класса есть конструктор по умолчанию, сгенерированный компилятором, и любые ваши компонентные данные и базовые классы также имеют конструкторы по умолчанию, сгенерированные компилятором, то компонентные данные таких базовых классов и компонентов, являющихся встроенными типами, также либо остаются с неопределенным значением, либо инициализируются нулевым значением, в зависимости от того, есть ли у внешнего класса свой явно вызываемый конструктор по умолчанию.

При всей замысловатости и потенциальной предрасположенности к допущению ошибок это правило находит свое применение. Если вы самостоятельно создали конструктор по умолчанию, эта возможность пропадает; либо компонентные данные, подобные `a`, всегда инициализируются (поскольку вы указали значение `a` или явно вызвали конструктор по умолчанию) или вообще не инициализируются (поскольку вы этого не сделали):

```
X::X():a(){}      <-- Всегда a==0
X::X():a(42){}    <-- Всегда a==42
X::X(){}          <-- ❶
```

Если вообще опустить инициализацию `a` из конструктора `X`, как в третьем примере ❶, то переменная `a` останется для нестатических экземпляров `X` не проинициализированной, а для экземпляров `X` со статической продолжительностью хранения будет проинициализирована нулевым значением.

При обычных обстоятельствах, если любой другой конструктор будет написан вами самостоятельно, компилятор больше не будет генерировать для вас конструктор по умолчанию, поэтому, если конструктор вам все же нужен, его придется создавать самостоятельно. Следовательно, эта возможность будет утрачена. Но путем явного объявления конструктора исходным можно заставить компилятор сгенерировать для вас конструктор по умолчанию, и такая возможность сохранится:

```
X::X() = default;    <-- Применяемые по умолчанию правила инициализации
```

Это актуально для атомарных типов (см. раздел 5.2), у которых есть конструкторы, явно объявленные конструкторами по умолчанию. У них всегда неопределенное значение по умолчанию, если только: а) у них нет статической продолжительности хранения (в силу чего они статически инициализируются нулевым значением); б) вы не выполнили явный вызов конструктора по умолчанию для запроса нулевой инициализации; в) вы не указали значение явным образом. Замечу, что в случае применения атомарных типов, чтобы разрешить статическую инициализацию, конструктор для инициализации со значением объявлен как `constexpr` (см. раздел А.4).

А.4. constexpr-функции

Целочисленные литералы, например `42`, являются *константными выражениями*, как и простые арифметические выражения вроде `23*2-4`. Как частью нового константного выражения можно даже воспользоваться `const`-переменными целочисленного типа, которые сами были инициализированы константными выражениями:

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

Для создания переменных, которыми можно воспользоваться в других константных выражениях, кроме использования константных выражений, есть несколько действий, которые могут выполняться *только* с константными выражениями.

- Задание границ массива:

```
int bounds=99;
int array[bounds];
const int bounds2=99;
int array2[bounds2];
```

Это ошибка, bounds не является константным выражением

Все верно, bounds2 является константным выражением

- Задание значения параметра шаблона, не являющегося типом:

```
template<unsigned size>
struct test
{};
test<bounds> ia;
test<bounds2> ia2;
```

Это ошибка, bounds не является константным выражением

Все верно, bounds2 является константным выражением

- Предоставление инициализатора для компонентных данных класса `static const` целочисленного типа в определении класса:

```
class X
{
    static const int the_answer=forty_two;
};
```

- Предоставление инициализатора для встроенного типа или агрегата, применяемого для статической инициализации:

```
struct my_aggregate
{
    int a;
    int b;
};
static my_aggregate ma1={forty_two,123};
int dummy=257;
static my_aggregate ma2={dummy,dummy};
```

Статическая инициализация

Динамическая инициализация

- Подобная статическая инициализация может применяться во избежание проблем, связанных с порядком проведения инициализации и состояниями гонки.

Здесь нет ничего нового — все это можно делать со стандартом C++ 1998 года издания. Но в той части стандарта C++11, где определяются *константные выражения*, появилось расширение, в котором представлено ключевое слово `constexpr`. В стандартах C++14 и C++17 возможности `constexpr` стали еще шире; но полное описание выходит за рамки данного приложения.

В первую очередь, ключевое слово `constexpr` является модификатором функции. Если параметр и возвращаемое значение функции отвечают определенным требованиям, а тело функции имеет достаточно простую конструкцию, такую функцию можно объявить с модификатором `constexpr`. В таком случае ее допускается применять в константных выражениях, например:

```
constexpr int square(int x)
{
    return x*x;
}
int array[square(5)];
```

В данном случае в массиве `array` будет 25 элементов, поскольку функция `square` объявлена с модификатором `constexpr`. Но лишь то, что функция *может* применяться в константном выражении, еще не означает, что все в ней используемое автоматически является константными выражениями:

```
int dummy=4;
int array[square(dummy)];
```

❶ Ошибка, dummy не является константным выражением

В данном примере `dummy` не является константным выражением ❶, следовательно, `square(dummy)` также им не является — это обычный вызов функции, и поэтому он не может использоваться для задания границ массива `array`.

A.4.1. constexpr и типы, определенные пользователем

До сих пор во всех примерах применялись встроенные типы, например `int`. Но новый стандарт C++ позволяет константным выражениям принадлежать к любому типу, удовлетворяющему требованиям, предъявляемым к литеральному типу. Чтобы тип класса мог считаться *литеральным*, необходимо соблюдение следующих условий:

- в классе должен быть тривиальный копирующий конструктор;
- в нем должен быть тривиальный деструктор;

- все нестатические компонентные данные и базовые классы должны относиться к тривиальным типам;
- в классе должен быть либо стандартный конструктор по умолчанию, либо конструктор с модификатором `constexpr`, отличный от копирующего конструктора.

Конструкторы с модификатором `constexpr` будут рассмотрены чуть позже. А пока мы сконцентрируемся на классах с тривиальным конструктором по умолчанию, например на классе `CX` из листинга А.3.

Листинг А.3. Класс с тривиальным конструктором по умолчанию

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;                    ❶
    CX(int a_, int b_):                ❷
        a(a_),b(b_)
    {}
    int get_a() const
    {
        return a;
    }
    int get_b() const
    {
        return b;
    }
    int foo() const
    {
        return a+b;
    }
};
```

Заметьте, что здесь конструктор по умолчанию явно объявляется *исходным* ❶ с применением спецификатора `= default` (см. раздел А.3), чтобы сохранить его тривиальность, несмотря на наличие конструктора, определенного пользователем ❷. Поэтому данный тип удовлетворяет всем требованиям, позволяющим квалифицировать его в качестве литерального типа, и его можно применить в константных выражениях. Можно, к примеру, предоставить `constexpr`-функцию, создающую новые экземпляры:

```
constexpr CX create_cx()
{
    return CX();
}
```

Можно также создать простую `constexpr`-функцию, копирующую свой параметр:

```
constexpr CX clone(CX val)
{
    return val;
}
```

Но это почти все, что позволено сделать в стандарте C++11, — constexpr-функция может только вызвать другие constexpr-функции. В стандарте C++14 это ограничение снято, и в constexpr-функции можно делать почти все что угодно, при условии, что она не изменяет никакие объекты с нелокальной областью видимости. Но даже в рамках стандарта C++11 можно применить constexpr к компонентным функциям и конструктору 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   ←❶
    {
        return a;
    }
    constexpr int get_b()   ←❷
    {
        return b;
    }
    constexpr int foo()
    {
        return a+b;
    }
};
```

Теперь в C++11 квалификатор const в функции get_a() ❶ избыточен, поскольку его применение и так подразумевается присутствием ключевого слова constexpr, а функция get_b() и так квалифицируется как const, даже притом, что квалификатор const опущен ❷. В стандарте C++14 ситуация изменилась (из-за расширенных возможностей constexpr-функций), поэтому get_b() больше не является подразумеваемо помечаемой как const. И теперь можно создавать более сложные constexpr-функции наподобие следующей:

```
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
```

Как бы ни было все это интересно, затраченные усилия не соразмерны с замысловатым способом вычисления каких-то границ массива или составной константы. Основным преимуществом константных выражений и `constexpr`-функций, задействующих типы, определяемые пользователем, является то, что литеральный тип, проинициализированный константным выражением, является статически проинициализированным, следовательно, его инициализация свободна от возникновения состояний гонки и проблем с порядком проведения инициализации:

```
CX si=half_double(CX(42,19));      ◄── Статически проинициализирован
```

Это относится также и к конструкторам. Если конструктор объявлен с ключевым словом `constexpr` и параметры конструктора являются константными выражениями, *инициализация* является *константной* и происходит как составная часть фазы статической инициализации. Это одно из наиважнейших изменений, внесенных в стандарт C++11, с точки зрения развития конкурентности: за счет разрешения конструкторов, определяемых пользователем, которые тем не менее могут проходить статическую инициализацию, можно избежать возникновения любых состояний гонки на этапе их инициализации, поскольку они гарантированно будут проинициализированы еще до запуска любого другого кода.

Это особенно существенно для таких классов, как `std::mutex` (см. подраздел 3.2.1) или `std::atomic<>` (см. подраздел 5.2.6), где может потребоваться использование глобальных экземпляров для синхронизации доступа к другим переменным и при этом не допустить состояний гонки. Это было бы невозможно, если бы конструктор мьютекса стал предметом состояния гонки, поэтому конструктор по умолчанию `std::mutex` объявлен с ключевым словом `constexpr`, чтобы гарантировать неизменную инициализацию мьютекса на этапе статической инициализации.

A.4.2. constexpr-объекты

До сих пор мы говорили только о `constexpr` по отношению к функциям. Но ключевое слово `constexpr` можно также применять и к объектам. Главным образом это делается в целях диагностики. Тем самым проверяется, что объект проинициализирован константным выражением, `constexpr`-конструктором или агрегатным инициализатором, составленным из константных выражений. Кроме того, объект при этом объявляется как `const`:

```
                                  ┌─ Все правильно
constexpr int i=45;               ◄──┘
constexpr std::string s("hello");    ◄── Ошибка; std::string не является
                                         литеральным типом
int foo();
constexpr int j=foo();            ◄── Ошибка; foo() не объявлена
                                      с ключевым словом constexpr
```

A.4.3. Требования к constexpr-функциям

Чтобы объявить функцию с модификатором `constexpr`, она должна отвечать определенным требованиям. Если она им не отвечает, объявление ее с этим модификато-

ром вызовет ошибку компиляции. В C++11 к `constexpr`-функциям предъявляются следующие требования:

- все параметры должны быть литерального типа;
- возвращаемое значение должно быть литерального типа;
- тело функции должно состоять из одной инструкции `return`;
- выражение в инструкции `return` должно быть константным;
- любой конструктор или оператор преобразования, используемый для построения из выражения возвращаемого значения, должен быть объявлен с модификатором `constexpr`.

Здесь нет ничего необычного. Должна сохраняться возможность встраивания функции в константное выражение, чтобы оно таковым и оставалось, и ничего изменять нельзя. `Constexpr`-функции относятся к *чистым функциям* и не вызывают побочных эффектов.

В C++14 требования были существенно снижены. Хотя сама идея о чистой функции без побочных эффектов осталась, телу функции разрешено значительно расширить содержимое:

- разрешено наличие нескольких инструкций `return`;
- объекты, созданные внутри функции, можно изменять;
- допускается применение циклов, условных выражений и коммутирующих инструкций.

К `constexpr`-функциям, являющимся компонентами класса, предъявляются дополнительные требования:

- `constexpr`-функции, являющиеся компонентами класса, не могут быть виртуальными;
- класс, для которого функция является компонентной, должен быть литерального типа.

К `constexpr`-конструкторам применяются другие правила:

- для компилятора C++11 тело конструктора должно быть пустым; для компилятора C++14 и компиляторов более поздних версий конструктор должен отвечать требованиям, предъявляемым к `constexpr`-функциям;
- каждый базовый класс должен быть проинициализирован;
- каждые не-`static` компонентные данные должны быть проинициализированы;
- любые выражения, используемые в списке инициализации компонента, должны квалифицироваться как константные выражения;
- конструкторы, выбранные для инициализации компонентных данных и базовых классов, должны быть `constexpr`-конструкторами;
- любой конструктор или оператор преобразования, используемый для построения компонентных данных и базовых классов из соответствующего выражения инициализации, должен быть объявлен с модификатором `constexpr`.

Это те же правила, что и для функций, за исключением отсутствия возвращаемого значения, поэтому инструкция `return` отсутствует. Вместо этого конструктор инициализирует все базовые компоненты и компоненты данных, имеющиеся в списке инициализации компонентов. Тривиальные копирующие конструкторы подразумеваемо являются `constexpr`-конструкторами.

A.4.4. constexpr и шаблоны

Когда модификатор `constexpr` применяется к шаблону функции или к компонентной функции шаблона класса, он игнорируется, если параметры или возвращаемые типы конкретного экземпляра шаблона не являются литеральными типами. Это позволяет создавать шаблоны функций, которые становятся `constexpr`-функциями при соответствующем типе параметров шаблона и обычными встраиваемыми функциями в ином случае, например:

```
template<typename T>
constexpr T sum(T a,T b)
{
    return a+b;
}
constexpr int i=sum(3,42);
std::string s=
    sum(std::string("hello"),
        std::string(" world"));
```

Все верно; sum<int> относится к constexpr

Все верно, но sum<std::string> не относится к constexpr

Функция должна удовлетворять всем другим требованиям, предъявляемым к `constexpr`-функциям. Объявлять `constexpr`-функцию с несколькими инструкциями только потому, что она является шаблоном функции, нельзя. Это неизменно вызовет ошибку компиляции.

A.5. Лямбда-функции

Благодаря своим возможностям существенно упростить код и избавить его от шаблонности, связанной с созданием вызываемых объектов, лямбда-функции считаются одними из наиболее интересных функций стандарта C++11. Синтаксис лямбда-функций C++11 позволяет определять функцию там, где она необходима в другом выражении. Он хорошо вписывается в предикаты, предоставляемые функциям ожидания `std::condition_variable` (как в примере из подраздела 4.1.1), поскольку позволяет быстро выражать семантику в понятиях доступных переменных, вместо того чтобы перехватывать необходимое состояние в компонентных переменных класса с помощью оператора вызова функции.

В простейшем случае *лямбда-выражение* определяет автономную функцию, не получающую параметров и полагающуюся только на глобальные переменные и функции. Ей даже не нужно возвращать значение. Такое лямбда-выражение представляет собой ряд инструкций, заключенных в фигурные скобки, перед которыми стоят квадратные скобки *(лямбда-интродуктор)*:

```
[]{                    ←— Лямбда-выражение начинается с []
    do_stuff();
    do_more_stuff();
}();                   ←— Завершение лямбда-выражения и его вызов
```

В этом примере лямбда-выражение тут же вызывается за счет того, что сразу за ним следуют круглые скобки, но обычно так не делается. Ведь для непосредственного вызова можно обойтись и без лямбда-выражения и внести инструкции непосредственно в исходный код. Чаще всего лямбда-выражение передается в качестве параметра шаблону функции, получающему в качестве одного из своих параметров вызываемый объект. В таком случае лямбда-выражению, вероятно, нужно будет получить параметры, или иметь возвращаемое значение, или использовать и то и другое. Если нужно получить параметры, их список следует указать сразу же за лямбда-интродуктором, как это делается с обычной функцией. Например, следующий код записывает все элементы вектора `std::cout`, используя в качестве разделителей символы новой строки:

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

Практически так же легко решить вопрос с возвращаемым значением. Если тело лямбда-функции состоит из одной инструкции `return`, возвращаемым типом лямбда-функции является тип возвращаемого выражения. Например, простой лямбда-функцией, показанной в листинге А.4, можно воспользоваться для ожидания установки флага с помощью `std::condition_variable` (см. подраздел 4.1.1).

Листинг А.4. Простая лямбда-функция с автоматически выводимым возвращаемым типом

```
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;});   ←—❶
}
```

Возвращаемый тип лямбда-функции, переданный `cond.wait()` ❶, выводится из типа `data_ready` и в результате является булевым. Когда условная переменная пробуждается от ожидания, она вызывает лямбда-функцию с заблокированным мьютексом и возвращается только после вызова `wait()`, когда `data_ready` имеет значение `true`.

А если создать тело лямбда-функции из одной инструкции `return` не представляется возможным? В таком случае придется указать тип возвращаемого значения явным образом. Это можно сделать, даже если тело функции состоит из одной инструкции `return`, но вы *должны* сделать это, если тело лямбда-функции представляет собой более сложную конструкцию. Тип возвращаемого значения указывается после списка параметров лямбда-функции за стрелкой (`->`). Если лямбда-функция не получает никаких параметров, для явного указания типа возвращаемого значения

нужно все равно выключить пустой список параметров. Наш предикат условной переменной можно записать следующим образом:

```
cond.wait(lk,[]()->bool{return data_ready;});
```

Указав возвращаемый тип, можно расширить область применения лямбда-функции до работы с регистрационными сообщениями или до выполнения более сложной обработки данных:

```
cond.wait(lk,[]()->bool{
    if(data_ready)
    {
        std::cout<<"Data ready"<<std::endl;
        return true;
    }
    else
    {
        std::cout<<"Data not ready, resuming wait"<<std::endl;
        return false;
    }
});
```

Показанные здесь лямбда-функции весьма эффективны и могут существенно упростить код, но наибольшая эффективность таких функций проявляется при захвате локальных переменных.

A.5.1. Лямбда-функции, ссылающиеся на локальные переменные

Лямбда-функции с *интродуктором* `[]` не могут ссылаться на локальные переменные из области видимости. Им разрешено использовать только глобальные переменные и все, что передано в качестве параметра. Если нужен доступ к локальной переменной, ее следует захватить. Проще всего это сделать путем *захвата* всего набора переменных внутри локальной области видимости, используя лямбда-интродуктор, имеющий вид `[=]`. Вот, собственно, и все, что нужно сделать, — теперь ваша лямбда-функция может обращаться к *копиям* локальных переменных, сделанных на момент создания лямбда-функции.

Чтобы посмотреть на это в действии, рассмотрим следующий пример простой функции:

```
std::function<int(int)> make_offseter(int offset)
{
    return [=](int j){return offset+j;};
}
```

Каждый вызов `make_offseter` возвращает посредством функции-оболочки `std::function<>` новый объект лямбда-функции. Эта возвращаемая функция добавляет предоставленное смещение к любому переданному ей параметру. Например, при выполнении кода:

```
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;
}
```

Числа `54`, `135` будут выведены дважды, поскольку функция, возвращаемая из первого вызова `make_offseter`, всегда прибавляет `42` к предоставленному аргументу, а функция, возвращаемая из второго вызова `make_offseter`, всегда прибавляет к предоставленному аргументу `123`.

Это наиболее безопасная форма захвата локальной переменной; здесь все копируется, что позволяет возвратить лямбда-функцию и вызвать ее за пределами области видимости функции по умолчанию. Но это не единственный вариант захвата; вместо этого можно все захватить по ссылке. В таком случае, если переменные, на которые ссылается лямбда-функция, были уничтожены из-за выхода из области видимости функции или блока, которым они принадлежали, вызов лямбда-функции может привести к неопределенному поведению, точно так же, как приводит к неопределенному поведению ссылка на уже уничтоженную переменную при любых других обстоятельствах.

Как показано в следующем примере, лямбда-функция, захватывающая все локальные переменные по ссылке, обозначается при создании с помощью интродуктора `[&]`:

```
int main()
{
    int offset=42;                                                    ❶
    std::function<int(int)> offset_a=[&](int j){return offset+j;};   ❷
❸   offset=123;
    std::function<int(int)> offset_b=[&](int j){return offset+j;};   ❹
    std::cout<<offset_a(12)<<","<<offset_b(12)<<std::endl;            ❺
❻   offset=99;
    std::cout<<offset_a(12)<<","<<offset_b(12)<<std::endl;            ❼
}
```

В функции `make_offseter` из предыдущего примера для захвата копии смещения использовался лямбда-интродуктор `[=]`, а в функции `offset_a` в этом примере для захвата смещения по ссылке используется лямбда-интродуктор `[&]` ❷. И неважно, что значением смещения по умолчанию является 42 ❶; результат вызова `offset_a(12)` всегда будет зависеть от текущего значения смещения `offset`. Перед созданием второй (идентичной) лямбда-функции `offset_b` ❹ значение `offset` изменяется на `123` ❸, и эта вторая лямбда-функция опять захватывает смещение по ссылке, следовательно, результат зависит от текущего значения `offset`.

Теперь в первой строке вывода ❺ `offset` все еще имеет значение `123`, поэтому вывод имеет вид `135,135`. Но во второй строке вывода ❼ `offset` был изменен на `99` ❽, поэтому теперь вывод имеет вид `111,111`. Обе функции, `offset_a` и `offset_b`, прибавляют текущее значение `offset` (`99`) к предоставленному аргументу (`12`).

Но благодаря особенностям C++ выбирать между всем или ничем не приходится. Можно выбрать вариант, при котором некоторые переменные будут захватываться по копии, а некоторые — по ссылке, и можно путем настройки лямбда-интродуктора захватывать только явно выбранные переменные. Если нужно *скопировать* все используемые переменные, за исключением одной или двух, можно воспользоваться лямбда-интродуктором вида `[=]`, но за знаком равенства поставить список переменных, захватываемых по ссылке, перед которыми поставить знаки амперсанда. Следующий пример выведет число `1239`, потому что `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 сделать и по-другому: захватить по ссылке переменные по умолчанию, а указанный поднабор переменных захватить копированием. В таком случае используется лямбда-интродуктор вида `[&]`, но за амперсандом следует список переменных, захватываемых копированием. Следующий пример выведет число `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 нужно захватить только поименованные переменные, то можно указать лишь список захватываемых переменных, а ведущий символ `=` или `&` опустить. При этом, если нужна не копия, а захват по ссылке, перед именами соответствующих переменных следует поставить знак амперсанда. Следующий код выведет число `5682`, потому что переменные `i` и `k` захвачены по ссылке, а переменная `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 вариант позволяет обеспечить захват только нужных переменных, поскольку любая ссылка на локальную переменную, отсутствующую в списке захвата, повлечет за собой ошибку компиляции. Выбор этого варианта заставляет проявлять осторожность при доступе к компонентам класса, если функция, содержащая лямбда-функцию, является компонентной. Компоненты класса не могут захватываться напрямую. Если доступ к ним нужен из вашей лямбда-функции, придется захватывать указатель `this`, добавляя его в список захвата. В следующем примере лямбда-функция захватывает `this`, открывая тем самым доступ к компоненту 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 конкурентности наибольшая польза от применения лямбда-функций проявляется в их использовании в качестве предиката для `std::condition_variable::wait()` (см. подраздел 4.1.1) и с `std::packaged_task<>` (см. подраздел 4.2.1) или с пулами потоков для упаковки небольших задач. Их также можно передать в качестве функции потока конструктору `std::thread` (см. подраздел 2.1.1) и в качестве просто функции при использовании таких параллельных алгоритмов, как `parallel_for_each()` (из подраздела 8.5.1).

С появлением стандарта C++14 лямбда-функции могут также быть *обобщенными*, где типы параметров объявляются как `auto`, а не как указанный тип. В таком случае оператор вызова функции подразумеваемо является шаблоном и тип параметра при вызове лямбда-функции выводится из представленного аргумента. Например:

```
auto f=[](auto x){ std::cout<<"x="<<x<<std::endl;};
f(42);             // x относится к типу int; выводится "x=42"
f("hello");        // x относится к типу const char*; выводится "x=hello"
```

В стандарт C++14 также добавлено понятие *обобщенных захватов*, позволяющее захватывать результаты выражений, а не непосредственную копию локальной переменной или ссылку на нее. Чаще всего это может пригодиться при захвате типов, предназначенных только для перемещения, путем их перемещения, а не при обязательном захвате их по ссылке. Например:

```
std::future<int> spawn_async_task(){
    std::promise<int> p;
    auto f=p.get_future();
    std::thread t([p=std::move(p)](){ p.set_value(find_the_answer());});
    t.detach();
    return f;
}
```

Здесь промис перемещен в лямбда-функцию путем обобщенного захвата `p=std::move(p)`, следовательно, можно безопасно отключиться от потока, не тревожась о появлении висячей ссылки на уничтоженную локальную переменную. Теперь, после создания лямбда-функции, исходная переменная `p` находится в состоянии «только для перемещения», что вынуждает получить фьючерс заранее.

A.6. Вариативные шаблоны

Вариативными считаются шаблоны с переменным числом параметров. Так же как ранее была возможность располагать вариативными функциями, например `printf`, получающими переменное число параметров, теперь можно располагать и вариативными шаблонами, имеющими переменное число параметров *шаблона*. Вариативные шаблоны повсеместно используются в C++ Thread Library. Например, вариативным функциональным шаблоном является конструктор для запуска потока `std::thread` (см. подраздел 2.1.1), к таким шаблонам также относится и `std::packaged_task<>` (см. подраздел 4.2.2). С пользовательской точки зрения достаточно знать, что шаблон принимает неограниченное число параметров, но, если захочется создать такой шаблон или же появится интерес к тому, как это все работает, без подробностей не обойтись.

Точно так же, как вариативные функции объявляются с помощью многоточия (`...`) в списке параметров функции, вариативные шаблоны объявляются с помощью многоточия в списке параметров шаблона:

```
template<typename ... ParameterPack>
class my_template
{};
```

Вариативные шаблоны можно также использовать для частичной специализации шаблона, даже если первичный шаблон не был вариативным. Например, первичный шаблон для `std::packaged_task<>` (см. подраздел 4.2.1) относится к простым шаблонам с одним параметром:

```
template<typename FunctionType>
class packaged_task;
```

Но этот первичный шаблон нигде не определяется; это всего лишь прототип для частичной специализации:

```
template<typename ReturnType,typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

В этой частичной специализации и содержится реальное определение класса. В главе 4 было показано, что для объявления задачи, получающей при ее вызове в качестве параметров объекты типов `std::string` и `double` и предоставляющей результат посредством объекта `std::future<int>`, можно воспользоваться кодом `std::packaged_task<int(std::string,double)>`.

В данном объявлении демонстрируются еще две особенности вариативных шаблонов. Первая из них относительно проста: наряду с вариативными (`Args`) в том же объявлении можно указывать и обычные параметры (например, `ReturnType`). Вторая особенность продемонстрирована в использовании в качестве списка аргументов специализации шаблона `Args...`, чтобы показать, что типы, составляющие `Args` при создании экземпляра шаблона, будут перечислены в этом месте. Поскольку это частичная специализация, данный код работает как сопоставление с образцом; типы, появляющиеся в реальном контексте в экземпляре, захватываются как `Args`. Вариативный параметр `Args` называется *пакетом параметров*, а использование `Args...` — *расширением пакета*.

Как и в вариативных функциях, вариативная часть может быть пустой или же может иметь множество записей. Например, в `std::packaged_task<my_class()>` параметром `ReturnType` является `my_class`, а пакет параметров `Args` пуст, в `std::packaged_task<void(int,double,my_class&,std::string*)>` параметром `ReturnType` является `void`, а пакет параметров `Args` — списком из `int`, `double`, `my_class&`, `std::string*`.

А.6.1. Расширение пакета параметров

Эффективность вариативных шаблонов проявляется в том, что можно сделать с расширением пакета параметров: здесь нет ограничений по расширению списка типов в его исходный вид. В первую очередь, расширением пакета параметров можно воспользоваться напрямую везде, где требуется список типов, например в списке аргументов для другого шаблона:

```
template<typename ... Params>
struct dummy
{
    std::tuple<Params...> data;
};
```

В данном случае единственный вариативный компонент `data` является экземпляром `std::tuple<>`, содержащим все указанные типы, следовательно, у `dummy<int,double,char>` имеется компонентный тип `std::tuple<int,double,char>`. Расширения пакетов параметров можно использовать с обычными типами:

```
template<typename ... Params>
struct dummy2
{
    std::tuple<std::string,Params...> data;
};
```

На этот раз у класса `tuple` есть дополнительный (первый) компонент типа `std::string`. Здесь привлекает возможность создания схемы с расширением пакета параметров, которая затем копируется для каждого имеющегося в расширении элемента. Для этого многоточие (`...`), обозначающее расширение пакета параметров,

помещается в конец схемы. Например, вместо простого создания кортежа элементов, предоставленных в пакете параметров, можно создать кортеж указателей на элементы или даже кортеж из интеллектуальных указателей `std::unique_ptr<>` на ваши элементы:

```
template<typename ... Params>
struct dummy3
{
    std::tuple<Params* ...> pointers;
    std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

Выражение, в котором дается описание типа, может быть сколь угодно сложным, при условии, что пакет параметров входит в состав этого выражения и что за выражением следует многоточие, обозначающее расширение. При расширении пакета параметров для каждой записи в пакете имеющийся в ней тип подставляется в выражение описания типа для создания соответствующего элемента в получающемся списке. Если ваш пакет параметров `Params` содержит типы `int, int, char`, то в результате расширения, указанного в `std::tuple<std::pair<std::unique_ptr<Params>,double> ... >`, получится `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 в этом шаблоне не должны быть вариативные параметры, но если это условие соблюдено, то размер пакета должен в точности совпадать с числом требуемых параметров шаблона:

```
template<typename ... Types>
struct dummy4
{
    std::pair<Types...> data;
};
dummy4<int,char> a;
dummy4<int> b;
dummy4<int,int,int> c;
```

Все верно, data относится к типу std::pair<int,char>

Ошибка; отсутствует второй тип

Ошибка; слишком много типов

Второй способ применения расширения пакета параметров подойдет для объявления списка параметров функции:

```
template<typename ... Args>
void foo(Args ... args);
```

Здесь создается новый пакет параметров `args`, являющийся списком параметров функции, а не списком типов, и его, как и прежде, можно расширить, указав многоточие. Теперь схемой с расширением пакета можно воспользоваться для объявления параметров функции так же, как и при расширении пакета в любом другом месте. Например, такая схема используется конструктором `std::thread` для получения всех аргументов функции по ссылке на *r*-значение (см. раздел А.1):

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

Затем пакет параметров функции может использоваться для вызова другой функции путем указания расширения пакета в списке аргументов вызываемой функции.

Как и в случае с расширениями типов, схемой можно воспользоваться для каждого выражения в списке получающихся аргументов. Например, одна широко распространенная идиома применения ссылок на *r*-значения заключается в использовании `std::forward<>` для предотвращения присутствия в предоставленных аргументах функции всего, что не является *r*-значениями:

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
    foo(std::forward<ArgTypes>(args)...);
}
```

Следует отметить, что в данном случае расширение пакета содержит как пакет типов `ArgTypes`, так и пакет параметров функции `args`, а многоточие указано после всего выражения. Если вызвать такой вот `bar`:

```
int i;
bar(i,3.141,std::string("hello"));
```

то результат расширения примет следующий 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));
}
```

Здесь первый аргумент вполне корректно передается `foo` в качестве ссылки на *l*-значение, а другие аргументы передаются в качестве ссылок на *r*-значения.

И последнее, что можно сделать с расширением пакета параметров, — это определить его длину с помощью оператора `sizeof...`. Все очень просто: `sizeof...(p)` возвращает число элементов в пакете параметров `p`. И неважно, является ли этот пакет пакетом параметров или пакетом аргументов функции; результат будет одинаковым. И это, наверное, единственный случай, когда можно будет воспользоваться пакетом параметров и не поставить за ним многоточие, поскольку оно уже является частью оператора `sizeof...`. Следующая функция возвращает число предоставленных ей аргументов:

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
    return sizeof... (Args);
}
```

Как и в случае применения обычного оператора `sizeof`, результатом выполнения `sizeof...` является константное выражение, следовательно, им можно воспользоваться для задания границ массива и т. д.

А.7. Автоматическое выведение типа переменной

C++ является статически типизированным языком: на время компиляции тип каждой переменной уже известен. Более того, программист обязан указать тип каждой переменной. В некоторых случаях из-за этого могут получаться весьма громоздкие имена. 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");
```

Обычно в качестве решения, позволяющего сократить длину идентификаторов типов и по возможности снизить количество проблем, возникающих из-за несовместимости типов, использовались псевдонимы типов (`typedef`). Этот прием работает и в C++11, но в данном стандарте появился и новый прием: если переменная инициализируется в своем определении значением того же типа, то в качестве типа можно указать `auto`. В таком случае компилятор автоматически выведет тип переменной, который станет таким же, как и у инициализировавшего ее значения. Пример итератора можно записать в виде:

```
auto iter=m.find("my key");
```

Теперь обычное ключевое слово `auto` можно украсить объявлением `const`-переменных, или указателя, или ссылочных переменных. Рассмотрим несколько примеров объявления переменных, использующих `auto` и соответствующий тип переменной:

```
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 для еще одного места в языке, где выводятся типы: в параметрах шаблонов функций. В объявлении вида:

```
some-type-expression-involving-auto var=some-expression;
```

тип `var` будет таким же, как и тип, выведенный для параметра шаблона функции с тем же выражением типа, за исключением замены `auto` именем параметра типа шаблона:

```
template<typename T>
void f(type-expression var);
f(some-expression);
```

Это означает, что типы массивов сводятся к указателям, а ссылки опускаются, если только в выражении типа переменная не объявлена явным образом как ссылка. Например:

```
int some_array[45];
auto p=some_array;    // int*
int& r=*p;
auto x=r;             // int
auto& y=r;            // int&
```

Это может существенно упростить объявление переменных, в частности там, где полноценный идентификатор типа слишком длинный или, возможно, даже неизвестен (например, тип результата вызова функции в шаблоне).

A.8. Локальные переменные потока

Локальные переменные потока позволяют располагать отдельными экземплярами переменной для каждого потока в вашей программе. Переменная помечается как локальная переменная потока путем ее объявления с ключевым словом `thread_local`. Локальными по отношению к потоку могут объявляться переменные в области видимости пространства имен, статические компонентные данные класса и локальные переменные, и о них говорят, что имеют *потоковое время жизни* (thread storage duration):

```
thread_local int x;
class X
{
    static thread_local std::string s;
};
static thread_local std::string X::s;
void foo()
{
    thread_local std::vector<int> v;
}
```

Локальные переменные потока в области видимости пространства имен

Локальные переменные потока в области видимости пространства имен

Статические компонентные данные класса, являющиеся локальными по отношению к потоку

Требуется определение X::s

Локальная переменная потока

Локальные переменные потока в области видимости пространства имен и локальные по отношению к потоку статические компонентные данные класса создаются перед первым использованием локальной переменной потока из той же единицы компиляции, но *когда именно* — не оговаривается. В некоторых реализациях локальные переменные потока можно создавать при запуске потока; а в некоторых — сразу же перед их первым использованием в каждом потоке, бывает и так, что их создают и в другие определенные моменты времени или же при каких-то обстоятельствах в зависимости от контекста их использования. Конечно же, если из конкретной единицы компиляции никакие локальные переменные потока не используются, нет никаких гарантий, что они вообще будут созданы. Это позволяет производить динамическую загрузку модулей, содержащих локальные переменные потока. Такие переменные можно создавать для конкретного потока при первой же его ссылке на локальную переменную потока из динамически загруженного модуля.

Локальные переменные потока, объявленные внутри функции, инициализируются при первом же проходе потока управления через их объявление в конкретном потоке. Если функция конкретным потоком не объявляется, то все локальные переменные потока, объявляемые в этой функции, не создаются. Это точно такое же поведение, которое проявляется в отношении локальных статических переменных, за исключением того, что оно применяется отдельно к каждому потоку.

У локальных переменных потока есть и другие свойства, присущие статическим переменным: перед любой предстоящей инициализацией (например, динамической) они проходят нулевую инициализацию и если при создании локальной переменной потока выдается исключение, то для прекращения выполнения приложения вызывается функция `std::terminate()`.

Деструкторы для всех локальных переменных потока, созданных в конкретном потоке, запускаются при возвращении из функции потока в порядке, обратном созданию этих переменных. Поскольку порядок инициализации не определяется, то важно обеспечить отсутствие взаимозависимостей между деструкторами таких переменных. Если выход из деструктора локальной переменной потока выполняется с выдачей исключения, то, как и для конструктора, вызывается функция `std::terminate()`.

Локальные переменные потока также уничтожаются для потока, если этот поток вызывает функцию `std::exit()` или возвращается из функции `main()` (что эквивалентно вызову `std::exit()` с возвращаемым значением функции `main()`). Если при выходе из приложения продолжают выполняться другие потоки, деструкторы локальных переменных потока в таких потоках не вызываются.

Хотя у каждого потока адреса для локальных переменных разные, возможность получить обычный указатель на эти переменные все же есть. Затем указатель ссылается на объект в том потоке, который получает адрес, и может использоваться, позволяя другим потокам обращаться к этому объекту. Обращение к объекту после его уничтожения вызывает (как и всегда) неопределенное поведение, поэтому, если указатель на локальную переменную потока передается другому потоку, нужно проследить за тем, чтобы он не разыменовывался после завершения того потока, которому он принадлежит.

A.9. Выведение аргументов шаблона класса

Стандарт C++17 расширяет замысел автоматического выведения типов параметров шаблона: если объявляется объект шаблонного типа, то во многих случаях тип параметров шаблона можно вывести из инициализатора объекта. В частности, если объект объявляется с именем шаблона класса, без указания списка аргументов шаблона, то конструкторы, объявленные в шаблоне класса, используются для выведения аргументов шаблона из инициализатора объекта в соответствии с обычными правилами выведения типов для шаблонов функций.

Например, `std::lock_guard` получает единственный параметр шаблона, который является типом мьютекса. Конструктор также получает единственный параметр, ссылающийся на этот тип. Если объявить, что объект относится к типу `std::lock_guard`, то тип параметра можно вывести из типа предоставленного мьютекса:

```
std::mutex m;
std::lock_guard guard(m); // выводится std::lock_guard<std::mutex>
```

Этот же принцип применим и к шаблону класса `std::scoped_lock`, за исключением того, что у него есть несколько параметров шаблона, которые можно вывести из нескольких мьютексных аргументов:

```
std::mutex m1;
std::shared_mutex m2;
std::scoped_lock guard(m1,m2);
// выводится std::scoped_lock<std::mutex,std::shared_mutex>
```

Для тех шаблонов, у которых конструкторы неправильно выводят типы, автор шаблона может написать явные руководства по выводу, гарантируя правильный вывод типов. Но такие возможности выходят за рамки данной книги.

Резюме

В этом приложении возможности языка, введенные стандартом C++11, рассмотрены поверхностно, поскольку мы разобрали только свойства, активно используемые в Thread Library. К другим новым свойствам языка, наряду с многочисленными незначительными изменениями, можно отнести статические утверждения, строго типизированные перечисления, делегирование конструкторов, поддержку Unicode, псевдонимы шаблонов и новую универсальную последовательность инициализации. Подробное описание всех новых свойств не входило в задачи написания данной книги и заслуживает отдельного издания. Существенное количество изменений было привнесено и стандартами C++14 и C++17, но опять же они выходят за рамки данной книги. На мой взгляд, наилучший обзор всего набора изменений, внесенных в стандарт на момент написания этой книги, дан в документации на сайте http://www.cppreference.com, а также в ответах на наиболее часто задаваемые вопросы (FAQ) по C++11 Бьерна Страуструпа (Bjarne Stroustrup) (http://www.research.att.com/~bs/C++0xFAQ.html), хотя можно также в скором времени рассчитывать на соответствующую актуализацию популярных справочников по C++.

Надеюсь, что краткого изложения новых свойств в этом приложении было вполне достаточно, чтобы показать, какое отношение они имеют к Thread Library, позволить вам создавать многопоточный код и разбираться в чужих кодах, использующих новые свойства. Хотя приложение дает достаточное представление для простого использования рассмотренных свойств, это лишь краткое введение, а не полный справочник или руководство по их использованию. Если есть намерение интенсивно применять эти новые свойства, я рекомендую приобрести соответствующий справочник или руководство.

Б Краткое сравнение библиотек для написания конкурентных программ

Поддержка конкурентности и многопоточности в языках программирования и библиотеках не нова, несмотря на сравнительно недавнюю стандартизацию такой поддержки в C++. Например, в языке Java поддержка многопоточности была уже в его первых выпусках. На платформах, соответствующих стандарту POSIX, предоставляется интерфейс языка C, предназначенный для многопоточности. А в языке Erlang предоставляется поддержка для реализации конкурентности на основе отправки сообщений. Существуют даже библиотеки классов C++, например Boost, в которые заключается соответствующий интерфейс программирования многопоточности, используемый в любой отдельно взятой платформе (будь то интерфейс POSIX C или что-либо иное), предназначенные для того, чтобы предоставить интерфейс, переносимый между поддерживаемыми платформами.

Для тех, у кого уже есть опыт написания многопоточных приложений и желание использовать новые свойства многопоточности C++, в этом приложении предлагаю сравнение свойств, доступных в Java, POSIX C, C++ с Boost Thread Library и C++11. Приведу также перекрестные ссылки на соответствующие главы в этой книге.

Функция	Java	POSIX C	Boost Threads	C++11	Ссылка на главу
Запуск потоков	Класс Java.lang.thread	Тип pthread_t и связанные с ним API-функции: pthread_create(), pthread_detach() и pthread_join()	Класс boost::thread и компонентные функции	Класс std::thread и компонентные функции	Глава 2
Взаимное исключение	Блоки synchronized	Тип pthread_mutex_t и связанные с ним API-функции: pthread_mutex_lock(), pthread_mutex_unlock() и т. д.	Класс boost::mutex и компонентные функции, а также шаблоны boost::lock_guard<> и boost::unique_lock<>	Класс std::mutex и компонентные функции, а также шаблоны std::lock_guard<> и std::unique_lock<>	Глава 3
Отслеживание или ожидание предиката	Методы wait() и notify() класса java.lang.Object, используемые внутри блоков synchronized	Тип pthread_cond_t и связанные с ним API-функции: pthread_cond_wait(), pthread_cond_timed_wait() и т. д.	Классы boost::condition_variable и boost::condition_variable_any и компонентные функции	Классы std::condition_variable и std::condition_variable_any и компонентные функции	Глава 4
Атомарные операции и модель памяти с прицелом на конкурентность	volatile-переменные, типы в пакете java.util.concurrent.atomic	Отсутствует	Отсутствует	Типы std::atomic_xxx, шаблон класса std::atomic<>, функция std::atomic_thread_fence()	Глава 5
Потокобезопасные контейнеры	Контейнеры в пакете java.util.concurrent	Отсутствует	Отсутствует	Отсутствует	Главы 6 и 7
Фьючерсы	Интерфейс java.util.concurrent.future и связанные с ними классы	Отсутствует	Шаблоны классов boost::unique_future<> и boost::shared_future<>	Шаблоны классов std::future<>, std::shared_future<> и std::atomic_future<>	Глава 4
Пулы потоков	Класс java.util.concurrent.ThreadPoolExecutor	Отсутствует	Отсутствует	Отсутствует	Глава 9
Прерывание потока	Метод interrupt() класса java.lang.Thread	pthread_cancel()	Компонентная функция interrupt() класса boost::thread	Отсутствует	Глава 9

Среда передачи сообщений и полный пример программы управления банкоматом

В главе 4 я приводил пример отправки сообщений между потоками с использованием среды передачи сообщений. Там применялась простая реализация кода управления банкоматом. Приведу полный код этого примера, включая среду передачи сообщений.

В листинге B.1 показан код очереди сообщений. Список сообщений хранится в виде указателей на базовый класс. Конкретный тип сообщений обрабатывается классом шаблона, являющимся производным от этого базового класса. Помещение записи в очередь приводит к созданию соответствующего экземпляра класса-оболочки и сохранению указателя на него. Извлечение записи приводит к возвращению этого указателя. Поскольку класс `message_base` не имеет компонентных функций, извлекающему потоку, прежде чем он сможет обратиться к сохраненному сообщению, потребуется приведение указателя к подходящему типу указателя `wrapped_message<T>`.

Листинг B.1. Простая очередь сообщений

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>
namespace messaging
{
    struct message_base
    {
        virtual ~message_base()
        {}
    };
```

Базовый класс записей вашей очереди

```
    template<typename Msg>
    struct wrapped_message:
        message_base
    {
        Msg contents;
        explicit wrapped_message(Msg const& contents_):
            contents(contents_)
        {}
    };
    class queue
    {
        std::mutex m;
        std::condition_variable c;
        std::queue<std::shared_ptr<message_base> > q;
    public:
        template<typename T>
        void push(T const& msg)
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(std::make_shared<wrapped_message<T> >(msg));
            c.notify_all();
        }
        std::shared_ptr<message_base> wait_and_pop()
        {
            std::unique_lock<std::mutex> lk(m);
            c.wait(lk,[&]{return !q.empty();});
            auto res=q.front();
            q.pop();
            return res;
        }
    };
}
```

У каждого типа сообщения своя специализация

Ваша очередь сообщений

Во внутренней очереди сохраняются указатели на message_base

Заключение размещенного сообщения в оболочку и сохранение указателя

Блокировка до минимального заполнения очереди

Отправкой сообщений занимается экземпляр класса `sender`, показанный в листинге B.2. Он представляет собой тонкую оболочку вокруг очереди сообщений, позволяющую лишь помещать сообщения. При копировании экземпляров класса `sender` копируется указатель на очередь, а не сама очередь.

Листинг B.2. Класс sender

```
namespace messaging
{
    class sender
    {
        queue*q;
    public:
        sender():
            q(nullptr)
        {}
        explicit sender(queue*q_):
            q(q_)
        {}
        template<typename Message>
```

sender является оболочкой вокруг указателя на очередь

Изначально созданный экземпляр класса sender не имеет очереди

Разрешение создания очереди из указателя

```
        void send(Message const& msg)
        {
            if(q)
            {
                q->push(msg);
            }
        }
    };
}
```

При отправке сообщение помещается в очередь

Получение сообщений устроено несколько сложнее. Приходится не только ждать сообщения из очереди, но также и проверять, соответствует ли его тип любому ожидаемому типу, и вызывать соответствующую функцию обработки. Все это начинается с класса `receiver`, показанного в листинге B.3.

Листинг B.3. Класс receiver

```
namespace messaging
{
    class receiver
    {
        queue q;
    public:
        operator sender()
        {
            return sender(&q);
        }
        dispatcher wait()
        {
            return dispatcher(&q);
        }
    };
}
```

receiver владеет очередью

Разрешение подразумеваемого преобразования в объект sender, ссылающийся на очередь

Ожидание, пока очередь не создаст dispatcher

Если `sender` ссылается на очередь сообщений, то `receiver` владеет этой очередью. Можно получить объект `sender`, ссылающийся на очередь, путем использования подразумеваемого преобразования. Работа по диспетчеризации сообщений начинается с вызова функции `wait()`. При этом создается объект `dispatcher`, ссылающийся на очередь из объекта `receiver`. Класс `dispatcher` показан в листинге B.4, где можно увидеть, что работа выполнена в деструкторе. В данном случае работа состоит из ожидания сообщения и его диспетчеризации.

Листинг B.4. Класс dispatcher

```
namespace messaging
{
    class close_queue
    {};
    class dispatcher
    {
        queue* q;
        bool chained;
        dispatcher(dispatcher const&)=delete;
```

Сообщение о закрытии очереди

Экземпляры dispatcher нельзя скопировать

```
        dispatcher& operator=(dispatcher const&)=delete;
        template<
            typename Dispatcher,
            typename Msg,
            typename Func>
        friend class TemplateDispatcher;
        void wait_and_dispatch()
        {
            for(;;)
            {
                auto msg=q->wait_and_pop();
                dispatch(msg);
            }
        }
        bool dispatch(
            std::shared_ptr<message_base> const& msg)
        {
            if(dynamic_cast<wrapped_message<close_queue>*>(msg.get()))
            {
                throw close_queue();
            }
            return false;
        }
    public:
        dispatcher(dispatcher&& other):
            q(other.q),chained(other.chained)
        {
            other.chained=true;
        }
        explicit dispatcher(queue* q_):
            q(q_),chained(false)
        {}
        template<typename Message,typename Func>
        TemplateDispatcher<dispatcher,Message,Func>
        handle(Func&& f)
        {
            return TemplateDispatcher<dispatcher,Message,Func>(
                q,this,std::forward<Func>(f));
        }
        ~dispatcher() noexcept(false)
        {
            if(!chained)
            {
                wait_and_dispatch();
            }
        }
    };
}
```

Разрешение экземплярам TemplateDispatcher обращаться к внутренним компонентам класса

❶ Цикл, ожидание и диспетчеризация сообщений

❷ dispatch() проверяет наличие сообщения close_queue и выдает исключение

Экземпляры dispatcher могут перемещаться

Источник не должен ждать сообщений

❸ Обработка конкретного типа сообщения с помощью TemplateDispatcher

❹ Деструктор может выдать исключение

Экземпляр `dispatcher`, возвращенный функцией `wait()`, будет тут же уничтожен, поскольку он временный и, как мы говорили, вся работа выполняется в деструкторе. Деструктор вызывает функцию `wait_and_dispatch()`, представляющую собой цикл ❶, в котором ожидается сообщение, передаваемое функции `dispatch()`. Сама

функция `dispatch()` ❷ предельно проста, она проверяет, не поступило ли сообщение о закрытии очереди `close_queue`, и выдает исключение, если такое сообщение поступило; в противном случае она возвращает `false`, чтобы показать, что это сообщение не было обработано. Из-за этого исключения `close_queue` деструктор помечен как `noexcept(false)`; без этой аннотации спецификацией исключения по умолчанию для деструктора стала бы `noexcept(true)` ❹, показывающая невозможность выдачи исключения, и тогда исключение `close_queue` привело бы к прекращению выполнения программы.

Но сама по себе функция `wait()` вызывается нечасто; большую часть времени приходится тратить на обработку сообщения. И здесь на первый план выходит компонентная функция `handle()` ❸. Это шаблон, и тип сообщения не выводится, поэтому нужно указать тип обрабатываемого сообщения и передать функцию (или вызываемый объект) для его обработки. Сама функция `handle()` для обработки сообщений указанного типа передает очередь, текущий объект `dispatcher` и функцию-обработчик новому экземпляру шаблона класса `TemplateDispatcher`, код которого показан в листинге B.5. Именно поэтому значение `chained` проверяется в деструкторе до ожидания сообщений. Тем самым предотвращается не только ожидание сообщений перемещенными откуда-то объектами, но и разрешается переложить ответственность за ожидание на ваш новый экземпляр `TemplateDispatcher`.

Листинг B.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;
```

Экземпляры TemplateDispatcher дружат друг с другом

Если сообщение обрабатывается, нужно выйти из цикла ❶

Проверка типа сообщения и вызов функции ❷

```
            }
            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;
        }
        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();
            }
        }
    };
}
```

❸ Привязка к предыдущему диспетчеру

❹ Дополнительные обработчики могут быть выстроены в цепочку

❺ Деструктор снова помечен как noexcept(false)

Шаблон класса `TemplateDispatcher<>` смоделирован по образцу класса `dispatcher` и почти идентичен этому классу. В частности, деструктор по-прежнему вызывает функцию `wait_and_dispatch()`, чтобы дождаться сообщения.

Поскольку, если сообщение обрабатывается, исключение не выдается, теперь нужно проверить, обработано ли сообщение в вашем цикле сообщений ❶. Обработка сообщений прекращается, как только сообщение успешно обработано, чтобы в очередной раз можно было ожидать другого набора сообщений. Если получено совпадение с указанным типом сообщения, то вместо выдачи исключения вызывается предоставленная функция ❷ (хотя функция-обработчик сама может выдать исключение). Если совпадение не получено, выполняется переход по цепочке к предыдущему диспетчеру ❸. В первом экземпляре это будет объект `dispatcher`, но если выстроить цепочку из вызовов `handle()` ❹, позволяя обрабатывать несколько типов сообщений, это может быть предыдущий экземпляр `TemplateDispatcher<>`, который в случае несовпадения типа сообщения, в свою очередь, обратится по цепочке к предыдущему обработчику. Поскольку любой из обработчиков может выдать

исключение (не исключая обработчик диспетчера по умолчанию, предназначенного для обработки сообщений `close_queue`), деструктор должен быть еще раз помечен как `noexcept(false)` ❺.

Эта простая среда позволяет помещать в очередь сообщение любого типа, а затем на стороне получателя выбирать путем проверки на совпадение сообщения, подлежащие обработке. Она также позволяет раздавать ссылку на очередь для помещения в нее сообщений, сохраняя закрытый характер получающей стороны.

Для завершения примера из главы 4 в листинге B.6 показан код сообщений, в листингах B.7, B.8 и B.9 — код различных конечных автоматов, а в листинге B.10 — управляющий код.

Листинг B.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 B.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 B.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 B.9. Конечный автомат интерфейса пользователя

```
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 B.10. Управляющий код

```
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();
}
```

Г Справочник по C++ Thread Library

Г.1. Заголовок <chrono>

Заголовок `<chrono>` предоставляет классы для представления моментов времени, продолжительности `duration` и часов, которые служат источником объектов `time_point`. В каждом классе часов есть статический компонент `is_steady`, показывающий, являются ли данные часы *стабильными* (это значит, что их не нужно подводить). 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;
    }
}
```

Г.1.1. Шаблон класса std::chrono::duration

Шаблон класса `std::chrono::duration` предназначен для представления продолжительности. Параметры шаблона `Rep` и `Period` являются соответственно типами данных для хранения значения продолжительности и экземпляра шаблона класса `std::ratio`, которым задается временной промежуток (в долях секунды) между последовательными тактами. Например, `std::chrono::duration<int, std::milli>` определяет количество миллисекунд, которое может храниться в значении типа `int`, `std::chrono::duration<short, std::ratio<1,50>>` — количество 50-х долей секунды, которое может храниться в значении типа `short`, а `std::chrono::duration<long long, std::ratio<60,1>>` — количество минут, которое может храниться в значении 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 или подобным числовому типом, определенным пользователем. `Period` должен быть экземпляром, созданным по шаблону `std::ratio<>`.

Псевдоним типа std::chrono::duration::rep

Это псевдоним типа для хранения числа тактов в значении `duration`.

Объявление

```
typedef Rep rep;
```

`rep` — тип значения, используемого для хранения внутреннего представления объекта `duration`.

Псевдоним типа std::chrono::duration::period

Это псевдоним типа для создания экземпляра по шаблону класса `std::ratio`, с помощью которого задается количество долей секунды, представляемых счетчиком про-

должительности. Например, если `period` — это `std::ratio<1,50>`, то объект `duration`, для которого `count()` возвращает значение *N*, представляет *N* 50-х долей секунды.

Объявление

```
typedef Period period;
```

Конструктор по умолчанию std::chrono::duration

Создает экземпляр `std::chrono::duration` со значением по умолчанию.

Объявление

```
constexpr duration() = default;
```

Результат

Внутреннее значение `duration` (типа `rep`) инициализируется значением по умолчанию.

Конструктор преобразования из значения счетчика td::chrono::duration

Создает 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 переопределения при условии, что `Rep2` может быть подразумеваемо преобразован в `Rep`, а также что либо `Rep` является типом с плавающей точкой, либо `Rep2` не является типом с плавающей точкой.

Постусловие

```
this->count()==static_cast<rep>(r)
```

Конструктор преобразования std::chrono::duration из другого значения std::chrono::duration

Создает экземпляр `std::chrono::duration` путем масштабирования значения 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 переопределения при условии, что `Rep` является типом с плавающей точкой или `Rep2` не является типом с плавающей точкой, а `Period2` является целым числом, кратным `Period` (то есть `ratio_divide<Period2,Period>::den==1`). Это позволяет избежать случайного усечения (и соответствующей потери точности) из-за сохранения продолжительности с меньшими периодами в переменной, представляющей продолжительность с большим периодом.

Постусловие

```
this->count()==duration_cast<duration<Rep,Period>>(d).count()
```

Примеры

```
duration<int,ratio<1,1000>> ms(5);
duration<int,ratio<1,1>> s(ms);
duration<double,ratio<1,1>> s2(ms);
duration<int,ratio<1,1000000>> us(ms);
```

Пять миллисекунд

Ошибка: сохранить ms в виде целых секунд невозможно

Все в порядке: s2.count()==0.005

Все в порядке: us.count()==5000

Компонентная функция std::chrono::duration::count

Извлекает значение продолжительности.

Объявление

```
constexpr rep count() const;
```

Возвращает

Внутреннее значение объекта `duration` в виде значения типа `rep`.

Унарный оператор плюс std::chrono::duration::operator+

Пустая операция: просто возвращает копию `*this`.

Объявление

```
constexpr duration operator+() const;
```

Возвращает

```
*this
```

Унарный оператор минус std::chrono::duration::operator–

Возвращает продолжительность таким образом, что значением `count()` становится отрицательное значение `this->count()`.

Объявление

```
constexpr duration operator-() const;
```

Возвращает

```
duration(-this->count());
```

Оператор прединкремента std::chrono::duration::operator++

Увеличивает значение внутреннего счетчика на единицу.

Объявление

```
duration& operator++();
```

Результат

```
++this->internal_count;
```

Возвращает

```
*this
```

Оператор постинкремента std::chrono::duration::operator++

Увеличивает значение внутреннего счетчика на единицу и возвращает значение `*this`, предшествовавшее увеличению.

Объявление

```
duration operator++(int);
```

Результат

```
duration temp(*this);
++(*this);
return temp;
```

Оператор предекремента std::chrono::duration::operator–

Уменьшает значение внутреннего счетчика на единицу.

Объявление

```
duration& operator–();
```

Результат

```
--this->internal_count;
```

Возвращает

```
*this
```

Оператор постдекремента std::chrono::duration::operator––

Уменьшает значение внутреннего счетчика на единицу и возвращает значение `*this`, предшествовавшее уменьшению.

Объявление

```
duration operator––(int);
```

Результат

```
duration temp(*this);
--(*this);
return temp;
```

Составной оператор присваивания std::chrono::duration::operator+=

Прибавляет значение счетчика другого объекта `duration` к значению внутреннего счетчика для `*this`.

Объявление

```
duration& operator+=(duration const& other);
```

Результат

```
internal_count+=other.count();
```

Возвращает

```
*this
```

Составной оператор присваивания std::chrono::duration::operator–=

Вычитает значение счетчика другого объекта `duration` из значения внутреннего счетчика для `*this`.

Объявление

```
duration& operator-=(duration const& other);
```

Результат

```
internal_count-=other.count();
```

Возвращает

```
*this
```

Составной оператор присваивания std::chrono::duration::operator*=

Умножает значение внутреннего счетчика для `*this` на заданное значение.

Объявление

```
duration& operator*=(rep const& rhs);
```

Результат

```
internal_count*=rhs;
```

Возвращает

```
*this
```

Составной оператор присваивания std::chrono::duration::operator/=

Делит значение внутреннего счетчика `*this` на указанное значение.

Объявление

```
duration& operator/=(rep const& rhs);
```

Результат

```
internal_count/=rhs;
```

Возвращает

```
*this
```

Составной оператор присваивания std::chrono::duration::operator%=

Подгоняет значение внутреннего счетчика для `*this` под остаток от деления на заданное значение.

Объявление

```
duration& operator%=(rep const& rhs);
```

Результат

```
internal_count%=rhs;
```

Возвращает

```
*this
```

Составной оператор присваивания std::chrono::duration::operator%=

Подгоняет значение внутреннего счетчика для `*this` под остаток от деления на заданное значение на значение счетчика другого объекта `duration`.

Объявление

```
duration& operator%=(duration const& rhs);
```

Результат

```
internal_count%=rhs.count();
```

Возвращает

```
*this
```

Статическая компонентная функция std::chrono::duration::zero

Возвращает объект `duration`, представляющий нулевое значение.

Объявление

```
constexpr duration zero();
```

Возвращает

```
duration(duration_values<rep>::zero());
```

Статическая компонентная функция std::chrono::duration::min

Возвращает объект `duration`, содержащий минимальное возможное значение для указанного экземпляра.

Объявление

```
constexpr duration min();
```

Возвращает

```
duration(duration_values<rep>::min());
```

Статическая компонентная функция std::chrono::duration::max

Возвращает объект `duration`, содержащий максимальное возможное значение для указанного экземпляра.

Объявление

```
constexpr duration max();
```

Возвращает

```
duration(duration_values<rep>::max());
```

Оператор сравнения для определения равенства std::chrono::duration

Сравнивает два объекта `duration` для определения равенства, даже если у них 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

Подразумеваемое преобразование должно быть либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Результат

Если `CommonDuration` является синонимом для `std::common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type`, то `lhs==rhs` возвращает `CommonDuration(lhs).count()==CommonDuration(rhs).count()`.

Оператор сравнения для определения неравенства std::chrono::duration

Сравнивает два объекта `duration` для определения неравенства, даже если у них 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

Должно быть подразумеваемое преобразование либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они

являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Возвращает

`!(lhs==rhs)`

Оператор сравнения «меньше» std::chrono::duration

Сравнивает два объекта `duration` для определения, не является ли один меньше другого, даже если у них 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

Должно быть подразумеваемое преобразование либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Результат

Если `CommonDuration` является синонимом для `std::common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type`, то `lhs<rhs` возвращает `CommonDuration(lhs).count()<CommonDuration(rhs).count()`.

Оператор сравнения «больше» std::chrono::duration

Сравнивает два объекта `duration` для определения, не является ли один больше другого, даже если у них 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

Должно быть подразумеваемое преобразование либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Возвращает

`rhs<lhs`

Оператор сравнения «меньше или равно» std::chrono::duration

Сравнивает два объекта `duration` для определения, не является ли один меньше другого или равным ему, даже если у них 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

Должно быть подразумеваемое преобразование либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Возвращает

```
!(rhs<lhs)
```

Оператор сравнения «больше или равно» std::chrono::duration

Сравнивает два объекта `duration` для определения, не является ли один больше другого или равным ему, даже если у них 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

Должно быть подразумеваемое преобразование либо `lhs` в `rhs`, либо наоборот. Если же такое преобразование невозможно ни для одной из частей или же они являются несхожими экземплярами `duration`, но каждая из них подразумеваемо может быть преобразована в другую, значит, выражение сформировано неверно.

Возвращает

```
!(lhs<rhs)
```

Некомпонентная функция std::chrono::duration_cast

Выполняет явное преобразование объекта `std::chrono::duration` в заданный экземпляр `std::chrono::duration`.

Объявление

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

Требования

Объект `ToDuration` должен быть экземпляром класса `std::chrono::duration`.

Возвращает

Продолжительность `d`, преобразованная в тип продолжительности, заданный объектом `ToDuration`. Преобразование выполняется таким образом, чтобы минимизировать любые вызванные им потери точности между различными масштабами и типами представления.

Г.1.2. Шаблон класса std::chrono::time_point

Шаблон класса `std::chrono::time_point` позволяет представить момент времени, измеренный по конкретным часам. Задается в виде продолжительности со времени начала эпохи отсчета этих конкретных часов. Параметр шаблона `Clock` идентифицирует часы (у разных часов должен быть уникальный тип), а параметр шаблона `Duration` является типом, используемым для измерения продолжительности со времени начала эпохи и должен быть экземпляром шаблона класса `std::chrono::duration`. По умолчанию значением `Duration` должен быть исходный тип продолжительности шаблона `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();
};
```

Конструктор по умолчанию std::chrono::time_point

Создает объект `time_point`, представляющий продолжительность с начала эпохи отсчета, задаваемой связанным шаблоном `Clock`; внутренняя продолжительность инициализируется значением `Duration::zero()`.

Объявление

```
time_point();
```

Постусловие

Для только что созданного исходного объекта `time_point` по имени `tp` `tp.time_since_epoch()==tp::duration::zero()`.

Конструктор продолжительности std::chrono::time_point

Создает объект `time_point`, представляющий продолжительность с начала эпохи отсчета, задаваемой связанным шаблоном `Clock`.

Объявление

```
explicit time_point(const duration& d);
```

Постусловие

Для объекта `time_point` по имени `tp`, созданного с использованием конструктора `tp(d)` для некоторой продолжительности `d`, `tp.time_since_epoch()==d`.

Конструктор преобразования std::chrono::time_point

Создает объект `time_point` из другого объекта `time_point` с таким же типом `Clock`, но с другим типом `Duration`.

Объявление

```
template <class Duration2>
time_point(const time_point<clock, Duration2>& t);
```

Требования

Тип `Duration2` подразумеваемо должен преобразовываться в `Duration`.

Результат

Как при вычислении `time_point(t.time_since_epoch())`.

Значение, возвращаемое функцией `t.time_since_epoch()`, проходит подразумеваемое преобразование в объект типа `Duration`, и это значение сохраняется в только что созданном объекте `time_point`.

Компонентная функция std::chrono::time_point::time_since_epoch

Извлекает продолжительность с начала эпохи отсчета для конкретного объекта `time_point`.

Объявление

```
duration time_since_epoch() const;
```

Возвращает

Значение `duration`, хранящееся в `*this`.

Составной оператор присваивания std::chrono::time_point::operator+=

Прибавляет заданную продолжительность к значению, сохраненному в указанном объекте `time_point`.

Объявление

```
time_point& operator+=(const duration& d);
```

Результат

Прибавляет `d` к внутреннему объекту продолжительности `*this`, как при вычислении `this->internal_duration += d`.

Возвращает

`*this`

Составной оператор присваивания std::chrono::time_point::operator–=

Вычитает заданную продолжительность из значения, сохраненного в указанном объекте `time_point`.

Объявление

```
time_point& operator-=(const duration& d);
```

Результат

Вычитает `d` из внутреннего объекта продолжительности `*this`, как при вычислении `this->internal_duration-=d`.

Возвращает

`*this`

Статическая компонентная функция std::chrono::time_point::min

Получает объект `time_point`, представляющий минимально возможное для данного типа значение.

Объявление

```
static constexpr time_point min();
```

Возвращает

```
time_point(time_point::duration::min())
```

Статическая компонентная функция std::chrono::time_point::max

Получает объект `time_point`, представляющий максимально возможное для данного типа значение.

Объявление

```
static constexpr time_point max();
```

Возвращает

```
time_point(time_point::duration::max())
```

Г.1.3. Класс std::chrono::system_clock

Класс `std::chrono::system_clock` предоставляет средства для получения текущего физического времени. Текущее время можно получить, вызвав функцию `std::chrono::system_clock::now()`. Экземпляры класса `std::chrono::system_clock::time_point` с помощью функций `std::chrono::system_clock::to_time_t()` и `std::chrono::system_clock::to_time_point()` можно преобразовать в `time_t` и обратно. Системные часы часто нестабильны, поэтому при последующем обращении к `std::chrono::system_clock::now()` можно возвратить время, предшествующее времени, возвращенному предыдущим вызовом (например, если часы операционной системы были вручную переведены или синхронизированы с внешними часами).

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;
};
```

Псевдоним типа std::chrono::system_clock::rep

Псевдоним целочисленного типа, используемого для хранения количества тактов в значении продолжительности `duration`.

Объявление

```
typedef unspecified-integral-type rep;
```

Псевдоним типа std::chrono::system_clock::period

Псевдоним типа для экземпляра шаблона класса `std::ratio`, задающего наименьшее количество секунд (или долей секунды) между различными значениями `duration` или `time_point`. Псевдоним `period` определяет точность часов, а не частоту тактов.

Объявление

```
typedef std::ratio<unspecified,unspecified> period;
```

Псевдоним типа std::chrono::system_clock::duration

Экземпляр шаблона класса `std::chrono::duration`, способный сохранять разницу между любыми двумя моментами времени, возвращаемыми общесистемными часами реального времени.

Объявление

```
typedef std::chrono::duration<
std::chrono::system_clock::rep,
std::chrono::system_clock::period> duration;
```

Псевдоним типа std::chrono::system_clock::time_point

Экземпляр шаблона класса `std::chrono::time_point`, способный сохранять моменты времени, возвращаемые общесистемными часами реального времени.

Объявление

```
typedef std::chrono::time_point<std::chrono::system_clock> time_point;
```

Статическая компонентная функция std::chrono::system_clock::now

Получает текущее время от общесистемных часов реального времени.

Объявление

```
time_point now() noexcept;
```

Возвращает

Объект `time_point`, представляющий текущее время общесистемных часов реального времени.

Выдает

Исключение `std::system_error` в случае ошибки.

Статическая компонентная функция std::chrono::system_clock::to_time_t

Выполняет преобразование экземпляра `time_point` в `time_t`.

Объявление

```
time_t to_time_t(time_point const& t) noexcept;
```

Возвращает

Значение `time_t`, представляющее тот же момент времени, что и `t`, округленный или усеченный с точностью до секунд.

Выдает

Исключение `std::system_error` в случае ошибки.

Статическая компонентная функция std::chrono::system_clock::from_time_t

Выполняет преобразование экземпляра `time_t` в `time_point`.

Объявление

```
time_point from_time_t(time_t const& t) noexcept;
```

Возвращает

Значение `time_point`, представляющее тот же момент времени, что и `t`.

Выдает

Исключение `std::system_error` в случае ошибки.

Г.1.4. Класс std::chrono::steady_clock

Класс `std::chrono::steady_clock` предоставляет доступ к общесистемным стабильным часам. Текущее время можно получить, вызвав функцию `std::chrono::steady_clock::now()`. Фиксированной связи между значениями, возвращенными `std::chrono::steady_clock::now()`, и физическими часами не существует. Стабильные часы нельзя перевести назад, поэтому, если один вызов `std::chrono::steady_clock::now()` произошел до другого вызова `std::chrono::steady_clock::now()`, второй вызов должен возвратить значение времени, равное тому, что было возвращено первым вызовом, или более позднее. Часы идут как можно равномернее.

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;
};
```

Псевдоним типа std::chrono::steady_clock::rep

Этот псевдоним типа предназначен для целочисленного типа, используемого для хранения числа тактов в значении `duration`.

Объявление

```
typedef unspecified-integral-type rep;
```

Псевдоним типа std::chrono::steady_clock::period

Этот псевдоним типа предназначен для экземпляров шаблона класса `std::ratio`, определяющих наименьшее количество секунд (или долей секунды) между различными значениями `duration` или `time_point`. Псевдоним `period` определяет точность часов, а не частоту тактов.

Объявление

```
typedef std::ratio<unspecified,unspecified> period;
```

Псевдоним типа std::chrono::steady_clock::duration

Экземпляр шаблона класса `std::chrono::duration`, способный сохранять разницу между любыми двумя моментами времени, возвращаемыми общесистемными стабильными часами.

Объявление

```
typedef std::chrono::duration<
std::chrono::steady_clock::rep,
std::chrono::steady_clock::period> duration;
```

Псевдоним типа std::chrono::steady_clock::time_point

Экземпляр шаблона класса `std::chrono::time_point`, способный сохранять моменты времени, возвращаемые общесистемными стабильными часами.

Объявление

```
typedef std::chrono::time_point<std::chrono::steady_clock> time_point;
```

Статическая компонентная функция std::chrono::steady_clock::now

Получает текущее время от общесистемных стабильных часов.

Объявление

```
time_point now() noexcept;
```

Возвращает

Объект `time_point`, представляющий текущее время общесистемных стабильных часов.

Выдает

Исключение `std::system_error` в случае ошибки.

Синхронизация

Если один вызов `std::chrono::steady_clock::now()` происходит раньше другого, то объект `time_point`, возвращенный первым вызовом, должен быть меньше или равен моменту времени, возвращенному вторым вызовом.

Г.1.5. Псевдоним типа std::chrono::high_resolution_clock

Класс `std::chrono::high_resolution_clock` предоставляет доступ к общесистемным часам с наивысшим разрешением. Как и для всех остальных часов, текущее время можно получить, вызвав функцию `std::chrono::high_resolution_clock::now()`. Идентификатор `std::chrono::high_resolution_clock` может быть псевдонимом типа для класса `std::chrono::system_clock` или класса `std::chrono::steady_clock`, или может быть отдельным типом.

Хотя у типа `std::chrono::high_resolution_clock` самое высокое разрешение среди всех предоставляемых библиотекой часов, вызов функции `std::chrono::high_resolution_clock::now()` все же занимает определенное время. При хронометрировании скоротечных операций нужно брать в расчет издержки на вызов `std::chrono::high_resolution_clock::now()`.

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;
};
```

Г.2. Заголовок <condition_variable>

Заголовок `<condition_variable>` предоставляет условные переменные, являющиеся механизмами синхронизации базового уровня, позволяющими потоку быть заблокированным, пока не поступит уведомление о выполнении определенного условия или не истечет время ожидания, заданное тайм-аутом.

Содержимое заголовка

```
namespace std
{
    enum class cv_status { timeout, no_timeout };

    class condition_variable;
    class condition_variable_any;
}
```

Г.2.1. Класс std::condition_variable

Класс `std::condition_variable` позволяет потоку ждать, пока условие не получит значение `true`. Экземпляры `std::condition_variable` не являются `CopyAssignable`, `CopyConstructible`, `MoveAssignable` или `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>);
```

Конструктор по умолчанию std::condition_variable

Создает объект `std::condition_variable`.

Объявление

```
condition_variable();
```

Результат

Создает новый экземпляр `std::condition_variable`.

Выдает

Исключение `std::system_error`, если условная переменная не была создана.

Деструктор std::condition_variable

Уничтожает объект `std::condition_variable`.

Объявление

```
~condition_variable();
```

Предусловия

Отсутствие потоков, заблокированных на `*this` в вызове `wait()`, `wait_for()` или `wait_until()`.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::condition_variable::notify_one

Пробуждает один из потоков, ожидающих условной переменной `std::condition_variable`.

Объявление

```
void notify_one() noexcept;
```

Результат

Пробуждает в точке вызова один из потоков, ожидающих `*this`. При отсутствии ожидающих потоков вызов не дает никакого результата.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable::notify_all

Пробуждает все потоки, ожидающие на `std::condition_variable`.

Объявление

```
void notify_all() noexcept;
```

Результат

Пробуждает в точке вызова все потоки, ожидающие на `*this`. При отсутствии ожидающих потоков вызов не дает никакого результата.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable::wait

Ожидает пробуждения `std::condition_variable` вызовом `notify_one()`/`notify_all()` или ложного пробуждения.

Объявление

```
void wait(std::unique_lock<std::mutex>& lock);
```

Предусловия

`lock.owns_lock()` возвращает `true`, и блокировка принадлежит вызывающему потоку.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait()` объект `lock` снова блокируется.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait()`, то на выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait(), принимающее предикат. Или же рекомендуется вызывать wait() в цикле, проверяющем предикат, связанный с условной переменной.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable::wait, принимающее предикат

Ожидает пробуждения `std::condition_variable` вызовом `notify_one()` или `notify_all()` и вычисления предиката в `true`.

Объявление

```
template<typename Predicate>
void wait(std::unique_lock<std::mutex>& lock,Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Выражение `lock.owns_lock()` должно вычисляться в `true`, а владельцем блокировки `lock` должен быть поток, вызывающий функцию `wait()`.

Результат

Как при вычислении:

```
while(!pred())
{
    wait(lock);
}
```

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred и выход из функции произойдет только при условии, что в результате вычисления (bool)pred() будет возвращено значение true.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable::wait_for

Ожидает, пока `std::condition_variable` не будет оповещена вызовом `notify_one()` или `notify_all()`, или пока не истечет определенный период времени, или пока поток не получит 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

`lock.owns_lock()` возвращает `true`, и блокировка принадлежит вызывающему потоку.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока, или же пока не пройдет период времени, заданный аргументом `relative_time`, или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait_for()` объект `lock` снова блокируется.

Возвращает

`std::cv_status::no_timeout`, если поток был разбужен вызовом `notify_one()`, вызовом `notify_all()` или в результате ложного пробуждения, в противном случае возвращает `std::cv_status::timeout`.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait_for()`, то при выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait_for() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait_for(), принимающее предикат. Или же рекомендуется вызывать wait_for() в цикле, проверяющем предикат, связанный с условной переменной. При работе с этой функцией особое внимание нужно обратить на допустимость тайм-аута; при многих обстоятельствах более подходящей может оказаться функция wait_until(). Поток можноь заблокировать на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable::wait_for, принимающее предикат

Ожидает, пока `std::condition_variable` не будет оповещена вызовом `notify_one()` или `notify_all()` и предикат не будет вычислен в `true` или пока не истечет определенный период времени.

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 допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Выражение `lock.owns_lock()` должно вычисляться в `true`, а владельцем блокировки `lock` должен быть поток, вызывающий функцию `wait_for()`.

Результат

Как при вычислении

```
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;
```

Возвращает

`true`, если самый последний вызов `pred()` завершился возвращением `true`, или `false`, если период времени, указанный с помощью `relative_time`, истек и функция `pred()` вернула `false`.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred, и выход из функции произойдет только при условии, что в результате вычисления (bool)pred() будет возвращено значение true или период времени, указанный с помощью relative_time, истек. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable::wait_until

Ожидает, пока `std::condition_variable` не будет оповещена вызовом `notify_one()` или `notify_all()`, пока не наступит указанный момент времени или пока поток не получит 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

`lock.owns_lock()` возвращает `true`, и блокировка принадлежит вызывающему потоку.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока, или же пока вызовом `Clock::now()` не будет возвращено время, равное или большее значения `absolute_time`, или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait_until()` объект `lock` снова блокируется.

Возвращает

`std::cv_status::no_timeout`, если поток был разбужен вызовом `notify_one()`, вызовом `notify_all()` или в результате ложного пробуждения, в противном случае возвращает `std::cv_status::timeout`.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait_until()`, то при выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait_until() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait_until(), принимающее предикат. Или же рекомендуется вызывать wait_until() в цикле, проверяющем предикат, связанный с условной переменной. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное или большее absolute_time в том месте, где поток стал разблокированным.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable::wait_until, принимающее предикат

Ожидает, пока `std::condition_variable` не будет оповещена вызовом `notify_one()` или `notify_all()` и предикат не будет вычислен в `true` или пока не наступит 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 допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Выражение `lock.owns_lock()` должно вычисляться в `true`, а владельцем блокировки `lock` должен быть поток, вызывающий функцию `wait()`.

Результат

Как при вычислении

```
while(!pred())
{
    if(wait_until(lock,absolute_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

Возвращает

`true`, если самый последний вызов `pred()` завершился возвращением `true`, или `false`, если в результате вызова `Clock::now()` было возвращено время, равное или большее времени, указанного с помощью `absolute_time`, и функция `pred()` вернула `false`.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred, и выход из функции произойдет только при условии, что в результате вычисления (bool)pred() будет возвращено значение true, или в результате вызова Clock::now() будет возвращено время, равное или большее времени, указанного с помощью absolute_time. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное или большее absolute_time в том месте, где поток стал разблокированным.

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Некомпонентная функция std::notify_all_at_thread_exit

Пробуждает все потоки, ожидающие определенной условной переменной `std::condition_variable`, при выходе из текущего потока.

Объявление

```
void notify_all_at_thread_exit(
    condition_variable& cv,unique_lock<mutex> lk);
```

Предусловия

`lock.owns_lock()` возвращает `true`, и блокировка принадлежит вызывающему потоку. Функция `lk.mutex()` должна возвращать такое же значение, как для любого объекта блокировки, предоставляемого компонентным функциям `wait()`, `wait_for()` или `wait_until()` объекта `cv` из конкурентно ожидающих потоков.

Результат

Передает владение блокировкой, удерживаемой `lk`, во внутреннее хранилище и планирует уведомление `cv` при выходе из текущего потока. Это уведомление должно быть таким же, как и при вычислении

```
lk.unlock();
cv.notify_all();
```

Выдает

Исключение `std::system_error`, если результат не может быть достигнут.

ПРИМЕЧАНИЕ

> Блокировка удерживается до выхода из потока, поэтому следует принять меры, позволяющие избежать взаимной блокировки. Рекомендуется как можно более быстрый выход из вызывающего потока и отказ от применения в нем каких-либо блокирующих операций.

Пользователь должен обеспечить невозможность ошибочного предположения со стороны ожидающих потоков факта выхода из вызывающего потока при их пробуждении, в частности по причине ложных пробуждений. Этой цели можно добиться путем тестирования в ожидающем потоке предиката, который приобретает значение `true` только в уведомляющем потоке, находящемся под защитой мьютекса и без снятия блокировки мьютекса до вызова `notify_all_at_thread_exit`.

Г.2.2. Класс std::condition_variable_any

Класс `std::condition_variable_any` позволяет потоку ожидать вычисления условия в `true`. Объект `std::condition_variable` может использоваться только с блокиров-

кой типа `std::unique_lock<std::mutex>`, а объект `std::condition_variable_any` может использоваться с любым типом блокировки, отвечающим требованиям концепции `Lockable`.

Экземпляры `std::condition_variable_any` не являются `CopyAssignable`, `CopyConstructible`, `MoveAssignable` и `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);
};
```

Конструктор по умолчанию std::condition_variable_any

Создает объект `std::condition_variable_any`.

Объявление

```
condition_variable_any();
```

Результат

Создает новый экземпляр `std::condition_variable_any`.

Выдает

Исключение `std::system_error`, если условная переменная не может быть создана.

Деструктор std::condition_variable_any

Уничтожает объект `std::condition_variable_any`.

Объявление

```
~condition_variable_any();
```

Предусловия

Отсутствие потоков, заблокированных на `*this` в вызове `wait()`, `wait_for()` или `wait_until()`.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::condition_variable_any::notify_one

Пробуждает один из потоков, ожидающих определенной условной переменной `std::condition_variable_any`.

Объявление

```
void notify_one() noexcept;
```

Результат

Пробуждает в точке вызова один из потоков, ожидающих `*this`. При отсутствии ожидающих потоков вызов не дает никакого результата.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable_any` выстраиваются

в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable_any::notify_all

Пробуждает все потоки, ожидающие определенной условной переменной `std::condition_variable_any`.

Объявление

```
void notify_all() noexcept;
```

Результат

Пробуждает в точке вызова все потоки, ожидающие `*this`. При отсутствии ожидающих потоков вызов не дает никакого результата.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable_any` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable_any::wait

Ожидает, пока `std::condition_variable_any` не будет оповещена вызовом `notify_one()` или `notify_all()` или пока поток не получит ложное пробуждение.

Объявление

```
template<typename Lockable>
void wait(Lockable& lock);
```

Предусловия

Тип `Lockable` является `Lockable`, блокировкой владеет объект `lock`.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait()` объект `lock` снова блокируется.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait()`, то при выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait(), принимающее предикат. Или же рекомендуется вызывать wait() в цикле, проверяющем предикат, связанный с условной переменной.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable_any` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable_any::wait, принимающее предикат

Ожидает пробуждения `std::condition_variable_any` вызовом `notify_one()` или `notify_all()` и вычисления предиката в `true`.

Объявление

```
template<typename Lockable,typename Predicate>
void wait(Lockable& lock,Predicate pred);
```

Предусловия

Выражение `pred()` должно быть допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Тип `Lockable` является `Lockable`, и блокировкой владеет объект `lock`.

Результат

Как при вычислении

```
while(!pred())
{
    wait(lock);
}
```

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred, и выход из функции произойдет только при условии, что в результате вычисления (bool)pred() будет возвращено значение true.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в по-

следовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable_any::wait_for

Ожидает, пока `std::condition_variable_any` не будет оповещена вызовом `notify_one()` или `notify_all()`, или пока не истечет определенный период времени, или пока поток не получит 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` является `Lockable`, и блокировкой владеет объект `lock`.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока, или же пока не пройдет период времени, заданный аргументом `relative_time`, или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait_for()` объект `lock` снова блокируется.

Возвращает

`std::cv_status::no_timeout`, если поток был разбужен вызовом `notify_one()`, вызовом `notify_all()` или в результате ложного пробуждения, в противном случае возвращает `std::cv_status::timeout`.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait_for()`, то при выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait_for() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait_for(), принимающее предикат. Или же рекомендуется вызывать wait_for() в цикле, проверяющем предикат, связанный с условной переменной. При работе с этой функцией особое внимание нужно обратить на допустимость тайм-аута; при многих обстоятельствах более подходящей может оказаться функция wait_until(). Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable_any::wait_for, принимающее предикат

Ожидает, пока `std::condition_variable_any` не будет оповещена вызовом `notify_one()` или `notify_all()` и предикат не будет вычислен в `true` или пока не истечет определенный период времени.

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 допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Тип `Lockable` является `Lockable`, и блокировкой владеет объект `lock`.

Результат

Как при вычислении

```
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;
```

Возвращает

`true`, если самый последний вызов `pred()` завершился возвращением `true`, или `false`, если период времени, указанный с помощью `relative_time`, истек и функция `pred()` вернула `false`.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred, и выход из функции произойдет только при условии,

что в результате вычисления (bool)pred() будет возвращено значение true, или период времени, указанный с помощью relative_time, истек. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Компонентная функция std::condition_variable_any::wait_until

Ожидает, пока `std::condition_variable_any` не будет оповещена вызовом `notify_one()` или `notify_all()`, пока не наступит указанный момент времени или пока поток не получит 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` является `Lockable`, и блокировкой владеет объект `lock`.

Результат

Атомарно снимает блокировку с предоставленного объекта `lock` и блокирует поток, пока он не будет разбужен вызовом `notify_one()` или `notify_all()` из другого потока, или же пока вызовом `Clock::now()` не будет возвращено время, равное или большее значения `absolute_time`, или же пока не произойдет ложное пробуждение потока. Перед возвратом управления из вызова `wait_until()` объект `lock` снова блокируется.

Возвращает

`std::cv_status::no_timeout`, если поток был разбужен вызовом `notify_one()`, вызовом `notify_all()` или в результате ложного пробуждения, в противном случае возвращает `std::cv_status::timeout`.

Выдает

Исключение `std::system_error`, если результат не может быть достигнут. Если объект `lock` был разблокирован в ходе вызова `wait_until()`, то при выходе он снова блокируется, даже если выход из функции происходит по выдаче исключения.

ПРИМЕЧАНИЕ

Ложные пробуждения означают, что вызывавший функцию wait_until() поток можно разбудить, даже если ни один из потоков не вызывал функцию notify_one() или notify_all(). Поэтому по возможности рекомендуется применять переопределение функции wait_until(), принимающее предикат. Или же рекомендуется вызывать wait_until() в цикле, проверяющем предикат, связанный с условной переменной. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное или большее absolute_time, в том месте, где поток стал разблокированным.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Переопределение компонентной функции std::condition_variable_any::wait_until, принимающее предикат

Ожидает, пока `std::condition_variable_any` не будет оповещена вызовом `notify_one()`, или `notify_all()` и предикат не будет вычислен в `true`, или пока не наступит 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 допустимым и должно возвращать значение, допускающее преобразование в тип `bool`. Тип `Lockable` является `Lockable`, и блокировкой владеет объект `lock`.

Результат

Как при вычислении

```
while(!pred())
{
    if(wait_until(lock,absolute_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

Возвращает

`true`, если самый последний вызов `pred()` завершился возвращением `true`, или `false`, если в результате вызова `Clock::now()` было возвращено время, равное или большее времени, указанного с помощью `absolute_time`, и функция `pred()` вернула `false`.

ПРИМЕЧАНИЕ

Возможность ложных пробуждений означает, что количество вызовов pred не определено. Мьютекс, ссылающийся на блокировку, будет заблокирован при каждом вызове pred, и выход из функции произойдет только при условии, что в результате вычисления (bool)pred() будет возвращено значение true или в результате вызова Clock::now() будет возвращено время, равное или большее времени, указанного с помощью absolute_time. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное или большее absolute_time, в том месте, где поток стал разблокированным.

Выдает

Любое исключение, выданное при вызове `pred`, или исключение `std::system_error`, если результат не может быть достигнут.

Синхронизация

Вызовы `notify_one()`, `notify_all()`, `wait()`, `wait_for()` и `wait_until()` в отношении одного и того же экземпляра `std::condition_variable` выстраиваются в последовательность. Вызов `notify_one()` или `notify_all()` разбудит только те потоки, которые приступили к ожиданию до этого вызова.

Г.3. Заголовок <atomic>

Заголовок `<atomic>` предоставляет набор базовых атомарных типов и операций над этими типами и шаблон класса для создания атомарных версий, определяемых пользователем типов, отвечающих определенным критериям.

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) see description

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);
}
```

Г.3.1. Псевдонимы типа std::atomic_xxx

Для совместимости с предстоящим стандартом языка C предоставляются псевдонимы `typedef` для атомарных целочисленных типов. Для C++17 здесь должны быть псевдонимы `typedef` для соответствия специализации `std::atomic<T>`; для предыдущих стандартов C++ взамен этого они могут быть базовыми классами этой специализации с тем же интерфейсом (табл. Г.1).

Таблица Г.1. Атомарные псевдонимы typedef 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>

Г.3.2. Макросы ATOMIC_xxx_LOCK_FREE

Эти макросы определяют, являются ли свободными от блокировки атомарные типы, соответствующие конкретным встроенным типам.

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
```

ATOMIC_xxx_LOCK_FREE может принимать значения 0, 1 или 2. Если значение равно 0, то операции как над знаковыми, так и над беззнаковыми атомарными типами, соответствующими названному типу, не обладают свободой от блокировок. Если значение равно 1, то операции для одних экземпляров этих типов могут быть, а для других не могут быть свободными от блокировок. И если значение равно 2, то эти операции всегда свободны от блокировок. Например, если значение ATOMIC_INT_LOCK_FREE равно 2, то операции над экземплярами std::atomic<int> и std::atomic<unsigned> всегда будут свободными от блокировок.

Макрос ATOMIC_POINTER_LOCK_FREE дает описание свойству свободы от блокировок тех операций, которые выполняются над специализациями атомарного указателя std::atomic<T*>.

Г.3.3. Макрос ATOMIC_VAR_INIT

Макрос ATOMIC_VAR_INIT предоставляет средство инициализации атомарной переменной конкретным значением.

Объявление

```
#define ATOMIC_VAR_INIT(value) см. описание
```

Макрос расширяется до последовательности лексем, которой можно воспользоваться для инициализации одного из стандартных атомарных типов конкретным значением в выражении такого вида:

```
std::atomic<type> x = ATOMIC_VAR_INIT(val);
```

Указанное значение должно быть совместимо с неатомарным типом, соответствующим 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 является атомарной, и любое обращение к уже инициализируемой переменной со стороны другого потока, при котором инициализация не происходит до этого обращения, приводит к гонке за данными, а следовательно, и к неопределенному поведению.

Г.3.4. Перечисление std::memory_order

Перечисление std::memory_order используется для задания ограничений на порядок 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 различными значениями порядка доступа к памяти, ведут себя следующим образом (подробное описание ограничений доступа к памяти дано в главе 5).

std::memory_order_relaxed

Операция не обеспечивает каких-либо дополнительных ограничений порядка доступа к памяти.

std::memory_order_release

Операция относится к операциям освобождения конкретного места в памяти. Поэтому она синхронизируется с операцией захвата того же места в памяти для чтения сохраненного значения.

std::memory_order_acquire

Операция относится к операциям захвата конкретного места в памяти. Если сохраненное значение было записано операцией освобождения, то это сохранение синхронизируется с данной операцией.

std::memory_order_acq_rel

Эта операция должна относиться к операциям чтения — изменения — записи, и она ведет себя, как будто в отношении конкретного места в памяти задан порядок доступа как `std::memory_order_acquire`, так и `std::memory_order_release`.

std::memory_order_seq_cst

Эта операция формирует часть единого глобального порядка последовательно согласованных операций. Кроме того, если это операция сохранения, то она ведет себя как операция с семантикой `std::memory_order_release`; если это операция загрузки, то она ведет себя как операция с семантикой `std::memory_order_acquire`.

И если это операция чтения — изменения — записи, то она ведет себя и как операция с пометкой `std::memory_order_acquire`, и как операция с пометкой `std::memory_order_release`. Эта семантика является исходной для всех операций.

std::memory_order_consume

Эта операция относится к операциям потребления конкретного места в памяти. В стандарте C++17 утверждается, что это ограничение порядка доступа к памяти применяться не должно.

Г.3.5. Функция std::atomic_thread_fence

Функция `std::atomic_thread_fence()` вставляет в код «барьер» для принудительной установки ограничений порядка доступа к памяти между операциями.

Объявление

```
extern "C" void atomic_thread_fence(std::memory_order order);
```

Результат

Вставляет барьер с требуемыми ограничениями порядка доступа к памяти.

Барьер со значением параметра `order`, равным `std::memory_order_release`, `std::memory_order_acq_rel` или `std::memory_order_seq_cst`, синхронизируется с операцией захвата того же места в памяти, если эта операция захвата считывает значение, сохраненное атомарной операцией, следующей за барьером в том же потоке, в котором поставлен барьер.

Операция освобождения синхронизируется с барьером со значением параметра `order`, равным `std::memory_order_acquire`, `std::memory_order_acq_rel` или `std::memory_order_seq_cst`, если эта операция освобождения сохраняет значение, считываемое атомарной операцией, предшествующей барьеру в том же потоке, в котором поставлен барьер.

Выдает

Ничего не выдает.

Г.3.6. Функция std::atomic_signal_fence

Функция `std::atomic_signal_fence()` вставляет в код «барьер» для принудительной установки ограничений порядка доступа к памяти между операциями потока и операциями в обработчике сигнала этого же потока.

Объявление

```
extern "C" void atomic_signal_fence(std::memory_order order);
```

Результат

Вставляет «барьер» с требуемыми ограничениями порядка доступа к памяти. Эквивалентна функции `std::atomic_thread_fence(order)`, за исключением того, что ограничения применяются только между потоком и обработчиком сигнала в том же самом потоке.

Выдает

Ничего не выдает.

Г.3.7. Класс std::atomic_flag

Класс `std::atomic_flag` предоставляет самый простой атомарный флаг. Это единственный тип данных, гарантированно свободный от блокировок согласно стандарту C++11 (хотя в большинстве реализации свободными от блокировок будут многие атомарные типы).

Экземпляр `std::atomic_flag` является либо *установленным*, либо *сброшенным*.

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 неопределено
```

Конструктор по умолчанию std::atomic_flag

Состояние установки или сброса созданного конструктором по умолчанию экземпляра `std::atomic_flag` не определяется. Для объектов со статической продолжительностью хранения инициализация должна быть статической.

Объявление

```
std::atomic_flag() noexcept = default;
```

Результат

Создает новый объект `std::atomic_flag` в неопределенном состоянии.

Выдает

Ничего не выдает.

Инициализация std::atomic_flag с помощью макроса ATOMIC_FLAG_INIT

Экземпляр `std::atomic_flag` можно проинициализировать с помощью макроса `ATOMIC_FLAG_INIT`, в таком случае он инициализируется в сброшенном состоянии. Для объектов со статической продолжительностью хранения инициализация должна быть статической.

Объявление

```
#define ATOMIC_FLAG_INIT unspecified
```

Использование

```
std::atomic_flag flag=ATOMIC_FLAG_INIT;
```

Результат

Создает новый объект `std::atomic_flag` в сброшенном состоянии.

Выдает

Ничего не выдает.

Компонентная функция std::atomic_flag::test_and_set

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.

Возвращает

`true`, если флаг был установлен в месте вызова, или `false`, если флаг был сброшен.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это атомарная операция чтения — изменения — записи места в памяти, содержащего *this.

Некомпонентная функция std::atomic_flag_test_and_set

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();
```

Некомпонентная функция std::atomic_flag_test_and_set_explicit

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);
```

Компонентная функция std::atomic_flag::clear

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

Предоставляемый параметр `order` должен иметь одно из следующих значений: `std::memory_order_relaxed`, `std::memory_order_release` или `std::memory_order_seq_cst`.

Результат

Атомарно сбрасывает флаг.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это атомарная операция сохранения места в памяти, содержащего *this.

Некомпонентная функция std::atomic_flag_clear

Атомарно сбрасывает флаг.

Объявление

```
void atomic_flag_clear(volatile atomic_flag* flag) noexcept;
void atomic_flag_clear(atomic_flag* flag) noexcept;
```

Результат

```
flag->clear();
```

Некомпонентная функция std::atomic_flag_clear_explicit

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);
```

Г.3.8. Шаблон класса std::atomic

Класс `std::atomic` предоставляет оболочку с атомарными операциями для любого типа, удовлетворяющего следующим требованиям.

Параметр шаблона `BaseType` должен:

- иметь стандартный конструктор по умолчанию;
- иметь стандартный копирующий оператор присваивания;
- иметь стандартный деструктор;
- допускать поразрядное сравнение на равенство.

Это означает допустимость объектов `std::atomic<some-built-in-type>`, а также объектов `std::atomic<some-simple-struct>` и недопустимость объектов, подобных `std::atomic<std::string>`.

Кроме основного шаблона, имеются специализации для встроенных целочисленных типов и указателей для предоставления дополнительных операций, например `x++`.

Экземпляры `std::atomic` не являются `CopyConstructible` или `CopyAssignable`, поскольку соответствующие им операции не могут выполняться в рамках единой атомарной операции.

Определение класса

```
template<typename BaseType>
struct atomic
{
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined;
    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 некомпонентные функции определены как шаблоны, они могут предоставляться в виде набора переопределяемых функций, и явная спецификация аргументов шаблона применяться не должна.

Конструктор по умолчанию std::atomic

Создает экземпляр `std::atomic` со значением инициализации по умолчанию.

Объявление

```
atomic() noexcept;
```

Результат

Создает новый объект `std::atomic` со значением инициализации по умолчанию. Для объектов со статической продолжительностью хранения эта инициализация является статической.

ПРИМЕЧАНИЕ

При инициализации конструктором по умолчанию экземпляров std::atomic с нестатической продолжительностью хранения рассчитывать на предсказуемость значения не получится.

Выдает

Ничего не выдает.

Некомпонентная функция std::atomic_init

Неатомарно сохраняет предоставленное значение в экземпляре `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()` в отношении экземпляра `atomic<BaseType>`, который не был создан исходным образом или над которым с момента создания уже были проведены какие-либо операции, приводит к неопределенному поведению.

ПРИМЕЧАНИЕ

Поскольку сохранение носит неатомарный характер, любое конкурентное обращение к объекту, указанному с помощью p из другого потока (даже с применением атомарных операций), приводит к гонке за данными.

Выдает

Ничего не выдает.

Конструктор преобразования std::atomic

Создает экземпляр `std::atomic` с предоставленным значением `BaseType`.

Объявление

```
constexpr atomic(BaseType b) noexcept;
```

Результат

Создает новый объект `std::atomic` со значением `b`. Для объектов со статической продолжительностью хранения инициализация является статической.

Выдает

Ничего не выдает.

Оператор преобразования и присваивания std::atomic

Сохраняет новое значение в `*this`.

Объявление

```
BaseType operator=(BaseType b) volatile noexcept;
BaseType operator=(BaseType b) noexcept;
```

Результат

```
return this->store(b);
```

Компонентная функция std::atomic::is_lock_free

Определяет, являются ли операции над `*this` свободными от блокировок.

Объявление

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

Возвращает

`true`, если операции над `*this` являются свободными от блокировок, или `false` в противном случае.

Выдает

Ничего не выдает.

Некомпонентная функция std::atomic_is_lock_free

Определяет, 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();
```

Статический элемент данных std::atomic::is_always_lock_free

Определяет, всегда ли операции над всеми объектами данного типа являются свободными от блокировок.

Объявление

```
static constexpr bool is_always_lock_free() = implementation-defined;
```

Значение

`true`, если операции над объектами данного типа всегда являются свободными от блокировок, или `false` в противном случае.

Компонентная функция std::atomic::load

Атомарно загружает текущее значение экземпляра `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

Предоставляемый параметр `order` должен иметь одно из следующих значений: `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_consume` или `std::memory_order_seq_cst`.

Результат

Атомарно загружает значение, сохраненное в `*this`.

Возвращает

Значение, сохраненное в `*this` в месте вызова.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это атомарная операция загрузки места в памяти, содержащего *this.

Некомпонентная функция std::atomic_load

Атомарно загружает текущее значение экземпляра `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();
```

Некомпонентная функция std::atomic_load_explicit

Атомарно загружает текущее значение экземпляра `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);
```

Оператор преобразования в тип BaseType — std::atomic::operator

Загружает значение, сохраненное в `*this`.

Объявление

```
operator BaseType() const volatile noexcept;
operator BaseType() const noexcept;
```

Результат

```
return this->load();
```

Компонентная функция std::atomic::store

Атомарно сохраняет новое значение в экземпляре `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

Предоставляемый параметр `order` должен иметь одно из следующих значений: `std::memory_order_relaxed`, `std::memory_order_release` или `std::memory_order_seq_cst`.

Результат

Атомарно сохраняет значение `new_value` в `*this`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это атомарная операция сохранения места в памяти, содержащего *this.

Некомпонентная функция std::atomic_store

Атомарно сохраняет новое значение в экземпляре `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);
```

Некомпонентная функция std::atomic_store_explicit

Атомарно сохраняет новое значение в экземпляре `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);
```

Компонентная функция std::atomic::exchange

Атомарно сохраняет новое значение и считывает старое.

Объявление

```
BaseType exchange(
    BaseType new_value,
    memory_order order = memory_order_seq_cst)
    volatile noexcept;
```

Результат

Атомарно сохраняет `new_value` в `*this` и извлекает имеющееся значение `*this`.

Возвращает

Значение `*this`, существовавшее непосредственно перед сохранением.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это атомарная операция чтения — изменения — записи в отношении места в памяти, содержащего *this.

Некомпонентная функция std::atomic_exchange

Атомарно сохраняет новое значение в `atomic<BaseType>` и считывает старое.

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);
```

Некомпонентная функция std::atomic_exchange_explicit

Атомарно сохраняет новое значение в `atomic<BaseType>` и считывает старое.

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);
```

Компонентная функция std::atomic::compare_exchange_strong

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу. Если значения не равны, обновляет ожидаемое значение считанным значением.

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 ожидаемое значение `expected` со значением, сохраненным в `*this`, используя для этого поразрядное сравнение, и сохраняет `new_value` в `*this`, если значения равны. В противном случае обновляет `expected` считанным значением.

Возвращает

`true`, если имевшееся значение `*this` было равно значению `expected`, или `false` в противном случае.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это переопределение с тремя параметрами эквивалентно переопределению с четырьмя параметрами, где success_order==order и failure_order==order, за исключением того, что если order имеет значение std::memory_order_acq_rel, то failure_order имеет значение std::memory_order_acquire, а если order имеет значение std::memory_order_release, то failure_order имеет значение std::memory_order_relaxed.

ПРИМЕЧАНИЕ

Если при порядке доступа к памяти success_order результатом является true, то для места в памяти, содержащего *this, это будет атомарной операцией чтения — изменения — записи; в ином случае при порядке доступа к памяти failure_order для места в памяти, содержащего *this, это атомарная операция загрузки.

Некомпонентная функция std::atomic_compare_exchange_strong

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу. Если значения не равны, обновляет ожидаемое значение считанным значением.

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);
```

Некомпонентная функция std::atomic_compare_exchange_strong_explicit

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу. Если значения не равны, обновляет ожидаемое значение считанным значением.

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;
```

Компонентная функция std::atomic::compare_exchange_weak

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу и обновление можно выполнить атомарно. Если значения не равны или обновление нельзя выполнить атомарно, обновляет ожидаемое значение считанным значением.

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 поразрядно сравнивает ожидаемое значение со значением, сохраненным в `*this`, и сохраняет в `*this` значение `new_value`, если сравниваемые значения равны друг другу. Если значения не равны или обновление нельзя выполнить атомарно, обновляет ожидаемое значение считанным значением.

Возвращает

`true`, если имевшееся значение `*this` было равно значению `expected` и `new_value` было успешно сохранено в `*this`, или `false` в противном случае.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это переопределение с тремя параметрами эквивалентно переопределению с четырьмя параметрами, где success_order==order и failure_order==order, за исключением того, что если order имеет значение std::memory_order_acq_rel, то failure_order имеет значение std::memory_order_acquire, а если order имеет значение std::memory_order_release, то failure_order имеет значение std::memory_order_relaxed.

ПРИМЕЧАНИЕ

Если при порядке доступа к памяти success_order результатом является true, то для места в памяти, содержащего *this, это будет атомарной операцией чтения — изменения — записи; в противном случае при порядке доступа к памяти failure_order для места в памяти, содержащего *this, это атомарная операция загрузки.

Некомпонентная функция std::atomic_compare_exchange_weak

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу и обновление можно выполнить атомарно. Если значения не равны или обновление нельзя выполнить атомарно, обновляет ожидаемое значение считанным значением.

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);
```

Некомпонентная функция std::atomic_compare_exchange_weak_explicit

Атомарно сравнивает значение с ожидаемым значением и сохраняет новое значение, если они равны друг другу и обновление можно выполнить атомарно. Если значения не равны или обновление нельзя выполнить атомарно, обновляет ожидаемое значение считанным значением.

Объявление

```
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);
```

Г.3.9. Специализации шаблона std::atomic

Специализации шаблона класса `std::atomic` предоставляются для целочисленных и указательных типов. В дополнение к операциям, предоставляемым основным шаблоном, для целочисленных типов эти специализации предоставляют атомарные операции сложения, вычитания и поразрядные операции, а для типов указателей — атомарные арифметические операции надуказателями.

Специализации предоставляются 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>
```

И `std::atomic<T*>` для всех типов `T`.

Г.3.10. Специализации std::atomic<integral-type>

Специализации `std::atomic<integral-type>` шаблона класса `std::atomic` предоставляют атомарный целочисленный тип данных для каждого фундаментального целочисленного типа с полным набором операций.

К этим специализациям шаблона класса `std::atomic<>` применимы следующие описания:

```
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 являются `CopyConstructible` или `CopyAssignable`, поскольку соответствующие им операции не могут выполняться в рамках единой атомарной операции.

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;
```

Такую же семантику имеют и те операции, которые также предоставляются основным шаблоном (см. подраздел Г.3.8).

Компонентная функция std::atomic<integral-type>::fetch_add

Атомарно загружает значение и заменяет его суммой этого значения и предоставленного значения `i`.

Объявление

```
integral-type 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 извлекает имеющееся значение `*this` и сохраняет в `*this` сумму *`old-value`* + `i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_add

Атомарно считывает значение из экземпляра `atomic<integral-type>` и заменяет его значением плюс предоставленное значение `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);
```

Некомпонентная функция std::atomic_fetch_add_explicit

Атомарно считывает значение из экземпляра `atomic<integral-type>` и заменяет его значением плюс предоставленное значение `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);
```

Компонентная функция std::atomic<integral-type>::fetch_sub

Атомарно загружает значение и заменяет его разностью этого значения и предоставленного значения `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 извлекает имеющееся значение `*this` и сохраняет в `*this` разность *`old-value`* `- i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_sub

Атомарно считывает значение из экземпляра `atomic<integral-type>` и заменяет его значением минус предоставленное значение `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);
```

Некомпонентная функция std::atomic_fetch_sub_explicit

Атомарно считывает значение из экземпляра `atomic<integral-type>` и заменяет его значением с вычетом предоставленного значения `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);
```

Компонентная функция std::atomic<integral-type>::fetch_and

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 извлекает имеющееся значение `*this` и сохраняет в `*this` результат вычисления `old-value & i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_and

Атомарно считывает 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);
```

Некомпонентная функция std::atomic_fetch_and_explicit

Атомарно считывает 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);
```

Компонентная функция std::atomic<integral-type>::fetch_or

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 извлекает имеющееся значение `*this` и сохраняет в `*this` результат вычисления `old-value | i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_or

Атомарно считывает 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);
```

Некомпонентная функция std::atomic_fetch_or_explicit

Атомарно считывает 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);
```

Компонентная функция std::atomic<integral-type>::fetch_xor

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 извлекает имеющееся значение `*this` и сохраняет в `*this` результат вычисления *`old-value`* `^ i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_xor

Атомарно считывает 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);
```

Некомпонентная функция std::atomic_fetch_xor_explicit

Атомарно считывает 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);
```

Оператор прединкремента std::atomic<integral-type>::operator++

Атомарно увеличивает значение, сохраненное в `*this`, на единицу и возвращает новое значение.

Объявление

```
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
```

Результат

```
return this->fetch_add(1) + 1;
```

Оператор постинкремента std::atomic<integral-type>::operator++

Атомарно увеличивает на единицу значение, сохраненное в `*this`, и возвращает старое значение.

Объявление

```
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
```

Результат

```
return this->fetch_add(1);
```

Оператор предекремента std::atomic<integral-type>::operator––

Атомарно уменьшает на единицу значение, сохраненное в `*this`, и возвращает новое значение.

Объявление

```
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
```

Результат

```
return this->fetch_sub(1) - 1;
```

Оператор постдекремента std::atomic<integral-type>::operator––

Атомарно уменьшает на единицу значение, сохраненное в `*this`, и возвращает старое значение.

Объявление

```
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;
```

Результат

```
return this->fetch_sub(1);
```

Составной оператор присваивания std::atomic<integral-type>::operator+=

Атомарно складывает предоставленное значение со значением, сохраненным 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;
```

Составной оператор присваивания std::atomic<integral-type>::operator–=

Атомарно вычитает предоставленное значение из значения, сохраненного 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;
```

Составной оператор присваивания std::atomic<integral-type>::operator&=

Атомарно заменяет значение, сохраненное в `*this`, результатом операции «поразрядное И» между предоставленным значением и значением, сохраненным 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_and(i) & i;
```

Составной оператор присваивания std::atomic<integral-type>::operator|=

Атомарно заменяет значение, сохраненное в `*this`, результатом операции «поразрядное ИЛИ» между предоставленным значением и значением, сохраненным 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_or(i,std::memory_order_seq_cst) | i;
```

Составной оператор присваивания std::atomic<integral-type>::operator^=

Атомарно заменяет значение, сохраненное в `*this`, результатом операции «поразрядное исключающее ИЛИ» между предоставленным значением и значением, сохраненным 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_xor(i,std::memory_order_seq_cst) ^ i;
```

Частичная специализация std::atomic<T*>

Частичная специализация `std::atomic<T*>` шаблона класса `std::atomic` предоставляет атомарный тип данных для каждого типа указателя с полным набором операций.

Экземпляры `std::atomic<T*>` не являются `CopyConstructible` или `CopyAssignable`, поскольку соответствующие им операции не могут выполняться в рамках единой атомарной операции.

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;
```

Такую же семантику имеют и те операции, которые также предоставляются основным шаблоном (см. подраздел Г.3.8).

Компонентная функция std::atomic<T*>::fetch_add

Атомарно загружает значение и заменяет его суммой этого значения и предоставленного значения `i` с применением стандартных правил арифметики указателей, 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 извлекает имеющееся значение `*this` и сохраняет в `*this` результат вычисления `old-value + i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_add

Атомарно считывает значение из экземпляра `atomic<T*>` и заменяет его этим значением плюс предоставленное значение `i` с применением стандартных правил арифметики указателей.

Объявление

```
T* atomic_fetch_add(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_add(atomic<T*>* p, ptrdiff_t i) noexcept;
```

Результат

```
return p->fetch_add(i);
```

Некомпонентная функция std::atomic_fetch_add_explicit

Атомарно считывает значение из экземпляра `atomic<T*>` и заменяет его этим значением плюс предоставленное значение `i` с применением стандартных правил 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);
```

Компонентная функция std::atomic<T*>::fetch_sub

Атомарно загружает значение и заменяет его этим значением минус предоставленное значение `i` с применением стандартных правил арифметики указателей, 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 извлекает имеющееся значение `*this` и сохраняет в `*this` результат вычисления `old-value - i`.

Возвращает

Значение `*this`, непосредственно предшествующее сохранению.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Для места в памяти, содержащего *this, эта функция является атомарной операцией чтения — изменения — записи.

Некомпонентная функция std::atomic_fetch_sub

Атомарно считывает значение из экземпляра `atomic<T*>` и заменяет его этим значением минус предоставленное значение `i` с применением стандартных правил 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);
```

Некомпонентная функция std::atomic_fetch_sub_explicit

Атомарно считывает значение из экземпляра `atomic<T*>` и заменяет его этим значением минус предоставленное значение `i` с применением стандартных правил 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);
```

Оператор прединкремента std::atomic<T*>::operator++

Атомарно увеличивает на единицу значение, сохраненное 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;
```

Оператор постинкремента std::atomic<T*>::operator++

Атомарно увеличивает на единицу значение, сохраненное в `*this`, и возвращает старое значение.

Объявление

```
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
```

Результат

```
return this->fetch_add(1);
```

Оператор предекремента std::atomic <T*>::operator––

Атомарно уменьшает на единицу значение, сохраненное 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;
```

Оператор постдекремента std::atomic<T*>::operator––

Атомарно уменьшает на единицу значение, сохраненное 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);
```

Составной оператор присваивания std::atomic <T*>::operator+=

Атомарно прибавляет предоставленное значение к значению, сохраненному 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;
```

Составной оператор присваивания std::atomic<T*>::operator–=

Атомарно вычитает предоставленное значение из значения, сохраненного 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;
```

Г.4. Заголовок <future>

Заголовок `<future>` предоставляет средства 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);
}
```

Г.4.1. Шаблон класса std::future

Шаблон класса `std::future` предоставляет средства для ожидания результата асинхронной операции, выполненной в другом потоке, в сочетании с шаблонами классов `std::promise` и `std:: pack-aged_task` и шаблоном функции `std::async`, которые могут использоваться для предоставления результата выполнения асинхронной операции. На конкретный результат выполнения асинхронной операции в любой момент может ссылаться только один экземпляр `std::future`.

Экземпляры `std::future` могут быть `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` или `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);
};
```

Конструктор по умолчанию std::future

Создает объект `std::future` без связанного с ним результата выполнения асинхронной операции.

Объявление

```
future() noexcept;
```

Результат

Создает новый экземпляр `std::future`.

Постусловия

`valid()` возвращает `false`.

Выдает

Ничего не выдает.

Перемещающий конструктор std::future

Создает один объект `std::future` из другого, передавая принадлежность результата выполнения асинхронной операции, связанного с другим объектом `std::future`, в только что созданный экземпляр.

Объявление

```
future(future&& other) noexcept;
```

Результат

Создает новый экземпляр `std::future`, перемещая в него содержимое объекта `other`.

Постусловия

Результат выполнения асинхронной операции, связанный с объектом `other` до вызова конструктора, связывается с только что созданным объектом `std::future`. У объекта `other` больше нет связанного с ним результата выполнения асинхронной операции. Выражение `this->valid()` возвращает тот же результат, который возвращался выражением `other.valid()` до вызова этого конструктора. Выражение `other.valid()` возвращает `false`.

Выдает

Ничего не выдает.

Перемещающий оператор присваивания std::future

Передает принадлежность результата выполнения асинхронной операции, связанного с одним объектом `std::future`, другому объекту.

Объявление

```
future(future&& other) noexcept;
```

Результат

Передает принадлежность асинхронного состояния между экземплярами `std::future`.

Постусловия

Результат выполнения асинхронной операции, связанный с объектом `other` до вызова конструктора, связывается с `*this`, а у `other` связанного с ним результата выполнения асинхронной операции больше не остается. Принадлежность асинхронного состояния (если таковое имеется), связанного с `*this` до совершения вызова, и состояние, если это последняя ссылка, уничтожаются. Выражение `this->valid()` возвращает тот же результат, который возвращался выражением `other.valid()` до вызова этого конструктора. Выражение `other.valid()` возвращает `false`.

Выдает

Ничего не выдает.

Деструктор std::future

Уничтожает объект `std::future`.

Объявление

```
~future();
```

Результат

Уничтожает `*this`. Если это последняя ссылка на результат выполнения асинхронной операции, связанная с `*this` (если таковая имеется), то уничтожается и этот результат выполнения асинхронной операции.

Выдает

Ничего не выдает.

Компонентная функция std::future::share

Создает новый экземпляр `std::shared_future` и передает принадлежность результата выполнения асинхронной операции, связанной с `*this`, только что созданному экземпляру `std::shared_future`.

Объявление

```
shared_future<ResultType> share();
```

Результат

Как при вычислении `shared_future<ResultType>(std::move(*this))`.

Постусловия

Результат выполнения асинхронной операции, связанный с `*this` до вызова `share()` (если такой имелся), связывается с только что созданным экземпляром `std::shared_future`. Выражение `this->valid()` возвращает `false`.

Выдает

Ничего не выдает.

Компонентная функция std::future::valid

Проверяет, связан ли экземпляр `std::future` с результатом выполнения асинхронной операции.

Объявление

```
bool valid() const noexcept;
```

Возвращает

`true`, если у `*this` есть связанный с ним результат выполнения асинхронной операции, или `false` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::future::wait

Вызывает отложенную функцию, если она содержится в состоянии, связанном с `*this`, в противном случае ожидает готовности результата выполнения асинхронной операции, связанной с экземпляром `std::future`.

Объявление

```
void wait();
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если связанное состояние содержит отложенную функцию, вызывает эту функцию и сохраняет возвращенное значение или выданное исключение в качестве результата выполнения асинхронной операции. В противном случае блокируется, пока не будет готов результат выполнения асинхронной операции, связанной с `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::future::wait_for

Ожидает, пока не будет готов результат выполнения асинхронной операции, связанной с экземпляром `std::future`, или пока не истечет заданный период времени.

Объявление

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, происходит немедленный возврат управления без блокировки. В противном случае происходит блокировка до готовности результата выполнения асинхронной операции, связанной с `*this`, или до истечения периода времени, заданного аргументом `relative_time`.

Возвращает

`std::future_status::deferred`, если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, `std::future_status::ready`, если готов результат выполнения асинхронной операции, связанной с `*this`, `std::future_status::timeout`, если истек период времени, заданный аргументом `relative_time`.

ПРИМЕЧАНИЕ

Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Выдает

Ничего не выдает.

Компонентная функция std::future::wait_until

Ожидает, пока не будет готов результат выполнения асинхронной операции, связанной с экземпляром `std::future`, или пока не истечет заданный период времени.

Объявление

```
template<typename Clock,typename Duration>
future_status wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой

еще не началось, происходит немедленный возврат управления без блокировки. В противном случае происходит блокировка до готовности результата выполнения асинхронной операции, связанной с `*this`, или до тех пор, пока вызов функции `Clock::now()` не вернет время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`std::future_status::deferred`, если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, `std::future_status::ready`, если готов результат выполнения асинхронной операции, связанной с `*this`, `std::future_status::timeout`, если в результате вызова функции `Clock::now()` возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

ПРИМЕЧАНИЕ

По продолжительности блокировки вызывающего потока не дается никаких гарантий, если же функция возвращает std::future_status::timeout, то гарантируется только то, что функция Clock::now() возвращает время, равное времени absolute_time на тот момент, когда поток становится разблокированным, или более позднее.

Выдает

Ничего не выдает.

Компонентная функция std::future::get

Если связанное состояние содержит отложенную функцию из вызова `std::async`, вызывает эту функцию и возвращает результат. В противном случае ожидает готовность результата выполнения асинхронной операции, связанной с экземпляром `std::future`, а затем возвращает сохраненное значение или выдает сохраненное исключение.

Объявление

```
void future<void>::get();
R& future<R&>::get();
R future<R>::get();
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если состояние, связанное с `*this`, содержит отложенную функцию, происходит вызов этой функции и возвращается результат или же выполняется распространение любого выданного исключения. В противном случае происходит блокировка, пока не будет готов результат выполнения асинхронной операции, связанной с `*this`. Если результатом будет сохраненное исключение, происходит выдача этого исключения. В противном случае возвращается сохраненное значение.

Возвращает

Если связанное состояние содержит отложенную функцию, возвращается результат вызова этой функции. В противном случае, если тип `ResultType` относится к `void`, возвращается просто управление. Если типом `ResultType` является `R&` для некоторого типа `R`, возвращается сохраненная ссылка. В противном случае возвращается сохраненное значение.

Выдает

Исключение, выданное путем отложенного исключения или сохраненное в качестве результата выполнения асинхронной операции, если таковое имеется.

Постусловие

```
this->valid() == false
```

Г.4.2. Шаблон класса std::shared_future

Шаблон класса `std::shared_future` предоставляет средства ожидания результата выполнения асинхронной операции от другого потока совместно с шаблонами классов `std::promise` и `std::packaged_task` и шаблоном функции `std::async`, который может применяться для предоставления этого результата.

Экземпляры `std::shared_future` являются `CopyConstructible` и `CopyAssignable`. Объект `std::shared_future` можно также создать за счет перемещения из объекта `std::future` с тем же `ResultType`.

Обращения к конкретному экземпляру `std::shared_future` не синхронизированы. Поэтому обращения к одному и тому же экземпляру `std::shared_future` из нескольких потоков без внешней синхронизации небезопасны. Но обращения к связанному состоянию синхронизированы, поэтому безопасно обращаться без внешней синхронизации к разным экземплярам `std::shared_future`, совместно использующим связанное состояние, могут сразу несколько потоков.

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;
};
```

Конструктор по умолчанию std::shared_future

Создает объект `std::shared_future` без связанного результата выполнения асинхронной операции.

Объявление

```
shared_future() noexcept;
```

Результат

Создает новый экземпляр `std::shared_future`.

Постусловия

Для только что созданного экземпляра функция `valid()` возвращает `false`.

Выдает

Ничего не выдает.

Перемещающий конструктор std::shared_future

Создает один объект `std::shared_future` из другого, перемещая принадлежность результата выполнения асинхронной операции, связанного с другим объектом `std::shared_future`, в только что созданный экземпляр.

Объявление

```
shared_future(shared_future&& other) noexcept;
```

Результат

Создает новый экземпляр `std::shared_future`.

Постусловия

Результат выполнения асинхронной операции, связанный с объектом `other` до вызова конструктора, связывается с только что созданным объектом `std::shared_future`. У объекта `other` связанного с ним результата выполнения асинхронной операции больше нет.

Выдает

Ничего не выдает.

Конструктор перемещения в std::shared_future из std::future

Создает объект `std::shared_future` из объекта `std::future`, перемещая принадлежность связанного с `std::future` результата выполнения асинхронной операции в только что созданный объект `std::shared_future`.

Объявление

```
shared_future(std::future<ResultType>&& other) noexcept;
```

Результат

Создает новый экземпляр `std::shared_future`.

Постусловия

Результат выполнения асинхронной операции, связанный с другим объектом до вызова конструктора, связывается с только что созданным объектом `std::shared_future`. У другого объекта больше нет связанного с ним результата выполнения асинхронной операции.

Выдает

Ничего не выдает.

Копирующий конструктор std::shared_future

Создает один объект `std::shared_future` из другого таким образом, что и исходный объект и его копия ссылаются на результат выполнения асинхронной операции, связанный с исходным объектом `std::shared_future`, если таковой результат имелся.

Объявление

```
shared_future(shared_future const& other);
```

Результат

Создает новый экземпляр `std::shared_future`.

Постусловия

Результат выполнения асинхронной операции, связанный с другим объектом до вызова конструктора, связывается с только что созданным объектом `std::shared_future` и с другим объектом.

Выдает

Ничего не выдает.

Деструктор std::shared_future

Уничтожает объект `std::shared_future`.

Объявление

```
~shared_future();
```

Результат

Уничтожает `*this`. Если больше нет экземпляра `std::promise` или экземпляра `std::packaged_task`, связанного с результатом выполнения асинхронной операции, который также связан с `*this`, и это последний экземпляр `std::shared_future`, связанный с результатом выполнения асинхронной операции, то уничтожается и этот результат.

Выдает

Ничего не выдает.

Компонентная функция std::shared_future::valid

Проверяет, связан ли экземпляр `std::shared_future` с результатом выполнения асинхронной операции.

Объявление

```
bool valid() const noexcept;
```

Возвращает

`true`, если у `*this` есть связанный с ним результат выполнения асинхронной операции, или `false` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::shared_future::wait

Если состояние, связанное с `*this`, содержит отложенную функцию, вызывает эту функцию. В противном случае ожидает готовности результата выполнения асинхронной операции, связанной с экземпляром `std::shared_future`.

Объявление

```
void wait() const;
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Вызовы функций `get()` и `wait()` из нескольких потоков в отношении экземпляров `std::shared_future`, совместно использующих одно и то же связанное состояние, выполняются последовательно. Если связанное состояние содержит отложенную функцию, то первый вызов `get()` или `wait()` приводит к вызову отложенной функции и результатом вызова асинхронной операции становится сохранение возвращенного значения или выдача исключения. Ставит блокировку до готовности результата выполнения асинхронной операции, связанной с `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::shared_future::wait_for

Ожидает, пока не будет готов результат выполнения асинхронной операции, связанной с экземпляром `std::shared_future`, или пока не истечет заданный период времени.

Объявление

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time) const;
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, происходит немедленный возврат управления без блокировки. В противном случае происходит блокировка до готовности результата выполнения асинхронной операции, связанной с `*this`, или до истечения периода времени, заданного аргументом `relative_time`.

Возвращает

`std::future_status::deferred`, если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, `std::future_status::ready`, если готов результат выполнения асинхронной операции, связанной с `*this`, `std::future_status::timeout`, если истек период времени, заданный аргументом `relative_time`.

ПРИМЕЧАНИЕ

Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Выдает

Ничего не выдает.

Компонентная функция std::shared_future::wait_until

Ожидает, пока не будет готов результат выполнения асинхронной операции, связанной с экземпляром `std::shared_future`, или пока не истечет заданный период времени.

Объявление

```
template<typename Clock,typename Duration>
bool wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time) const;
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой

еще не началось, происходит немедленный возврат управления без блокировки. В противном случае происходит блокировка до готовности результата выполнения асинхронной операции, связанной с `*this`, или до тех пор, пока вызов функции `Clock::now()` не вернет время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`std::future_status::deferred`, если результат выполнения асинхронной операции, связанной с `*this`, содержит отложенную функцию, актуализируемую из вызова `std::async`, выполнение которой еще не началось, `std::future_status::ready`, если готов результат выполнения асинхронной операции, связанной с `*this`, `std::future_status::timeout`, если в результате вызова функции `Clock::now()` возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

ПРИМЕЧАНИЕ

По продолжительности блокировки вызывающего потока не дается никаких гарантий, если же функция возвращает std::future_status::timeout, то гарантируется только то, что функция Clock::now() возвращает время, равное времени absolute_time на тот момент, когда поток становится разблокированным, или более позднее.

Выдает

Ничего не выдает.

Компонентная функция std::shared_future::get

Если связанное состояние содержит отложенную функцию из вызова `std::async`, вызывает эту функцию и возвращает результат. В противном случае ожидает готовность результата выполнения асинхронной операции, связанной с экземпляром `std::future`, а затем возвращает сохраненное значение или выдает сохраненное исключение.

Объявление

```
void shared_future<void>::get() const;
R& shared_future<R&>::get() const;
R const& shared_future<R>::get() const;
```

Предусловия

Выражение `this->valid()` должно возвращать `true`.

Результат

Вызовы функций `get()` и `wait()` из нескольких потоков в отношении экземпляров `std::shared_future`, совместно использующих одно и то же связанное состояние, выполняются последовательно. Если связанное состояние содержит отложенную функцию, то первый вызов `get()` или `wait()` приводит к вызову отложенной функции, и результатом вызова асинхронной операции становится сохранение

возвращенного значения или выданного исключения. Ставит блокировку до готовности результата выполнения асинхронной операции, связанной с `*this`. Если в результате выполнения асинхронной операции хранится исключение, выдает это исключение. В противном случае возвращает сохраненное значение.

Возвращает

Если тип `ResultType` относится к `void`, возвращается просто управление. Если типом `ResultType` является `R&` для некоторого типа `R`, возвращается сохраненная ссылка. В противном случае возвращается `const`-ссылка на сохраненное значение.

Выдает

Сохраненное исключение, если таковое имеется.

Г.4.3. Шаблон класса std::packaged_task

Шаблон класса `std::packaged_task` упаковывает функцию или другой вызываемый объект, поэтому при вызове функции через экземпляр `std::packaged_task` результат сохраняется в качестве результата выполнения асинхронной операции, который можно извлечь через экземпляр `std::future`.

Экземпляры `std::packaged_task` могут быть `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` или `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();
};
```

Конструктор по умолчанию std::packaged_task

Создает объект `std::packaged_task`.

Объявление

```
packaged_task() noexcept;
```

Результат

Создает экземпляр `std::packaged_task` без связанной задачи или связанного результата выполнения асинхронной операции.

Выдает

Ничего не выдает.

Конструирование std::packaged_task из вызываемого объекта

Создает объект `std::packaged_task` со связанной задачей и связанным результатом выполнения асинхронной операции.

Объявление

```
template<typename Callable>
packaged_task(Callable&& func);
```

Предусловия

Выражение `func(args...)`, в котором каждый элемент `args-`*i* в списке `args...` должен быть значением соответствующего типа `ArgTypes-`*i* из списка `ArgTypes...`, должно быть допустимым. Возвращаемое значение должно допускать преобразование в `ResultType`.

Результат

Создает экземпляр `std::packaged_task` со связанным, но еще не готовым результатом выполнения асинхронной операции типа `ResultType` и со связанной задачей типа `Callable`, являющейся копией `func`.

Выдает

Исключение типа `std::bad_alloc`, если конструктор не в состоянии распределить память для результата выполнения асинхронной операции. Любое исключение, выданное в результате вызова конструктора копирования или конструктора перемещения объекта `Callable`.

Конструирование std::packaged_task из вызываемого объекта с распределителем

Создает объект `std::packaged_task` со связанной задачей и связанным результатом выполнения асинхронной операции, используя предоставленный распределитель памяти под результат выполнения асинхронной операции и задачу.

Объявление

```
template<typename Allocator,typename Callable>
packaged_task(
    std::allocator_arg_t, Allocator const& alloc,Callable&& func);
```

Предусловия

Выражение `func(args...)`, в котором каждый элемент `args-`*i* в списке `args...` должен быть значением соответствующего типа `ArgTypes-`*i* из списка `ArgTypes...`, должно быть допустимым. Возвращаемое значение должно допускать преобразование в `ResultType`.

Результат

Создает экземпляр `std::packaged_task` со связанным, но еще не готовым результатом выполнения асинхронной операции типа `ResultType` и связанной задачей типа `Callable`, являющейся копией `func`. Память для результата выполнения асинхронной операции и операции распределяется посредством распределителя `alloc` или через соответствующую копию.

Выдает

Любое исключение, выданное распределителем при попытке распределения памяти под результат выполнения асинхронной операции или под задачу. Любое исключение, выданное в результате вызова конструктора копирования или конструктора перемещения объекта `Callable`.

Перемещающий конструктор std::packaged_task

Создает один объект `std::packaged_task` из другого, перемещая принадлежность связанных с другим объектом `std::packaged_task` результата выполнения асинхронной операции и задачи в только что созданный экземпляр.

Объявление

```
packaged_task(packaged_task&& other) noexcept;
```

Результат

Создает новый экземпляр `std::packaged_task`.

Постусловия

Связанные до вызова конструктора с `other` результат выполнения асинхронной операции и задача становятся связанными с только что созданным объектом `std::packaged_task`. У объекта `other` связанного с ним результата выполнения асинхронной операции больше нет.

Выдает

Ничего не выдает.

Перемещающий оператор присваивания std::packaged_task

Перемещает принадлежность результата выполнения асинхронной операции, связанного с одним объектом `std::packaged_task` в другой объект.

Объявление

```
packaged_task& operator=(packaged_task&& other) noexcept;
```

Результат

Перемещает принадлежность результата выполнения асинхронной операции и задачи, связанной с `other`, в `*this` и сбрасывает любой прежний результат выполнения асинхронной операции, как при вычислении `std::packaged_task(other).swap(*this)`.

Постусловия

Связанные до вызова перемещающего оператора присваивания с `other` результат выполнения асинхронной операции и задача становятся связанными с `*this`. У объекта `other` связанного с ним результата выполнения асинхронной операции больше нет.

Возвращает

`*this`

Выдает

Ничего не выдает.

Компонентная функция std::packaged_task::swap

Производит обмен принадлежности результатов выполнения асинхронной операции между двумя объектами `std::packaged_task`.

Объявление

```
void swap(packaged_task& other) noexcept;
```

Результат

Производит обмен принадлежности результатов выполнения асинхронной операции и задач, связанных с `other` и `*this`.

Постусловия

Результат выполнения асинхронной операции и задача, связанные с `other` до вызова `swap` (если таковые имелись), связаны с `*this`. Результат выполнения асинхронной операции и задача, связанные с `*this` до вызова `swap` (если таковые имелись), связаны с `other`.

Выдает

Ничего не выдает.

Деструктор std::packaged_task

Уничтожает объект `std::packaged_task`.

Объявление

```
~packaged_task();
```

Результат

Уничтожает `*this`. Если у `*this` есть связанный с ним результат выполнения асинхронной операции и этот результат не содержит сохраненной задачи или

исключения, он становится готовым с исключением `std::future_error`, содержащим код ошибки `std::future_errc::broken_promise`.

Выдает

Ничего не выдает.

Компонентная функция std::packaged_task::get_future

Извлекает экземпляр `std::future` для результата выполнения асинхронной операции, связанного с `*this`.

Объявление

```
std::future<ResultType> get_future();
```

Предусловия

У `*this` есть связанный с ним результат выполнения асинхронной операции.

Возвращает

Экземпляр `std::future` для результата выполнения асинхронной операции, связанного с `*this`.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`, если объект `std::future` для этого результата выполнения асинхронной операции уже был получен в результате предыдущего вызова `get_future()`.

Компонентная функция std::packaged_task::reset

Связывает экземпляр `std::packaged_task` с новым результатом выполнения асинхронной операции для той же задачи.

Объявление

```
void reset();
```

Предусловия

У `*this` есть связанная с ним асинхронно выполняемая задача.

Результат

Как при вычислении `*this=packaged_task(std::move(f))`, где `f` — сохраненная задача, связанная с `*this`.

Выдает

Исключение типа `std::bad_alloc`, если под новый результат выполнения асинхронной операции не удается распределить память.

Компонентная функция std::packaged_task::valid

Проверяет у `*this` наличие связанной с ним задачи или результата выполнения асинхронной операции.

Объявление

```
bool valid() const noexcept;
```

Возвращает

`true`, если у `*this` есть связанные с ним задача и результат выполнения асинхронной операции, или `false` в противном случае.

Выдает

Ничего не выдает.

Оператор вызова функции std::packaged_task::operator()

Вызывает задачу, связанную с экземпляром `std::packaged_task`, и сохраняет возвращенное значение или исключение в связанном результате выполнения асинхронной операции.

Объявление

```
void operator()(ArgTypes... args);
```

Предусловия

У `*this` имеется связанная с ним задача.

Результат

Вызывает связанную задачу `func`, как при вычислении `INVOKE(func,args...)`. Если возвращение происходит обычным порядком, сохраняет возвращаемое значение в результате выполнения асинхронной операции, связанном с `*this`. Если при вызове возвращается исключение, сохраняет его в результате выполнения асинхронной операции, связанном с `*this`.

Постусловия

Результат выполнения асинхронной операции, связанной с `*this`, готов и имеет сохраненное значение или исключение. Все потоки, заблокированные в ожидании результата выполнения асинхронной операции, разблокируются.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в результате выполнения асинхронной операции уже есть сохраненное значение или исключение.

Синхронизация

Успешный вызов оператора вызова функции «синхронизируется с» вызовом `std::future<ResultType>::get()` или `std::shared_future<ResultType>::get()`, в результате которого извлекается сохраненное значение или исключение.

Компонентная функция std::packaged_task::make_ready_at_thread_exit

Вызывает задачу, связанную с экземпляром `std::packaged_task`, и сохраняет возвращенное значение или исключение в связанном результате выполнения асинхронной операции, не делая этот результат готовым до выхода из потока.

Объявление

```
void make_ready_at_thread_exit(ArgTypes... args);
```

Предусловия

У `*this` имеется связанная с ним задача.

Результат

Вызывает связанную задачу `func`, как при вычислении `INVOKE(func,args...)`. Если возвращение происходит обычным порядком, сохраняет возвращаемое значение в результате выполнения асинхронной операции, связанном с `*this`. Если при вызове возвращается исключение, сохраняет его в результате выполнения асинхронной операции, связанном с `*this`. Планирует придание готовности связанному асинхронному состоянию при выходе из текущего потока.

Постусловия

В результате выполнения асинхронной операции, связанном с `*this`, имеется сохраненное значение или исключение, но готовность ему не придается, пока не начнется выход из текущего потока. Потоки, заблокированные в ожидании результата выполнения асинхронной операции, будут разблокированы при выходе из текущего потока.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если результат выполнения асинхронной операции уже имеет сохраненное значение или исключение. Исключение типа `std::future_error` с кодом ошибки `std::future_errc::no_state`, если у `*this` не имеется связанного с ним асинхронного состояния.

Синхронизация

Завершение потока, выполнившего успешный вызов функции `make_ready_at_thread_exit()`, синхронизируется с вызовом функции `std::future<ResultType>::get()` или функции `std::shared_future<ResultType>::get()`, которая извлекает сохраненное значение или исключение.

Г.4.4. Шаблон класса std::promise

Шаблон класса `std::promise` предоставляет средства для установки результата выполнения асинхронной операции, который можно получить из другого потока через экземпляр `std::future`.

Параметр шаблона `ResultType` является типом значения, которое можно сохранить в асинхронном результате.

Объект `std::future`, связанный с результатом выполнения асинхронной операции конкретного экземпляра `std::promise`, можно получить путем вызова компонентной функции `get_future()`. Результатом выполнения асинхронной операции становится либо значение типа `ResultType` при вызове компонентной функции `set_value()`, либо исключение при вызове компонентной функции `set_exception()`.

Экземпляры `std::promise` могут быть `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(см. описание);
    void set_exception(std::exception_ptr p);
};
```

Конструктор по умолчанию std::promise

Создает объект `std::promise`.

Объявление

```
promise();
```

Результат

Создает экземпляр `std::promise` с еще неготовым связанным результатом выполнения асинхронной операции типа `ResultType`.

Выдает

Исключение типа `std::bad_alloc`, если конструктор не в состоянии распределить память под результат выполнения асинхронной операции.

Конструктор-распределитель std::promise

Создает объект `std::promise`, используя предоставленный распределитель для выделения памяти под связанный результат выполнения асинхронной операции.

Объявление

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

Результат

Создает экземпляр `std::promise` с еще неготовым связанным результатом выполнения асинхронной операции типа `ResultType`. Память под результат выполнения асинхронной операции выделяется посредством распределителя `alloc`.

Выдает

Любое исключение, выданное распределителем при попытке выделения памяти под результат выполнения асинхронной операции.

Перемещающий конструктор std::promise

Создает один объект `std::promise` из другого, перемещая принадлежность результата выполнения асинхронной операциии, связанного с другим объектом `std::promise`, в только что созданный экземпляр.

Объявление

```
promise(promise&& other) noexcept;
```

Результат

Создает новый экземпляр `std::promise`.

Постусловия

Результат выполнения асинхронной операции, связанный с объектом `other` до вызова конструктора, связывается с только что созданным объектом `std::promise`. У объекта `other` больше нет связанного с ним результата выполнения асинхронной операции.

Выдает

Ничего не выдает.

Перемещающий оператор присваивания std::promise

Перемещает принадлежность результата выполнения асинхронной операции, связанного с одним объектом `std::promise`, в другой объект.

Объявление

```
promise& operator=(promise&& other) noexcept;
```

Результат

Перемещает принадлежность результата выполнения асинхронной операции, связанного с `other`, в `*this`. Если у `*this` уже есть связанный с ним результат выполнения асинхронной операции, этот результат делается готовым с исключением типа `std::future_error` и кодом ошибки `std::future_errc::broken_promise`.

Постусловия

Результат выполнения асинхронной операции, связанный с объектом `other` до вызова перемещающего оператора присваивания, связывается `*this`. У объекта `other` больше нет связанного с ним результата выполнения асинхронной операции.

Возвращает

`*this`

Выдает

Ничего не выдает.

Компонентная функция std::promise::swap

Производит обмен принадлежности результатов выполнения асинхронной операции между двумя объектами `std::promise`.

Объявление

```
void swap(promise& other);
```

Результат

Производит обмен принадлежности результатов выполнения асинхронной операции и задач, связанных с `other` и `*this`.

Постусловия

Результат выполнения асинхронной операции, связанный с `other` до вызова `swap` (если таковой имелся), связан с `*this`. Результат выполнения асинхронной операции, связанный с `*this` до вызова `swap` (если таковой имелся), связан с `other`.

Выдает

Ничего не выдает.

Деструктор std::promise

Уничтожает объект `std::promise`.

Объявление

```
~promise();
```

Результат

Уничтожает `*this`. Если у `*this` есть связанный с ним результат выполнения асинхронной операции и этот результат не содержит сохраненного значения или исключения, он становится готовым с исключением `std::future_error`, содержащим код ошибки `std::future_errc::broken_promise`.

Выдает

Ничего не выдает.

Компонентная функция std::promise::get_future

Извлекает экземпляр `std::future` для результата выполнения асинхронной операции, связанного с `*this`.

Объявление

```
std::future<ResultType> get_future();
```

Предусловия

У `*this` имеется связанный с ним результат выполнения асинхронной операции.

Возвращает

Экземпляр `std::future` для результата выполнения асинхронной операции, связанного с `*this`.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`, если объект `std::future` для этого результата выполнения асинхронной операции уже был получен в результате предыдущего вызова `get_future()`.

Компонентная функция std::promise::set_value

Сохраняет значение результата выполнения асинхронной операции, связанного 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

У `*this` имеется связанный с ним результат выполнения асинхронной операции.

Результат

Сохраняет `r` в результате выполнения асинхронной операции, связанном с `*this`, если `ResultType` не относится к типу `void`.

Постусловия

Связанный с `*this` результат выполнения асинхронной операции готов и имеет сохраненное значение. Все потоки, заблокированные в ожидании результата выполнения асинхронной операции, разблокируются.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если у результата выполнения асинхронной операции уже есть сохраненное значение или исключение. Любые исключения, выданные копирующим или перемещающим конструктором `r`.

Синхронизация

Несколько конкурентных вызовов `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` выполняются последовательно. Успешный вызов `set_value()` «происходит до» вызова функции `std::future<ResultType>::get()` или вызова функции `std::shared_future<ResultType>::get()`, извлекающей сохраненное значение.

Компонентная функция std::promise::set_value_at_thread_exit

Сохраняет значение в результате выполнения асинхронной операции, связанном с `*this`, не делая этот результат готовым до выхода из текущего 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

У `*this` есть связанный с ним результат выполнения асинхронной операции.

Результат

Сохраняет `r` в результате выполнения асинхронной операции, связанном с `*this`, если `ResultType` не относится к типу `void`. Помечает результат выполнения асинхронной операции как имеющий сохраненное значение. Планирует придание готовности связанному результату выполнения асинхронной операции при выходе из текущего потока.

Постусловия

Результат выполнения асинхронной операции, связанный с `*this`, имеет сохраненное значение, но не готов, пока не будет выполнен выход из текущего потока. Потоки, заблокированные в ожидании результата выполнения асинхронной операции, будут разблокированы при выходе из текущего потока.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если результат выполнения асинхронной операции уже имеет сохраненное значение или исключение. Любые исключения, выданные копирующим или перемещающим конструктором `r`.

Синхронизация

Несколько конкурентных вызовов `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` выполняются последовательно. Завершение потока, выполнившего успешный вызов функции `set_value_at_thread_exit()`, «происходит до» вызова функции `std::future<ResultType>::get()` или вызова функции `std::shared_future<ResultType>::get()`, извлекающей сохраненное значение.

Компонентная функция std::promise::set_exception

Сохраняет исключение в результате выполнения асинхронной операции, связанном с `*this`.

Объявление

```
void set_exception(std::exception_ptr e);
```

Предусловия

У `*this` имеется связанный с ним результат выполнения асинхронной операции. Выражение `(bool)e` вычисляется в `true`.

Результат

Сохраняет `e` в результате выполнения асинхронной операции, связанном с `*this`.

Постусловия

Результат выполнения асинхронной операции, связанный с `*this`, готов и имеет сохраненное исключение. Все потоки, заблокированные в ожидании результата выполнения асинхронной операции, разблокируются.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в результате выполнения асинхронной операции уже имеется сохраненное значение или исключение.

Синхронизация

Несколько конкурентных вызовов `set_value()` и `set_exception()` выполняются последовательно. Успешный вызов функции `set_exception()` «происходит до» вызова функции `std::future<Result-Type>::get()` или функции `std::shared_future<ResultType>::get()`, извлекающей сохраненное исключение.

Компонентная функция std::promise::set_exception_at_thread_exit

Сохраняет исключение в результате выполнения асинхронной операции, связанном с `*this`, не делая этот результат готовым до выхода из текущего потока.

Объявление

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

Предусловия

У `*this` имеется связанный с ним результат выполнения асинхронной операции. Выражение `(bool)e` вычисляется в `true`.

Результат

Сохраняет `e` в результате выполнения асинхронной операции, связанном с `*this`. Планирует придание готовности связанному результату выполнения асинхронной операции при выходе из текущего потока.

Постусловия

Результат выполнения асинхронной операции, связанной с `*this`, имеет сохраненное исключение, но не готов, пока не будет выполнен выход из текущего потока. Потоки, заблокированные в ожидании результата выполнения асинхронной операции, будут разблокированы при выходе из текущего потока.

Выдает

Исключение типа `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`, если в результате выполнения асинхронной операции уже имеется сохраненное значение или исключение.

Синхронизация

Несколько конкурентных вызовов `set_value()`, `set_value_at_thread_exit()`, `set_exception()` и `set_exception_at_thread_exit()` выполняются последовательно. Завершение потока, выполнившего успешный вызов функции `set_exception_at_thread_exit()`, «происходит до» вызова функции `std::future<ResultType>::get()` или вызова функции `std::shared_future<ResultType>::get()`, извлекающей сохраненное исключение.

Г.4.5. Шаблон функции std::async

Шаблон функции `std::async` предоставляет простой способ запуска автономных асинхронных задач для задействования доступной аппаратной конкурентности. Вызов функции `std::async` возвращает объект `std::future`, который будет содержать результат выполнения задачи. В зависимости от политики запуска задача либо запускается в асинхронном режиме в своем собственном потоке, либо в синхронном режиме в том потоке, который вызвал в отношении данного фьючерса компонентную функцию `wait()` или `get()`.

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)` является допустимым для предоставленных значений `func` и `args`. Объекты типа `Callable` и все компоненты `Args` являются `MoveConstructible`.

Результат

Создает копии `func` и `args...` во внутреннем хранилище (обозначаемом соответственно `fff` и `xyz...`).

Если `policy` имеет значение `std::launch::async`, запускает `INVOKE(fff,xyz...)` в своем собственном потоке. Возвращенный объект `std::future` станет готовым, как только этот поток завершится, и будет содержать либо возвращаемое значение, либо исключение, выданное в результате вызова функции. Деструктор последнего объекта-фьючерса, связанного с асинхронным состоянием возвращенного `std::future`, блокируется до готовности фьючерса.

Если `policy` имеет значение `std::launch::deferred`, `fff` и `xyz...` сохраняются в возвращенном объекте `std::future` в виде вызова отложенной функции. Первый вызов в отношении фьючерса, совместно использующего то же самое связанное состояние

компонентных функций `wait()` или `get()`, приведет к выполнению в синхронном режиме функции `INVOKE(fff,xyz...)` в потоке, который вызвал `wait()` или `get()`.

Из вызова функции `get()` в отношении данного объекта `std::future` будет возвращено либо значение, полученное в результате выполнения функции `INVOKE(fff, xyz...)`, либо исключение, выданное этой функцией.

Если `policy` имеет значение `std::launch::async|std::launch::deferred` или же аргумент `policy` опущен, поведение будет таким же, как и при задании политики `std::launch::async` или политики `std::launch::deferred`. Реализация будет выбирать поведение исходя из конкретной обстановки вызова, чтобы задействовать доступную аппаратную конкурентность без превышения лимитов.

Во всех случаях при вызове `std::async` происходит немедленное возвращение управления.

Синхронизация

Завершение вызова функции «происходит до» успешного возвращения из вызова `wait()`, `get()`, `wait_for()` или `wait_until()` в отношении любого экземпляра `std::future` или `std::shared_future`, ссылающегося на то же связанное состояние, что и объект `std:: future`, возвращенный из вызова `std::async`. Если `policy` имеет значение `std::launch::async`, завершение потока, в котором произошел вызов функции, также «происходит до» успешного возвращения из этих вызовов.

Выдает

Исключение `std::bad_alloc`, если память под требуемое внутреннее хранилище не может быть выделена, в противном случае выдает исключение `std::future_error`, когда результат не может быть достигнут, или же выдано любое исключение в ходе создания `fff` или `xyz...`.

Г.5. Заголовок <mutex>

Заголовок `<mutex>` предоставляет средства обеспечения взаимного исключения: типы мьютексов, типы блокировок и функции, а также механизм, гарантирующий сугубо однократное выполнение операции.

Содержимое заголовка

```
namespace std
{
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;
    class shared_mutex;
    class shared_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 LockableType>
    class shared_lock;

    template<typename ... LockableTypes>
    class scoped_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...);
}
```

Г.5.1. Класс std::mutex

Класс `std::mutex` предоставляет базовое взаимное исключение и средства синхронизации для потоков, которые могут применяться для защиты совместно используемых данных. До обращения к данным, защищенным мьютексом, этот мьютекс должен быть заблокирован путем вызова функции `lock()` или функции `try_lock()`. Одновременно удерживать мьютекс может только один поток, поэтому если другой поток также пытается заблокировать мьютекс, то он, соответственно, потерпит неудачу (при вызове `try_lock()`) или заблокируется (при вызове `lock()`). Как только поток завершит обращение к совместно используемым данным, он должен вызвать функцию `unlock()` для снятия блокировки и разрешения другим потокам завладеть ею.

Экземпляр `std::mutex` является `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();
};
```

Конструктор по умолчанию std::mutex

Создает объект `std::mutex`.

Объявление

```
constexpr mutex() noexcept;
```

Результат

Создает экземпляр `std::mutex`.

Постусловия

Изначально только что созданный объект `std::mutex` находится в разблокированном состоянии.

Выдает

Ничего не выдает.

Деструктор std::mutex

Уничтожает объект `std::mutex`.

Объявление

```
~mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::mutex::lock

Получает блокировку объекта `std::mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку объекта `*this`.

Результат

Блокирует текущий поток, пока не будет получена блокировка объекта `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::mutex::try_lock

Пытается получить блокировку объекта `std::mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку объекта `*this`.

Результат

Пытается получить блокировку `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::mutex::unlock

Освобождает блокировку объекта `std::mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает блокировку `*this`, удерживаемую текущим потоком. Если имеются какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один из них.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Г.5.2. Класс std::recursive_mutex

Класс `std::recursive_mutex` предоставляет базовое взаимное исключение и средства синхронизации для потоков, которые можно применять для защиты совместно используемых данных. До обращения к данным, защищенным мьютексом, этот мьютекс должен быть заблокирован путем вызова функции `lock()` или функции `try_lock()`. Одновременно удерживать мьютекс может только один поток, поэтому если другой поток также пытается заблокировать `recursive_mutex` мьютекс, то он, соответственно, потерпит неудачу (при вызове `try_lock()`) или заблокируется (при вызове `lock()`). Как только поток завершит обращение к совместно используемым данным, он должен вызвать функцию `unlock()` для снятия блокировки и разрешения другим потокам завладеть ею.

Это рекурсивный мьютекс, поэтому поток, удерживающий блокировку конкретного экземпляра `std::recursive_mutex`, может и дальше вызывать `lock()` или `try_lock()`, чтобы увеличивать количество блокировок. Мьютекс не может быть заблокирован другим потоком, пока поток, получивший блокировку, не вызвал функцию `unlock` столько раз, сколько было успешных вызовов функций `lock()` или `try_lock()`.

Экземпляр `std::recursive_mutex` является `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;
};
```

Конструктор по умолчанию std::recursive_mutex

Создает объект `std::recursive_mutex`.

Объявление

```
recursive_mutex() noexcept;
```

Результат

Создает экземпляр `std::recursive_mutex`.

Постусловия

Изначально только что созданный объект `std::recursive_mutex` находится в разблокированном состоянии.

Выдает

Исключение типа `std::system_error`, если не может создать новый экземпляр `std::recursive_mutex`.

Деструктор std::recursive_mutex

Уничтожает объект `std::recursive_mutex`.

Объявление

```
~recursive_mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::recursive_mutex::lock

Получает блокировку объекта `std::recursive_mutex` для текущего потока.

Объявление

```
void lock();
```

Результат

Блокирует текущий поток, пока не будет получена блокировка `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком. Если вызывающий поток уже удерживает блокировку `*this`, значение счетчика блокировок увеличивается на единицу.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::recursive_mutex::try_lock

Пытается получить блокировку объекта `std::recursive_mutex` для текущего потока.

Объявление

```
bool try_lock() noexcept;
```

Результат

Пытается получить блокировку `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Получена новая блокировка `*this` для вызывающего потока, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если вызывающий поток уже удерживает блокировку *this, функция возвращает true, а значение счетчика блокировок *this, удерживаемых вызывающим потоком, увеличивается на единицу. Если текущий поток еще не удерживает блокировку *this, функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::recursive_mutex::unlock

Освобождает блокировку объекта `std::recursive_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает блокировку `*this`, удерживаемую текущим потоком. Если это последняя блокировка `*this`, удерживаемая вызывающим потоком, и есть какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один из них.

Постусловия

Количество блокировок `*this`, удерживаемых вызывающим потоком, уменьшается на единицу.

Выдает

Ничего не выдает.

Г.5.3. Класс std::timed_mutex

Класс `std::timed_mutex` предоставляет поддержку блокировок с тайм-аутами в виде надстроек над базовыми взаимными исключениями и средствами синхронизации, предоставляемыми `std::mutex`. До обращения к данным, защищенным мьютексом, этот мьютекс должен быть заблокирован путем вызова одной из функций `lock()`,

`try_lock()`, `try_lock_for()` или `try_lock_until()`. Если блокировка уже удерживается другим потоком, попытка получения блокировки потерпит неудачу (при вызове `try_lock()`), произойдет блокировка потока до появления возможности получения блокировки (при вызове `lock()`) или поток будет заблокирован, пока не появится возможность получить блокировку или не выйдет время на попытку ее получения (при вызове `try_lock_for()` или `try_lock_until()`). Как только блокировка будет получена (при этом неважно, какая именно функция использовалась для ее получения), она должна быть освобождена путем вызова функции `unlock()`, прежде чем мьютекс сможет быть заблокирован другим потоком.

Экземпляр `std::timed_mutex` является `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);
};
```

Конструктор по умолчанию std::timed_mutex

Создает объект `std::timed_mutex`.

Объявление

```
timed_mutex();
```

Результат

Создает экземпляр `std::timed_mutex`.

Постусловия

Изначально только что созданный объект `std::timed_mutex` находится в разблокированном состоянии.

Выдает

Исключение типа `std::system_error`, если не может создать новый экземпляр `std::timed_mutex`.

Деструктор std::timed_mutex

Уничтожает объект `std::timed_mutex`.

Объявление

```
~timed_mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::timed_mutex::lock

Получает блокировку объекта `std::timed_mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток, пока не будет получена блокировка объекта `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::timed_mutex::try_lock

Пытается получить блокировку объекта `std::timed_mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Пытается получить блокировку `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::timed_mutex::try_lock_for

Пытается получить блокировку объекта `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 блокировку `*this`.

Результат

Пытается получить блокировку `*this` для вызывающего потока за время, заданное аргументом `relative_time`. Если функция `relative_time.count()` возвращает ноль или отрицательное значение, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока не будет получена блокировка или не истечет время, заданное аргументом `relative_time`.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Компонентная функция std::timed_mutex::try_lock_until

Пытается получить блокировку объекта `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 блокировку `*this`.

Результат

Пытается получить блокировку `*this` для вызывающего потока до времени, заданного аргументом `absolute_time`. Если при входе в функцию `absolute_time<=Clock::now()`, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока либо не будет получена блокировка, либо в результате вызова функции `Clock::now()` не будет возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное absolute_time в тот момент, когда поток стал разблокированным, или более позднее.

Компонентная функция std::timed_mutex::unlock

Освобождает блокировку объекта `std::timed_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает блокировку объекта `*this`, удерживаемую текущим потоком. Если имеются какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один из них.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Г.5.4. Класс std::recursive_timed_mutex

Класс `std::recursive_timed_mutex` предоставляет поддержку блокировок с таймаутами в виде надстроек над взаимным исключением и средством синхронизации, предоставляемыми `std::recursive_mutex`. До обращения к данным, защищенным мьютексом, этот мьютекс должен быть заблокирован путем вызова одной из функций `lock()`, `try_lock()`, `try_lock_for()` или `try_lock_until()`. Если блокировка уже удерживается другим потоком, попытка получения блокировки потерпит неудачу (при вызове `try_lock()`), произойдет блокировка потока до появления возможности получения блокировки (при вызове `lock()`) или поток будет заблокирован, пока не появится возможность получить блокировку или не выйдет время на попытку ее получения (при вызове `try_lock_for()` или `try_lock_until()`). Как только блокировка будет получена (при этом неважно, какая именно функция использовалась для ее получения), она должна быть освобождена путем вызова функции `unlock()`, прежде чем мьютекс сможет быть заблокирован другим потоком.

Это рекурсивный мьютекс, поэтому поток, удерживающий блокировку конкретного экземпляра `std::recursive_timed_mutex`, может получать дополнительные блокировки этого экземпляра посредством любой функции получения блокировки. Прежде чем другой поток сможет получить блокировку данного экземпляра, все эти блокировки должны быть освобождены соответствующим вызовом функции `unlock()`.

Экземпляр `std::recursive_timed_mutex` должен быть `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);
};
```

Конструктор по умолчанию std::recursive_timed_mutex

Создает объект `std::recursive_timed_mutex`.

Объявление

```
recursive_timed_mutex();
```

Результат

Создает экземпляр `std::recursive_timed_mutex`.

Постусловия

Изначально только что созданный объект `std::recursive_timed_mutex` находится в разблокированном состоянии.

Выдает

Исключение типа `std::system_error`, если не может создать новый экземпляр `std::recursive_timed_mutex`.

Деструктор std::recursive_timed_mutex

Уничтожает объект `std::recursive_timed_mutex`.

Объявление

```
~recursive_timed_mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::recursive_timed_mutex::lock

Получает блокировку объекта `std::recursive_timed_mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток, пока не будет получена блокировка объекта `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком. Если вызывающий поток уже удерживает блокировку `*this`, значение счетчика блокировок увеличивается на единицу.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::recursive_timed_mutex::try_lock

Пытается получить блокировку объекта `std::recursive_timed_mutex` для текущего потока.

Объявление

```
bool try_lock() noexcept;
```

Результат

Пытается получить блокировку `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если вызывающий поток уже удерживает блокировку *this, функция возвращает true, а значение счетчика блокировок *this, удерживаемых вызывающим потоком, увеличивается на единицу. Если текущий поток еще не удерживает блокировку *this, функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::recursive_timed_mutex::try_lock_for

Пытается получить блокировку объекта `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 получить блокировку `*this` для вызывающего потока за время, заданное аргументом `relative_time`. Если функция `relative_time.count()` возвращает ноль или отрицательное значение, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока не будет получена блокировка или не истечет время, заданное аргументом `relative_time`.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если вызывающий поток уже удерживает блокировку *this, функция возвращает true, а значение счетчика блокировок *this, удерживаемых вызывающим потоком, увеличивается на единицу. Если текущий поток еще не удерживает блокировку *this, функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Компонентная функция std::recursive_timed_mutex::try_lock_until

Пытается получить блокировку объекта `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 получить блокировку `*this` для вызывающего потока до времени, заданного аргументом `absolute_time`. Если при входе в функцию `absolute_time<=Clock::now()`, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока либо не будет получена блокировка, либо в результате вызова функции `Clock::now()`

не будет возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если вызывающий поток уже удерживает блокировку *this, функция возвращает true, а значение счетчика блокировок *this, удерживаемых вызывающим потоком, увеличивается на единицу. Если текущий поток еще не удерживает блокировку *this, функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное absolute_time в тот момент, когда поток стал разблокированным, или более позднее.

Компонентная функция std::recursive_timed_mutex::unlock

Освобождает блокировку объекта `std::recursive_timed_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает блокировку объекта `*this`, удерживаемую текущим потоком. Если это последняя блокировка `*this`, удерживаемая вызывающим потоком, и есть какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один из них.

Постусловия

Количество блокировок `*this`, удерживаемых вызывающим потоком, уменьшается на единицу.

Выдает

Ничего не выдает.

Г.5.5. Класс std::shared_mutex

Класс `std::shared_mutex` предоставляет взаимное исключение и средства синхронизации для потоков, которые можно применять для защиты совместно используемых данных, которые часто считываются и редко изменяются. Он позволяет одному потоку удерживать исключительную блокировку или одному или нескольким потокам удерживать совместно используемую блокировку. До изменения данных, защищенных мьютексом, этот мьютекс должен получить исключительную блокировку путем вызова функции `lock()` или функции `try_lock()`. Одновременно исключительная блокировка может удерживаться только одним потоком, поэтому если другой поток также попытается заблокировать мьютекс, то он, соответственно, потерпит неудачу (при вызове `try_lock()`) или заблокируется (при вызове `lock()`). Как только поток завершит внесение изменений в совместно используемые данные, он должен вызвать функцию `unlock()` для снятия блокировки и разрешения другим потокам завладеть ею. Потоки, которым нужно лишь прочитать защищаемые данные, могут получить совместно используемую блокировку путем вызова функции `lock_shared()` или функции `try_lock_shared()`. Одновременно удерживать совместно используемую блокировку могут сразу несколько потоков, поэтому, если один поток удерживает такую блокировку, то другой поток может получить такую же блокировку. Если поток пытается получить исключительную блокировку, ему придется подождать. Как только поток, получивший совместно используемую блокировку, завершит обращение к защищаемым данным, он должен вызвать функцию `unlock_shared()`, чтобы освободить совместно используемую блокировку.

Экземпляр `std::shared_mutex` является `Lockable`.

Определение класса

```
class shared_mutex
{
public:
    shared_mutex(shared_mutex const&)=delete;
    shared_mutex& operator=(shared_mutex const&)=delete;

    shared_mutex() noexcept;
    ~shared_mutex();

    void lock();
    void unlock();
    bool try_lock();

    void lock_shared();
    void unlock_shared();
    bool try_lock_shared();
};
```

Конструктор по умолчанию std::shared_mutex

Создает объект `std::shared_mutex`.

Объявление

```
shared_mutex() noexcept;
```

Результат

Создает экземпляр `std::shared_mutex`.

Постусловия

Изначально только что созданный объект `std::shared_mutex` находится в разблокированном состоянии.

Выдает

Ничего не выдает.

Деструктор std::shared_mutex

Уничтожает объект `std::shared_mutex`.

Объявление

```
~shared_mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::shared_mutex::lock

Получает исключительную блокировку объекта `std::shared_mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток до появления возможности получения исключительной блокировки объекта `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением исключительной блокировки.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::shared_mutex::try_lock

Пытается получить исключительную блокировку объекта `std::shared_mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Пытается получить исключительную блокировку `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением исключительной блокировки, если функция возвратила `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::shared_mutex::unlock

Освобождает исключительную блокировку объекта `std::shared_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает исключительную блокировку объекта `*this`, удерживаемую текущим потоком. Если какие-либо потоки заблокированы в ожидании получения блокировки `*this`, разблокирует один поток, ожидающий получения исключительно блокировки или несколько потоков, ожидающих получения совместно используемой блокировки.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Компонентная функция std::shared_mutex::lock_shared

Получает совместно используемую блокировку объекта `std::shared_mutex` для текущего потока.

Объявление

```
void lock_shared();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток, пока не будет получена совместно используемая блокировка `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением совместно используемой блокировки.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::shared_mutex::try_lock_shared

Пытается получить совместно используемую блокировку объекта `std::shared_mutex` для текущего потока.

Объявление

```
bool try_lock_shared();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Пытается получить совместно используемую блокировку объекта `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением совместно используемой блокировки, если функция возвращает `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::shared_mutex::unlock_shared

Освобождает совместно используемую блокировку объекта `std::shared_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock_shared();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает совместно используемую блокировку объекта `*this`, удерживаемую текущим потоком. Если это последняя совместно используемая блокировка `*this` и есть какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один поток, ожидающий эксклюзивной блокировки, или несколько потоков, ожидающих совместно используемой блокировки.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Г.5.6. Класс std::shared_timed_mutex

Класс `std::shared_mutex` предоставляет поддержку блокировок с тайм-аутами в виде надстроек над взаимным исключением и средством синхронизации для потоков, которые можно применять для защиты совместно используемых данных, которые часто считываются и редко изменяются. Он позволяет удерживать одному потоку исключительную блокировку или одному или нескольким потокам — совместно используемую блокировку. До изменения данных, защищенных мьютексом, этот мьютекс должен получить исключительную блокировку путем вызова функции `lock()` или функции `try_lock()`. Одновременно исключительная блокировка может удерживаться только одним потоком, поэтому если другой поток также попытается заблокировать мьютекс, то он, соответственно, потерпит неудачу (при вызове `try_lock()`) или заблокируется (при вызове `lock()`). Как только поток завершит внесение изменений в совместно используемые данные, он должен вызвать функцию `unlock()` для снятия блокировки и разрешения другим потокам завладеть ею. Потоки, которым нужно только прочитать защищаемые данные, могут получить совместно используемую блокировку путем вызова функции `lock_shared()` или `try_lock_shared()`. Одновременно удерживать совместно используемую блокировку могут сразу несколько потоков, поэтому если один поток удерживает такую блокировку, то другой поток может получить такую же блокировку. Если поток пытается получить исключительную блокировку, ему придется подождать. Как только поток, получивший совместно используемую блокировку, завершит обращение к защищаемым данным, он должен вызвать функцию `unlock_shared()`, чтобы освободить совместно используемую блокировку.

Экземпляр `std::shared_timed_mutex` должен быть `Lockable`.

Определение класса

```
class shared_timed_mutex
{
public:
    shared_timed_mutex(shared_timed_mutex const&)=delete;
    shared_timed_mutex& operator=(shared_timed_mutex const&)=delete;

    shared_timed_mutex() noexcept;
    ~shared_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);

    void lock_shared();
    void unlock_shared();
    bool try_lock_shared();

    template<typename Rep,typename Period>
    bool try_lock_shared_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    bool try_lock_shared_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

Конструктор по умолчанию std::shared_timed_mutex

Создает объект `std::shared_timed_mutex`.

Объявление

```
shared_timed_mutex() noexcept;
```

Результат

Создает экземпляр `std::shared_timed_mutex`.

Постусловия

Изначально только что созданный объект `std::shared_timed_mutex` находится в разблокированном состоянии.

Выдает

Ничего не выдает.

Деструктор std::shared_timed_mutex

Уничтожает объект `std::shared_timed_mutex`.

Объявление

```
~shared_timed_mutex();
```

Предусловия

Объект `*this` не должен быть заблокирован.

Результат

Уничтожает `*this`.

Выдает

Ничего не выдает.

Компонентная функция std::shared_timed_mutex::lock

Получает исключительную блокировку объекта `std::shared_timed_mutex` для текущего потока.

Объявление

```
void lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток, пока появится возможность получения исключительной блокировки `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением исключительной блокировки.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::shared_timed_mutex::try_lock

Пытается получить исключительную блокировку объекта `std::shared_timed_mutex` для текущего потока.

Объявление

```
bool try_lock();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Пытается получить исключительную блокировку объекта `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением исключительной блокировки, если функция возвращает `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::shared_timed_mutex::try_lock_for

Пытается получить исключительную блокировку объекта `std::shared_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 блокировку `*this`.

Результат

Пытается получить исключительную блокировку `*this` для вызывающего потока за время, заданное аргументом `relative_time`. Если функция `relative_time.count()` возвращает ноль или отрицательное значение, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока не будет получена блокировка или не истечет время, заданное аргументом `relative_time`.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Компонентная функция std::shared_timed_mutex::try_lock_until

Пытается получить исключительную блокировку объекта `std::shared_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 блокировку `*this`.

Результат

Пытается получить исключительную блокировку `*this` для вызывающего потока до времени, заданного аргументом `absolute_time`. Если при входе в функцию `absolute_time<=Clock::now()`, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока либо не будет получена блокировка, либо в результате вызова функции `Clock::now()` не будет возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное absolute_time в тот момент, когда поток стал разблокированным, или более позднее.

Компонентная функция std::shared_timed_mutex::unlock

Освобождает исключительную блокировку объекта `std::shared_timed_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает исключительную блокировку объекта `*this`, удерживаемую текущим потоком. Если имеются какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один поток, ожидающий получения исключительной блокировки, или несколько потоков, ожидающих получения совместно используемой блокировки.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Компонентная функция std::shared_timed_mutex::lock_shared

Получает совместно используемую блокировку объекта `std::shared_timed_mutex` для текущего потока.

Объявление

```
void lock_shared();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Блокирует текущий поток, пока не появится возможность получения совместно используемой блокировки `*this`.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением совместно используемой блокировки.

Выдает

Исключение типа `std::system_error`, если произошла ошибка.

Компонентная функция std::shared_timed_mutex::try_lock_shared

Пытается получить совместно используемую блокировку объекта `std::shared_timed_mutex` для текущего потока.

Объявление

```
bool try_lock_shared();
```

Предусловия

Вызывающий поток не должен удерживать блокировку `*this`.

Результат

Пытается получить совместно используемую блокировку объекта `*this` для вызывающего потока без его блокировки.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком с применением совместно используемой блокировки, если функция возвращает `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком.

Компонентная функция std::shared_timed_mutex::try_lock_shared_for

Пытается получить совместно используемую блокировку объекта `std::shared_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 блокировку `*this`.

Результат

Пытается получить совместно используемую блокировку `*this` для вызывающего потока за время, заданное аргументом `relative_time`. Если функция `relative_time.count()` возвращает ноль или отрицательное значение, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока не будет получена блокировка или не истечет время, заданное аргументом `relative_time`.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Поток может быть заблокирован на время дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Компонентная функция std::shared_timed_mutex::try_lock_until

Пытается получить совместно используемую блокировку объекта `std::shared_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 блокировку `*this`.

Результат

Пытается получить совместно используемую блокировку `*this` для вызывающего потока до времени, заданного аргументом `absolute_time`. Если при входе в функцию `absolute_time<=Clock::now()`, вызов тут же возвращает управление, как если бы это был вызов функции `try_lock()`. В противном случае вызов блокирует поток, пока либо не будет получена блокировка, либо в результате вызова функции `Clock::now()` не будет возвращено время, равное времени, заданному аргументом `absolute_time`, или более позднее.

Возвращает

`true`, если блокировка была получена для вызывающего потока, и `false` в противном случае.

Постусловия

Объект `*this` заблокирован вызывающим потоком, если функция возвратила значение `true`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Функция может не получить блокировку (и возвратить значение false), даже если блокировка *this не удерживается никаким другим потоком. Насчет продолжительности блокировки не дается никаких гарантий, только если функция вернула false, вызов Clock::now() возвращает время, равное absolute_time в тот момент, когда поток стал разблокированным, или более позднее.

Компонентная функция std::shared_timed_mutex::unlock_shared

Освобождает совместно используемую блокировку объекта `std::shared_timed_mutex`, удерживаемую текущим потоком.

Объявление

```
void unlock_shared();
```

Предусловия

Вызывающий поток должен удерживать блокировку `*this`.

Результат

Освобождает совместно используемую блокировку объекта `*this`, удерживаемую текущим потоком. Если это последняя совместно используемая блокировка `*this` и есть какие-либо потоки, заблокированные в ожидании получения блокировки `*this`, разблокируется один поток, ожидающий эксклюзивной блокировки, или несколько потоков, ожидающих совместно используемой блокировки.

Постусловия

Объект `*this` не заблокирован вызывающим потоком.

Выдает

Ничего не выдает.

Г.5.7. Шаблон класса std::lock_guard

Шаблон класса `std::lock_guard` предоставляет простую оболочку для принадлежности блокировки. Тип блокируемого мьютекса задается параметром шаблона `Mutex` и должен быть `Lockable`. Указанный мьютекс блокируется в конструкторе и разблокируется в деструкторе. Тем самым предоставляется простое средство блокировки мьютекса для блока кода и предоставления гарантии, что мьютекс будет разблокирован при выходе управления из блока независимо от того, произошел он по достижении конца блока за счет использования инструкции управляющего потока, например `break` или `return`, или за счет выдачи исключения.

Экземпляры `std::lock_guard` являются `MoveConstructible`, `CopyConstructible` или `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;
};
```

Блокирующий конструктор std::lock_guard

Создает экземпляр `std::lock_guard`, который блокирует предоставленный мьютекс.

Объявление

```
explicit lock_guard(mutex_type& m);
```

Результат

Создает экземпляр `std::lock_guard`, который ссылается на предоставленный мьютекс. Вызывает `m.lock()`.

Выдает

Любое исключение, выданное функцией `m.lock()`.

Постусловия

Объекту `*this` принадлежит блокировка `m`.

Принимающий блокировку конструктор std::lock_guard

Создает экземпляр `std::lock_guard`, который владеет предоставленным мьютексом.

Объявление

```
lock_guard(mutex_type& m,std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть блокировкой `m`.

Результат

Создает экземпляр `std::lock_guard`, ссылающийся на предоставленный мьютекс и получающий владение блокировкой `m`, удерживаемой вызывающим потоком.

Выдает

Ничего не выдает.

Постусловия

Объект `*this` владеет блокировкой `m`, удерживаемой вызывающим потоком.

Деструктор std::lock_guard

Уничтожает экземпляр `std::lock_guard` и разблокирует соответствующий мьютекс.

Объявление

```
~lock_guard();
```

Результат

Вызывает функцию `m.unlock()` для экземпляра мьютекса, `m`, представленного при создании объекта `*this`.

Выдает

Ничего не выдает.

Г.5.8. Шаблон класса std::scoped_lock

Шаблон класса `std::scoped_lock` предоставляет простую оболочку принадлежности сразу для нескольких мьютексов. Тип блокируемого мьютекса задается пакетом параметров шаблона `Mutexes`, где каждый параметр должен быть `Lockable`. Указанные мьютексы блокируются в конструкторе и разблокируются в деструкторе. Тем самым предоставляется простое средство блокировки набора мьютексов для блока кода и предоставления гарантии, что мьютексы будут разблокированы при выходе управления из блока независимо от того, произошел он по достижении конца блока за счет использования инструкции управляющего потока, например `break` или `return`, или за счет выдачи исключения.

Экземпляры `std::scoped_lock` не являются `MoveConstructible`, `CopyConstructible` или `CopyAssignable`.

Определение класса

```
template <class ... Mutexes>
class scoped_lock
{
public:
    explicit scoped_lock(Mutexes& ... m);
    scoped_lock(Mutexes& ... m, adopt_lock_t);
    ~scoped_lock();

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

Блокирующий конструктор std::scoped_lock

Создает экземпляр `std::scoped_lock`, который блокирует предоставленные мьютексы.

Объявление

```
explicit scoped_lock(Mutexes& ... m);
```

Результат

Создает экземпляр `std::scoped_lock`, ссылающийся на предоставленные мьютексы. Использует комбинацию вызовов `m.lock()`, `m.try_lock()` и `m.unlock()` к каждому из мьютексов во избежание взаимной блокировки, применяя такой же алгоритм, что и свободная функция `std::lock()`.

Выдает

Любые исключения, выданные вызовами `m.lock()` и `m.try_lock()`.

Постусловия

Объект `*this` владеет блокировкой предоставленных мьютексов.

Принимающий блокировку конструктор std::scoped_lock

Создает экземпляр `std::scoped_lock`, владеющий блокировкой предоставленных мьютексов; они уже должны быть заблокированы вызывающим потоком.

Объявление

```
scoped_lock(Mutexes& ... m,std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть блокировкой мьютексов в `m`.

Результат

Создает экземпляр `std::scoped_lock`, ссылающийся на предоставленные мьютексы и получающий владение блокировкой мьютексов в `m`, удерживаемой вызывающим потоком.

Выдает

Ничего не выдает.

Постусловия

Объект `*this` владеет блокировкой предоставленных мьютексов, удерживаемой вызывающим потоком.

Деструктор std::scoped_lock

Уничтожает экземпляр `std::scoped_lock` и разблокирует соответствующие мьютексы.

Объявление

```
~scoped_lock();
```

Результат

Вызывает `m.unlock()` для каждого экземпляра мьютекса `m`, предоставленного при создании объекта `*this`.

Выдает

Ничего не выдает.

Г.5.9. Шаблон класса std::unique_lock

Шаблон класса `std::unique_lock` предоставляет оболочку принадлежности блокировки более общего плана, чем `std::lock_guard`.

Тип блокируемого мьютекса задается параметром шаблона `Mutex` и должен быть `BasicLockable`. Как правило, указанный мьютекс блокируется в конструкторе и разблокируется в деструкторе, но предоставляются также дополнительные конструкторы и компонентные функции, допускающие другие возможности. Тем самым дается средство блокировки мьютекса для блока кода и предоставления гарантии, что мьютекс будет разблокирован при выходе управления из блока независимо от того, произошел он по достижении конца блока за счет использования инструкции управляющего потока, например `break` или `return`, или за счет выдачи исключения. Экземпляры `std::lock_guard` не являются `MoveConstructible`, `CopyConstructible` или `CopyAssignable`. Функции ожидания в классе `std::condition_variable` требуют экземпляра `std::unique_lock<std::mutex>`, и все экземпляры `std::unique_lock` подходят для использования с параметром `Lockable` для функций ожидания `std::condition_variable_any`.

Если предоставляемый тип `Mutex` является `Lockable`, то им также является объект `std::unique_lock<Mutex>`. Если в дополнение к этому предоставляемый тип `Mutex` является `TimedLockable`, то им также является и объект `std::unique_lock<Mutex>`.

Экземпляры `std::unique_lock` являются `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` или `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;
};
```

Конструктор по умолчанию std::unique_lock

Создает экземпляр `std::unique_lock` без связанного с ним мьютекса.

Объявление

```
unique_lock() noexcept;
```

Результат

Создает экземпляр `std::unique_lock`, у которого нет связанного с ним мьютекса.

Постусловия

```
this->mutex()==NULL, this->owns_lock()==false
```

Блокирующий конструктор std::unique_lock

Создает экземпляр `std::unique_lock`, блокирующий предоставленный мьютекс.

Объявление

```
explicit unique_lock(mutex_type& m);
```

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.lock()`.

Выдает

Любые исключения, выданные `m.lock()`.

Постусловия

```
this->owns_lock()==true, this->mutex()==&m
```

Принимающий блокировку конструктор std::unique_lock

Создает экземпляр `std::unique_lock`, владеющий блокировкой предоставленного мьютекса.

Объявление

```
unique_lock(mutex_type& m,std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть блокировкой `m`.

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс и получающий владение блокировкой `m`, удерживаемой вызывающим потоком.

Выдает

Ничего не выдает.

Постусловия

```
this->owns_lock()==true, this->mutex()==&m
```

Конструктор отложенной блокировки std::unique_lock

Создает экземпляр `std::unique_lock`, не владеющий блокировкой предоставленного мьютекса.

Объявление

```
unique_lock(mutex_type& m,std::defer_lock_t) noexcept;
```

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс.

Выдает

Ничего не выдает.

Постусловия

```
this->owns_lock()==false, this->mutex()==&m
```

Конструктор попытки блокирования std::unique_lock

Создает экземпляр `std::unique_lock`, связанный с предоставленным мьютексом и предпринимающий попытку заблокировать этот мьютекс.

Объявление

```
unique_lock(mutex_type& m,std::try_to_lock_t);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `Lockable`.

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс.

Вызывает `m.try_lock()`.

Выдает

Ничего не выдает.

Постусловия

Функция `this->owns_lock()` возвращает результат вызова `m.try_lock()`, `this->mutex()==&m`.

Конструктор попытки блокирования с тайм-аутом продолжительности std::unique_lock

Создает экземпляр `std::unique_lock`, связанный с предоставленным мьютексом, и пытается получить блокировку этого мьютекса.

Объявление

```
template<typename Rep,typename Period>
unique_lock(
    mutex_type& m,
    std::chrono::duration<Rep,Period> const& relative_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `TimedLockable`.

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.try_lock_for(relative_time)`.

Выдает

Ничего не выдает.

Постусловия

Функция `this->owns_lock()` возвращает результат вызова `m.try_lock_for()`, `this->mutex()==&m`.

Конструктор попытки блокирования с тайм-аутом по моменту времени time_point std::unique_lock

Создает экземпляр `std::unique_lock`, связанный с предоставленным мьютексом, и пытается получить блокировку этого мьютекса.

Объявление

```
template<typename Clock,typename Duration>
unique_lock(
    mutex_type& m,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `TimedLockable`.

Результат

Создает экземпляр `std::unique_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.try_lock_until(absolute_time)`.

Выдает

Ничего не выдает.

Постусловия

Функция `this->owns_lock()` возвращает результат вызова `m.try_lock_until()`, `this->mutex()==&m`.

Перемещающий конструктор std::unique_lock

Перемещает принадлежность блокировки из одного объекта `std::unique_lock` в только что созданный объект `std::unique_lock`.

Объявление

```
unique_lock(unique_lock&& other) noexcept;
```

Результат

Создает экземпляр `std::unique_lock`. Если `other` владел блокировкой мьютекса до вызова конструктора, то теперь этой блокировкой владеет только что созданный объект `std::unique_lock`.

Постусловия

Для только что созданного объекта `std::unique_lock`, `x` , результат вычисления `x.mutex()` равен значению `other.mutex()` до вызова конструктора, а результат вычисления `x.owns_lock()` равен значению `other.owns_lock()` до вызова конструктора. `other.mutex()==NULL`, `other.owns_lock()==false`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объекты std::unique_lock не являются CopyConstructible, поэтому копирующий конструктор отсутствует и имеется только этот перемещающий конструктор.

Перемещающий оператор присваивания std::unique_lock

Перемещает принадлежность блокировки из одного объекта `std::unique_lock` в другой объект `std::unique_lock`.

Объявление

```
unique_lock& operator=(unique_lock&& other) noexcept;
```

Результат

Если `this->owns_lock()` возвращает `true` до вызова, вызывает `this->unlock()`. Если `other` владел совместно используемой блокировкой мьютекса до присваивания, то теперь этой блокировкой владеет `*this`.

Постусловия

Значение `this->mutex()` равно значению `other.mutex()` до присваивания, а значение `this->owns_lock()` равно значению `other.owns_lock()` до присваивания. `other.mutex()==NULL` , `other.owns_lock()==false`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объекты std::unique_lock не являются CopyConstructible, поэтому оператор копирующего присваивания отсутствует и имеется только этот оператор перемещающего присваивания.

Деструктор std::unique_lock

Уничтожает экземпляр `std::unique_lock` и разблокирует соответствующий мьютекс, если им владел уничтоженный экземпляр.

Объявление

```
~unique_lock();
```

Результат

Если `this->owns_lock()` возвращает `true`, вызывает `this->mutex->unlock()`.

Выдает

Ничего не выдает.

Компонентная функция std::unique_lock::swap

Производит обмен принадлежности связанных с ними блокировок `unique_locks` выполнения между двумя объектами `std::unique_lock`.

Объявление

```
void swap(unique_lock& other) noexcept;
```

Результат

Если `other` владеет блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `*this`. Если `*this` владеет блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `other`.

Постусловия

Значение `this->mutex()` равно значению `other.mutex()` до вызова. Значение `other.mutex()` равно значению `this->mutex()` до вызова. Значение `this->owns_lock()` равно значению `other.owns_lock()` до вызова. Значение `other.owns_lock()` равно значению `this->owns_lock()` до вызова.

Выдает

Ничего не выдает.

Некомпонентная функция обмена для std::unique_lock

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)
```

Выдает

Ничего не выдает.

Компонентная функция std::unique_lock::lock

Получает блокировку мьютекса, связанную с `*this`.

Объявление

```
void lock();
```

Предусловия

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->lock()`.

Выдает

Любые исключения, выданные вызовом `this->mutex()->lock()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

```
this->owns_lock()==true
```

Компонентная функция std::unique_lock::try_lock

Пытается получить блокировку мьютекса, связанную с `*this`.

Объявление

```
bool try_lock();
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `Lockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock()`.

Возвращает

`true`, если вызов `this->mutex()->try_lock()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Компонентная функция std::unique_lock::unlock

Освобождает блокировку мьютекса, связанную с `*this`.

Объявление

```
void unlock();
```

Предусловия

```
this->mutex()!=NULL, this->owns_lock()==true
```

Результат

Вызывает `this->mutex()->unlock()`.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_unlock()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если на входе `this->owns_lock()==false`.

Постусловия

```
this->owns_lock()==false
```

Компонентная функция std::unique_lock::try_lock_for

Пытается получить блокировку мьютекса, связанную с `*this`, в течение заданного времени.

Объявление

```
template<typename Rep, typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `TimedLockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock_for(relative_time)`.

Возвращает

`true`, если вызов `this->mutex()->try_lock_for()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_for()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Компонентная функция std::unique_lock::try_lock_until

Пытается получить блокировку мьютекса, связанную с `*this`, в течение заданного времени.

Объявление

```
template<typename Clock, typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::unique_lock`, должен быть `TimedLockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock_until(absolute_time)`.

Возвращает

`true`, если вызов `this->mutex()->try_lock_until()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_until()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловие

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Булева компонентная функция std::unique_lock::operator

Проверяет принадлежность блокировки мьютекса объекту `*this`.

Объявление

```
explicit operator bool() const noexcept;
```

Возвращает

```
this->owns_lock()
```

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это оператор явного преобразования, поэтому он подразумеваемо вызывается в контексте, где результат используется как булево значение, а не там, где результат будет рассматриваться как целочисленное значение, равное 0 или 1.

Компонентная функция std::unique_lock::owns_lock

Проверяет принадлежность блокировки мьютекса объекту `*this`.

Объявление

```
bool owns_lock() const noexcept;
```

Возвращает

`true`, если `*this` владеет блокировкой мьютекса, или `false` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::unique_lock::mutex

Возвращает мьютекс, связанный с `*this`, если таковой имеется.

Объявление

```
mutex_type* mutex() const noexcept;
```

Возвращает

Указатель на мьютекс, связанный с `*this`, если таковой имеется, или `NULL` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::unique_lock::release

Возвращает мьютекс, связанный с `*this`, если таковой имеется, и освобождает эту связь.

Объявление

```
mutex_type* release() noexcept;
```

Результат

Разрывает связь мьютекса с `*this` без разблокировки любых удерживаемых блокировок.

Возвращает

Указатель на мьютекс, связанный с `*this` до вызова, если таковой имелся, или `NULL` в противном случае.

Постусловия

```
this->mutex()==NULL, this->owns_lock()==false
```

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если до вызова this->owns_lock() возвратил бы true, то тогда ответственность за разблокировку мьютекса возлагалась бы на вызывающий код.

Г.5.10. Шаблон класса std::shared_lock

Шаблон класса `std::shared_lock` предоставляет эквивалент `std::unique_lock`, за исключением того, что он получает совместно используемую, а не исключительную блокировку. Тип блокируемого мьютекса задается параметром шаблона `Mutex`, который должен обладать свойством `SharedLockable`. Как правило, указанный мьютекс блокируется в конструкторе и разблокируется в деструкторе, но предоставляются также дополнительные конструкторы и компонентные функции, допускающие

другие возможности. Тем самым предоставляется средство блокировки мьютекса для блока кода и предоставления гарантии, что мьютекс будет разблокирован при выходе управления из блока независимо от того, произошел он по достижении конца блока за счет использования инструкции управляющего потока, например `break` или `return`, или за счет выдачи исключения. Все экземпляры `std::shared_lock` подходят для использования с параметром `Lockable` для функций ожидания `std::condition_variable_any`.

Каждый объект `std::shared_lock<Mutex>` является `Lockable`. Если в дополнение к этому предоставляемый тип `Mutex` является `SharedTimedLockable`, то `std::shared_lock<Mutex>` также является `TimedLockable`.

Экземпляры `std::shared_lock` являются `MoveConstructible` и `MoveAssignable`, но не `CopyConstructible` или `CopyAssignable`.

Определение класса

```
template <class Mutex>
class shared_lock
{
public:
    typedef Mutex mutex_type;

    shared_lock() noexcept;
    explicit shared_lock(mutex_type& m);
    shared_lock(mutex_type& m, adopt_lock_t);
    shared_lock(mutex_type& m, defer_lock_t) noexcept;
    shared_lock(mutex_type& m, try_to_lock_t);

    template<typename Clock,typename Duration>
    shared_lock(
        mutex_type& m,
        std::chrono::time_point<Clock,Duration> const& absolute_time);

    template<typename Rep,typename Period>
    shared_lock(
        mutex_type& m,
        std::chrono::duration<Rep,Period> const& relative_time);

    ~shared_lock();

    shared_lock(shared_lock const& ) = delete;
    shared_lock& operator=(shared_lock const& ) = delete;

    shared_lock(shared_lock&& );
    shared_lock& operator=(shared_lock&& );

    void swap(shared_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;
};
```

Конструктор по умолчанию std::shared_lock

Создает экземпляр `std::shared_lock` без связанного с ним мьютекса.

Объявление

```
shared_lock() noexcept;
```

Результат

Создает экземпляр `std::shared_lock`, не имеющий связанного с ним мьютекса.

Постусловия

```
this->mutex()==NULL, this->owns_lock()==false
```

Блокирующий конструктор std::shared_lock

Создает экземпляр `std::shared_lock`, получающий совместно используемую блокировку предоставленного мьютекса.

Объявление

```
explicit shared_lock(mutex_type& m);
```

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.lock_shared()`.

Выдает

Любые исключения, выданные `m.lock_shared()`.

Постусловия

```
this->owns_lock()==true, this->mutex()==&m
```

Принимающий блокировку конструктор std::shared_lock

Создает экземпляр `std::shared_lock`, владеющий блокировкой предоставленного мьютекса.

Объявление

```
shared_lock(mutex_type& m,std::adopt_lock_t);
```

Предусловия

Вызывающий поток должен владеть совместно используемой блокировкой `m`.

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс и получающий владение совместно используемой блокировкой `m`, удерживаемой вызывающим потоком.

Выдает

Ничего не выдает.

Постусловия

```
this->owns_lock()==true, this->mutex()==&m
```

Конструктор отложенной блокировки std::shared_lock

Создает экземпляр `std::shared_lock`, не владеющий блокировкой предоставленного мьютекса.

Объявление

```
shared_lock(mutex_type& m,std::defer_lock_t) noexcept;
```

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс.

Выдает

Ничего не выдает.

Постусловия

```
this->owns_lock()==false, this->mutex()==&m
```

Конструктор попытки блокирования std::shared_lock

Создает экземпляр `std::shared_lock`, связанный с предоставленным мьютексом и предпринимающий попытку получения совместно используемой блокировки этого мьютекса.

Объявление

```
shared_lock(mutex_type& m,std::try_to_lock_t);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `Lockable`.

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.try_lock_shared()` .

Выдает

Ничего не выдает.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock_shared()`, `this ->mutex()==&m`.

Конструктор попытки блокирования с тайм-аутом продолжительностью std::shared_lock

Создает экземпляр `std::shared_lock`, связанный с предоставленным мьютексом и предпринимающий попытку получения совместно используемой блокировки этого мьютекса.

Объявление

```
template<typename Rep,typename Period>
shared_lock(
    mutex_type& m,
    std::chrono::duration<Rep,Period> const& relative_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `SharedTimedLockable`.

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.try_lock_shared_for(relative_time)`.

Выдает

Ничего не выдает.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock_shared()`, `this ->mutex()==&m`.

Конструктор попытки блокирования с тайм-аутом по моменту времени time_point std::shared_lock

Создает экземпляр `std::shared_lock`, связанный с предоставленным мьютексом и предпринимающий попытку получения совместно используемой блокировки этого мьютекса.

Объявление

```
template<typename Clock,typename Duration>
shared_lock(
    mutex_type& m,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `SharedTimedLockable`.

Результат

Создает экземпляр `std::shared_lock`, ссылающийся на предоставленный мьютекс. Вызывает `m.try_lock_shared_until(absolute_time)`.

Выдает

Ничего не выдает.

Постусловия

`this->owns_lock()` возвращает результат вызова `m.try_lock_shared()`, `this ->mutex()==&m`.

Перемещающий конструктор std::shared_lock

Перемещает принадлежность совместно используемой блокировки из одного объекта `std::shared_lock` в только что созданный объект `std::shared_lock`.

Объявление

```
shared_lock(shared_lock&& other) noexcept;
```

Результат

Создает экземпляр `std::shared_lock`. Если `other` владел совместно используемой блокировкой мьютекса до вызова конструктора, то теперь этой блокировкой владеет только что созданный объект `std::shared_lock`.

Постусловия

Для только что созданного объекта `std::shared_lock`, `x`, результат вычисления `x.mutex()` равен значению `other.mutex()` до вызова конструктора, а результат вычисления `x.owns_lock()` равен значению `other.owns_lock()` до вызова конструктора. `other.mutex()==NULL`, `other.owns_lock()==false`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объект std::shared_lock не является CopyConstructible, поэтому копирующий конструктор отсутствует и имеется только этот перемещающий конструктор.

Перемещающий оператор присваивания std::shared_lock

Перемещает принадлежность совместно используемой блокировки из одного объекта `std::shared_lock` в другой объект `std::shared_lock`.

Объявление

```
shared_lock& operator=(shared_lock&& other) noexcept;
```

Результат

Если `this->owns_lock()` возвращает `true` до вызова, вызывает `this->unlock()`. Если `other` владел совместно используемой блокировкой мьютекса до присваивания, то теперь этой блокировкой владеет `*this`.

Постусловия

Значение `this->mutex()` равно значению `other.mutex()` до присваивания, а значение `this->owns_lock()` равно значению `other.owns_lock()` до присваивания. `other.mutex()==NULL` , `other.owns_lock()==false`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объект std::shared_lock не является CopyConstructible, поэтому оператор копирующего присваивания отсутствует и имеется только этот оператор перемещающего присваивания.

Деструктор std::shared_lock

Уничтожает экземпляр `std::shared_lock` и разблокирует соответствующий мьютекс, если им владел уничтоженный экземпляр.

Объявление

```
~shared_lock();
```

Результат

Если `this->owns_lock()` возвращает `true`, вызывает `this->mutex->unlock_shared()`.

Выдает

Ничего не выдает.

Компонентная функция std::shared_lock::swap

Производит обмен принадлежности связанных с ними блокировок `shared_locks` выполнения между двумя объектами `std::shared_locks`.

Объявление

```
void swap(shared_lock& other) noexcept;
```

Результат

Если `other` владеет блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `*this`. Если `*this` владеет блокировкой мьютекса до вызова, то теперь этой блокировкой владеет `other`.

Постусловия

Значение `this->mutex()` равно значению `other.mutex()` до вызова. Значение `other.mutex()` равно значению `this->mutex()` до вызова. Значение `this->owns_lock()` равно значению `other.owns_lock()` до вызова. Значение `other.owns_lock()` равно значению `this->owns_lock()` до вызова.

Выдает

Ничего не выдает.

Некомпонентная функция обмена для std::shared_lock

Производит обмен принадлежности связанных с ними блокировок мьютекса между двумя объектами `std::shared_lock`.

Объявление

```
void swap(shared_lock& lhs,shared_lock& rhs) noexcept;
```

Результат

```
lhs.swap(rhs)
```

Выдает

Ничего не выдает.

Компонентная функция std::shared_lock::lock

Получает совместно используемую блокировку мьютекса, связанную с `*this`.

Объявление

```
void lock();
```

Предусловия

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->lock_shared()`.

Выдает

Любые исключения, выданные вызовом `this->mutex()->lock_shared()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

```
this->owns_lock()==true
```

Компонентная функция std::shared_lock::try_lock

Пытается получить совместно используемую блокировку мьютекса, связанную с `*this`.

Объявление

```
bool try_lock();
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `Lockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock_shared()`.

Возвращает

`true`, если вызов `this->mutex()->try_lock_shared()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_shared()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Компонентная функция std::shared_lock::unlock

Освобождает совместно используемую блокировку мьютекса, связанную с `*this`.

Объявление

```
void unlock();
```

Предусловия

```
this->mutex()!=NULL, this->owns_lock()==true
```

Результат

Вызывает `this->mutex()->unlock_shared()`.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_shared()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если на входе `this->owns_lock()==false`.

Постусловия

```
this->owns_lock()==false
```

Компонентная функция std::shared_lock::try_lock_for

Пытается получить совместно используемую блокировку мьютекса, связанную с `*this`, в течение заданного времени.

Объявление

```
template<typename Rep, typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `SharedTimedLockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock_shared_for(relative_time)`.

Возвращает

`true`, если вызов `this->mutex()->try_lock_shared_for()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_shared_for()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловия

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Компонентная функция std::shared_lock::try_lock_until

Пытается получить совместно используемую блокировку мьютекса, связанную с `*this`, в течение заданного времени.

Объявление

```
template<typename Clock, typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Предусловия

Тип `Mutex`, использованный для создания экземпляра `std::shared_lock`, должен быть `SharedTimedLockable`.

```
this->mutex()!=NULL, this->owns_lock()==false
```

Результат

Вызывает `this->mutex()->try_lock_shared_until(absolute_time)`.

Возвращает

`true`, если вызов `this->mutex()->try_lock_shared_until()` возвращает `true`, или `false` в противном случае.

Выдает

Любые исключения, выданные вызовом `this->mutex()->try_lock_shared_until()`. Исключение `std::system_error` с кодом ошибки `std::errc::operation_not_permitted`, если `this->mutex()==NULL`. Исключение `std::system_error` с кодом ошибки `std::errc::resource_deadlock_would_occur`, если на входе `this->owns_lock()==true`.

Постусловие

Если функция возвращает `true`, `this->owns_lock()==true`, в противном случае `this->owns_lock()==false`.

Булева компонентная функция std::shared_lock::operator

Проверяет принадлежность блокировки совместно используемого мьютекса объекту `*this`.

Объявление

```
explicit operator bool() const noexcept;
```

Возвращает

```
this->owns_lock()
```

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Это оператор явного преобразования, поэтому он подразумеваемо вызывается в контексте, где результат используется как булево значение, а не там, где результат будет рассматриваться как целочисленное значение, равное 0 или 1.

Компонентная функция std::shared_lock::owns_lock

Проверяет принадлежность блокировки совместно используемого мьютекса объекту `*this`.

Объявление

```
bool owns_lock() const noexcept;
```

Возвращает

`true`, если `*this` владеет совместно используемой блокировкой мьютекса, или `false` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::shared_lock::mutex

Возвращает мьютекс, связанный с `*this`, если таковой имеется.

Объявление

```
mutex_type* mutex() const noexcept;
```

Возвращает

Указатель на мьютекс, связанный с `*this`, если таковой имеется, или `NULL` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::shared_lock::release

Возвращает мьютекс, связанный с `*this`, если таковой имеется, и освобождает эту связь.

Объявление

```
mutex_type* release() noexcept;
```

Результат

Разрывает связь мьютекса с `*this` без разблокировки любых удерживаемых блокировок.

Возвращает

Указатель на мьютекс, связанный с `*this` до вызова, если таковой имелся, или `NULL` в противном случае.

Постусловия

```
this->mutex()==NULL, this->owns_lock()==false
```

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Если до вызова this->owns_lock() возвратил бы true, то тогда ответственность за разблокировку мьютекса возлагалась бы на вызывающий код.

Г.5.11. Шаблон функции std::lock

Шаблон функции `std::lock` предоставляет средство блокировки более чем одного мьютекса одновременно, без риска возникновения взаимной блокировки, получаемой в результате несогласованного порядка блокировок.

Объявление

```
template<typename LockableType1,typename... LockableType2>
void lock(LockableType1& m1,LockableType2& m2...);
```

Предусловия

Типы предоставленных блокируемых объектов, `LockableType1`, `LockableType2`, ..., должны быть `Lockable`.

Результат

Получает блокировку каждого из предоставленных блокируемых объектов, `m1`, `m2`, `...`, путем неуказанной последовательности вызовов компонентных функций `lock()`, `try_lock()` и `unlock()` этих типов, избегающих взаимной блокировки.

Постусловия

Текущий поток владеет блокировкой каждого из предоставленных блокируемых объектов.

Выдает

Любые исключения, выданные вызовами `lock()`, `try_lock()` и `unlock()`.

ПРИМЕЧАНИЕ

Если исключение распространяется за пределы вызова std::lock, то функция unlock() должна была быть вызвана для любого из объектов m1, m2, ..., для которого блокировка была получена в функции путем вызова lock() или try_lock().

Г.5.12. Шаблон функции std::try_lock

Шаблон функции `std::try_lock` позволяет предпринять попытку блокировки сразу набора блокируемых объектов, чтобы либо все они стали заблокированы, либо ни один из них не был заблокирован.

Объявление

```
template<typename LockableType1,typename... LockableType2>
int try_lock(LockableType1& m1,LockableType2& m2...);
```

Предусловия

Типы предоставленных блокируемых объектов, `LockableType1`, `LockableType2`, ..., должны быть `Lockable`.

Результат

Пытается получить блокировку каждого из предоставленных блокируемых объектов, `m1`, `m2`, `...`, путем вызова `try_lock()` по очереди для каждого из них. Если вызов `try_lock()` привел к возвращению значения `false` или выдал исключение, уже полученная блокировка освобождается путем вызова функции `unlock()` в отношении соответствующего блокируемого объекта.

Возвращает

`-1`, если были получены все блокировки (каждый вызов `try_lock()` привел к возвращению значения `true`), в противном случае — начинающийся с нуля индекс объекта, для которого вызов `try_lock()` привел к возвращению значения `false`.

Постусловия

Если функция вернула `-1`, то текущий поток владеет блокировкой каждого из предоставленных блокируемых объектов. В противном случае все блокировки, полученные в результате ее вызова, освобождаются.

Выдает

Любые исключения, выданные при вызове функции `try_lock()` .

ПРИМЕЧАНИЕ

Если исключение распространяется за пределы вызова std::try_lock, то функция unlock() должна была быть вызвана для любого из объектов m1, m2, ..., для которого блокировка была получена в функции путем вызова lock() или try_lock().

Г.5.13. Класс std::once_flag

Экземпляры класса `std::once_flag` используются вместе с шаблоном функции `std::call_once`, чтобы гарантировать, что конкретная функция вызывается только один раз, даже если вызов одновременно инициируется сразу несколькими потоками.

Экземпляры `std::once_flag` не являются `CopyConstructible`, `CopyAssignable`, `MoveConstructible` или `MoveAssignable`.

Определение класса

```
struct once_flag
{
    constexpr once_flag() noexcept;

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

Конструктор по умолчанию std::once_flag. Конструктор `std::once_flag` создает новый экземпляр `std::once_flag` в состоянии, показывающем, что связанная функция не была вызвана.

Объявление

```
constexpr once_flag() noexcept;
```

Результат

Создает новый экземпляр `std::once_flag` в состоянии, показывающем, что связанная функция не была вызвана. Поскольку конструктор помечен спецификатором `constexpr`, экземпляр со статической продолжительностью хранения создается в качестве составной части этапа статической инициализации, что исключает возможность возникновения гонки за данными и проблем, связанных с порядком инициализации.

Г.5.14. Шаблон функции std::call_once

Шаблон функции `std::call_once` используется с экземпляром `std::once_flag`, чтобы гарантировать, что конкретная функция вызывается только один раз, даже если вызов одновременно инициируется сразу несколькими потоками.

Объявление

```
template<typename Callable,typename... Args>
void call_once(std::once_flag& flag,Callable func,Args args...);
```

Предусловия

Выражение `INVOKE(func,args)` допустимо для предоставленных значений `func` и `args`. Объекты типа `Callable` и все компоненты `Args` являются `MoveConstructible`.

Результат

Вызовы `std::call_once` в отношении того же самого объекта `std::once_flag` выполняются последовательно.

Если ранее не было результативного вызова `std::call_once` в отношении того же самого объекта `std::once_flag`, аргумент `func` (или его копия) вызывается словно при вызове `INVOKE(func,args)`, а вызов `std::call_once` результативен только в том случае, если вызов `func` возвращается без выдачи исключения. Если выдано исключение, оно распространяется до вызывающего кода. Если ранее уже был результативный вызов `std::call_once` в отношении того же самого объекта `std::once_flag`, вызов `std::call_once` возвращает управление без вызова `func`.

Синхронизация

Завершение результативного вызова `std::call_once` в отношении объекта `std::once_flag` «происходит до» всех последующих вызовов `std::call_once` в отношении того же самого объекта `std::once_flag`.

Выдает

Исключение `std::system_error`, когда результат не может быть достигнут, или для любого исключения, распространенного из вызова `func`.

Г.6. Заголовок <ratio>

Заголовок `<ratio>` предоставляет поддержку арифметических действий с рациональными числами на этапе компиляции.

Содержимое заголовка

```
namespace std
{
    template<intmax_t N,intmax_t D=1>
    class ratio;

    // ratio-арифметика
    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 R1, class R2>
    using ratio_divide = см. описание;

    // ratio-сравнение
    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;
}
```

Г.6.1. Шаблон класса std::ratio

Шаблон класса `std::ratio` предоставляет механизм для выполнения арифметических действий на этапе компиляции, в которой используются рациональные значения, например одна вторая (`std::ratio<1,2>`), две трети (`std::ratio<2,3>`) или пятнадцать сорок третьих (`std::ratio<15,43>`). Он используется внутри стандартной библиотеки C++ для указания периода при создании экземпляра 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 нулем.

Описание

num и den являются числителем и знаменателем дроби N/D после ее предельного сокращения. den всегда является положительным числом. Если N и D имеют одинаковые значки, num является положительным числом; в противном случае num является отрицательным числом.

Примеры

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

Г.6.2. Псевдоним шаблона std::ratio_add

Псевдоним шаблона std::ratio_add предоставляет механизм для сложения двух значений std::ratio на этапе компиляции с применением арифметических действий с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

Предусловия

R1 и R2 должны быть экземплярами шаблона класса std::ratio.

Результат

ratio_add<R1,R2> определяется в качестве псевдонима для создания экземпляра std::ratio, представляющего сумму дробей, представленных параметрами R1 и R2, если эта сумма может быть вычислена без переполнения. Если при вычислении результата возникает переполнение, программа считается некорректной. В отсутствие арифметического переполнения у std::ratio_add<R1,R2> должны быть такие же значения num и den, что и у 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
```

Г.6.3. Псевдоним шаблона std::ratio_subtract

Псевдоним шаблона std::ratio_subtract предоставляет механизм для вычитания двух значений std::ratio на этапе компиляции с применением арифметических действий с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

Предусловия

`R1` и `R2` должны быть экземплярами шаблона класса `std::ratio`.

Результат

`ratio_subtract<R1,R2>` определяется в качестве псевдонима для создания экземпляра `std::ratio`, представляющего разность дробей, представленных параметрами `R1` и `R2`, если эта разность может быть вычислена без переполнения. Если при вычислении результата возникает переполнение, программа считается некорректной. В отсутствие арифметического переполнения у `std::ratio_subtract<R1,R2>` должны быть такие же значения `num` и `den`, что и у `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
```

Г.6.4. Псевдоним шаблона std::ratio_multiply

Псевдоним шаблона `std::ratio_multiply` предоставляет механизм для умножения двух значений `std::ratio` на этапе компиляции с применением арифметических действий с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

Предусловия

`R1` и `R2` должны быть экземплярами шаблона класса `std::ratio`.

Результат

`ratio_multiply<R1,R2>` определяется в качестве псевдонима для создания экземпляра `std::ratio`, представляющего произведение дробей, представленных параметрами `R1` и `R2`, если это произведение может быть вычислено без переполнения. Если при вычислении результата возникает переполнение, программа считается некорректной. В отсутствие арифметического переполнения у `std::ratio_multiply<R1,R2>` должны быть такие же значения `num` и `den`, что и у `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
```

Г.6.5. Псевдоним шаблона std::ratio_divide

Псевдоним шаблона `std::ratio_divide` предоставляет механизм для деления двух значений `std::ratio` на этапе компиляции с применением арифметических действий с рациональными числами.

Определение

```
template <class R1, class R2>
using ratio_divide = std::ratio<see below>;
```

Предусловия

`R1` и `R2` должны быть экземплярами шаблона класса `std::ratio`.

Результат

`ratio_divide<R1,R2>` определяется в качестве псевдонима для создания экземпляра `std::ratio`, представляющего частное дробей, представленных параметрами `R1` и `R2`, если это частное может быть вычислено без переполнения. Если при вычислении результата возникает переполнение, программа считается некорректной. В отсутствие арифметического переполнения у `ratio_divide<R1,R2>` должны быть такие же значения `num` и `den`, что и у `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
```

Г.6.6. Шаблон класса std::ratio_equal

Псевдоним шаблона `std::ratio_equal` предоставляет механизм для сравнения на равенство двух значений `std::ratio` на этапе компиляции с применением арифметических действий 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 экземплярами шаблона класса `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
```

Г.6.7. Шаблон класса std::ratio_not_equal

Псевдоним шаблона `std::ratio_not_equal` предоставляет механизм для сравнения на неравенство двух значений `std::ratio` на этапе компиляции с применением арифметических действий 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 экземплярами шаблона класса `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
```

Г.6.8. Шаблон класса std::ratio_less

Псевдоним шаблона `std::ratio_less` предоставляет механизм для сравнения двух значений `std::ratio` на этапе компиляции с применением арифметических действий с рациональными числами.

Определение класса

```
template <class R1, class R2>
class ratio_less:
    public std::integral_constant<bool,see below>
{};
```

Предусловия

`R1` и `R2` должны быть экземплярами шаблона класса `std::ratio`.

Результат

`std::ratio_less<R1,R2>` выводится из шаблона `std::integral_constant<bool, value>`, где `value` — это `(R1::num * R2::den) < (R2::num * R1::den)`. По возможности в реализации должен использоваться метод вычисления результата, избегающий переполнения. Если возникает переполнение, 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
```

Г.6.9. Шаблон класса std::ratio_greater

Псевдоним шаблона `std::ratio_greater` предоставляет механизм для сравнения двух значений `std::ratio` на этапе компиляции с применением арифметических действий 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 экземплярами шаблона класса `std::ratio`.

Г.6.10. Шаблон класса std::ratio_less_equal

Псевдоним шаблона `std::ratio_less_equal` предоставляет механизм для сравнения двух значений `std::ratio` на этапе компиляции с применением арифметических действий 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 экземплярами шаблона класса `std::ratio`.

Г.6.11. Шаблон класса std::ratio_greater_equal

Псевдоним шаблона `std::ratio_greater_equal` предоставляет механизм для сравнения двух значений `std::ratio` на этапе компиляции с применением арифметических действий 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 экземплярами шаблона класса `std::ratio`.

Г.7. Заголовок <thread>

Заголовок `<thread>` предоставляет средства управления потоками и идентификации потоков, а также функции для ввода текущего потока в спящий режим.

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);
    }
}
```

Г.7.1. Класс std::thread

Класс `std::thread` используется для управления потоком выполнения. Он предоставляет средства для запуска нового потока выполнения и ожидания завершения потока выполнения. Он также предоставляет средства для идентификации и другие функции для управления потоками 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);
```

Класс std::thread::id

Экземпляр `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);
```

Примечания

Значение `std::thread::id`, идентифицирующее конкретный поток выполнения, должно отличаться от значения изначально созданного экземпляра `std::thread::id` и от любого значения, представляющего другой поток выполнения.

Значение `std::thread::id` для конкретного потока непредсказуемо и может варьироваться между выполнениями одной и той же программы.

Объект `std::thread::id` является `CopyConstructible` и `CopyAssignable`, поэтому экземпляры `std::thread::id` можно свободно копировать и присваивать.

Конструктор по умолчанию std::thread::id

Создает объект `std::thread::id`, не представляющий какой-либо поток выполнения.

Объявление

```
id() noexcept;
```

Результат

Создает экземпляр `std::thread::id`, имеющий особое значение, *не относящееся ни к какому* потоку.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Все изначально созданные экземпляры std::thread::id хранят одно и то же значение.

Оператор сравнения для определения равенства std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не представляют ли они один и тот же поток выполнения.

Объявление

```
bool operator==(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

`true`, если и `lhs`, и `rhs` представляют один и тот же поток выполнения или оба они имеют особое значение, *не имеющее отношения ни к одному* из потоков. `false`, если `lhs` и `rhs` представляют разные потоки выполнения, или один представляет поток выполнения, а другой имеет особое значение, *не имеющее отношения ни к одному* из потоков.

Выдает

Ничего не выдает.

Оператор сравнения для определения неравенства std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не представляют ли они разные потоки выполнения.

Объявление

```
bool operator!=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

```
!(lhs==rhs)
```

Выдает

Ничего не выдает.

Оператор сравнения для определения соотношения «меньше» std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не находится ли один перед другим в общем порядке значений идентификаторов потоков.

Объявление

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

`true`, если значение `lhs` оказывает до значения `rhs` в общем порядке значений идентификаторов потоков. Если `lhs!=rhs`, то одно из вычислений, `lhs<rhs` или `rhs<lhs`, возвращает `true`, а другое возвращает `false`. Если `lhs==rhs`, оба вычисления, `lhs<rhs` и `rhs<lhs`, возвращают `false`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Особое значение, не имеющее отношения ни к одному из потоков, которое хранится в изначально созданном экземпляре std::thread::id, в сравнении оказывается меньше значения любого экземпляра std::thread::id, представляющего поток выполнения. Если два экземпляра std::thread::id равны, то ни один из них не меньше другого. Любой набор различных значений std::thread::id формирует общий порядок, остающийся постоянным на всем протяжении выполнения программы. Между выполнениями одной и той же программы этот порядок может варьироваться.

Оператор сравнения для определения соотношения «меньше или равно» std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не находится ли один перед другим в общем порядке значений идентификаторов потоков или не равен ли он ему.

Объявление

```
bool operator<=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

```
!(rhs<lhs)
```

Выдает

Ничего не выдает.

Оператор сравнения для определения соотношения «больше» std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не находится ли один после другого в общем порядке значений идентификаторов потоков.

Объявление

```
bool operator>(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

```
rhs<lhs
```

Выдает

Ничего не выдает.

Оператор сравнения для определения соотношения «больше или равно» std::thread::id

Сравнивает два экземпляра `std::thread::id`, чтобы определить, не находится ли один после другого в общем порядке значений идентификаторов потоков или не равен ли он ему.

Объявление

```
bool operator>=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

Возвращает

```
!(lhs<rhs)
```

Выдает

Ничего не выдает.

Оператор вставки в поток std::thread::id

Записывает строку представления 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

Вставляет строку представления значения `std::thread::id` в указанный поток.

Возвращает

```
out
```

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Формат строкового представления точно не определен. Экземпляры std::thread::id, сравниваемые на равенство, имеют одинаковые представления, а экземпляры, не равные друг другу, имеют различные представления.

Псевдоним типа std::thread::native_handle_type

`native_handle_type` является псевдонимом типа, которым можно воспользоваться вместе с API конкретной платформы.

Объявление

```
typedef implementation-defined native_handle_type;
```

ПРИМЕЧАНИЕ

Этот псевдоним типа носит необязательный характер. Если он присутствует, реализация должна предоставить тип, подходящий для использования с исходными API конкретной платформы.

Компонентная функция std::thread::native_handle

Возвращает значение типа `native_handle_type`, представляющее поток выполнения, связанный с `*this`.

Объявление

```
native_handle_type native_handle();
```

ПРИМЕЧАНИЕ

Эта функция носит необязательный характер. Если она присутствует, то возвращаемое значение должно быть подходящим для использования с исходными API конкретной платформы.

Конструктор по умолчанию std::thread

Создает объект `std::thread`, не связанный с потоком выполнения.

Объявление

```
thread() noexcept;
```

Результат

Создает экземпляр `std::thread`, не имеющий связанного с ним потока выполнения.

Постусловия

Для только что созданного объекта `std::thread`, `x`, `x.get_id()==id()`.

Выдает

Ничего не выдает.

Конструктор std::thread

Создает объект `std::thread`, связанный с новым потоком выполнения.

Объявление

```
template<typename Callable,typename Args...>
explicit thread(Callable&& func,Args&&... args);
```

Предусловия

`func` и каждый элемент `args` должны быть `MoveConstructible`.

Результат

Создает экземпляр `std::thread` и связывает его с только что созданным потоком выполнения. Копирует или перемещает `func` и каждый элемент `args` во внутреннее

хранилище, существующее на протяжении всего жизненного цикла нового потока выполнения. Выполняет `INVOKE(copy-of-func,copy-of-args)` в отношении нового потока выполнения.

Постусловия

Для только что созданного объекта `std::thread`, `x`, `x.get_id()!=id()`.

Выдает

Исключение типа `std::system_error`, если не может запустить новый поток. Любое исключение, выданное при копировании `func` или `args` во внутреннее хранилище.

Синхронизация

Вызов конструктора «происходит до» выполнения предоставленной функции, выполняемой в отношении только что созданного потока выполнения.

Перемещающий конструктор std::thread

Перемещает принадлежность потока выполнения из одного объекта `std::thread` в только что созданный объект `std::thread`.

Объявление

```
thread(thread&& other) noexcept;
```

Результат

Создает экземпляр `std::thread`. Если до вызова конструктора у `other` имелся связанный с ним поток выполнения, то теперь этот поток связан с только что созданным объектом `std::thread`. В противном случае только что созданный объект `std::thread` не имеет связанного с ним потока выполнения.

Постусловия

Для только что созданного объекта `std::thread`, `x`, `x.get_id()` равно значению `other.get_id()`, имевшемуся до вызова конструктора. `other.get_id()==id()`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объекты std::thread не являются CopyConstructible, поэтому копирующий конструктор отсутствует и есть только этот перемещающий конструктор.

Деструктор std::thread

Уничтожает объект `std::thread`.

Объявление

```
~thread();
```

Результат

Уничтожает `*this`. Если у `*this` имеется связанный с ним поток выполнения (в результате выполнения `this->joinable()` будет возвращено значение `true`), вызывает `std::terminate()` для прекращения выполнения программы.

Выдает

Ничего не выдает.

Перемещающий оператор присваивания std::thread

Перемещает принадлежность потока выполнения из одного объекта `std::thread` в другой объект `std::thread`.

Объявление

```
thread& operator=(thread&& other) noexcept;
```

Результат

Если до вызова `this->joinable()` возвращает `true`, вызывает `std::terminate()` для прекращения выполнения программы. Если до присваивания у `other` имелся связанный с ним поток выполнения, то теперь этот поток выполнения связан с `*this`. В противном случае у `*this` нет связанного с ним потока выполнения.

Постусловия

До вызова значение `this->get_id()` равно значению `other.get_id()`. `other.get_id()==id()`.

Выдает

Ничего не выдает.

ПРИМЕЧАНИЕ

Объекты std::thread не являются CopyAssignable, поэтому копирующий оператор присваивания отсутствует и имеется только перемещающий оператор присваивания.

Компонентная функция std::thread::swap

Производит обмен принадлежности связанных потоков выполнения между двумя объектами `std::thread`.

Объявление

```
void swap(thread& other) noexcept;
```

Результат

Если до вызова у `other` имелся связанный с ним поток выполнения, то теперь этот поток связан с `*this`. В противном случае у `*this` нет связанного с ним потока выполнения. Если до вызова у `*this` имелся связанный с ним поток выполнения, то теперь этот поток связан с `other`. В ином случае у `other` нет связанного с ним потока выполнения.

Постусловия

До вызова значение `this->get_id()` равно значению `other.get_id()`. До вызова значение `other.get_id()` равно значению `this->get_id()`.

Выдает

Ничего не выдает.

Некомпонентная функция обмена для std::threads

Производит обмен принадлежности связанных потоков выполнения между двумя объектами `std::thread`.

Объявление

```
void swap(thread& lhs,thread& rhs) noexcept;
```

Результат

```
lhs.swap(rhs)
```

Выдает

Ничего не выдает.

Компонентная функция std::thread::joinable

Выполняет запрос, имеется ли у `*this` связанный с ним поток выполнения.

Объявление

```
bool joinable() const noexcept;
```

Возвращает

`true`, если у `*this` имеется связанный с ним поток выполнения, или `false` в противном случае.

Выдает

Ничего не выдает.

Компонентная функция std::thread::join

Ожидает завершения потока выполнения, связанного с `*this`.

Объявление

```
void join();
```

Предусловия

Функция `this->joinable()` возвращает `true`.

Результат

Блокирует текущий поток до завершения потока выполнения, связанного с `*this`.

Постусловия

`this->get_id()==id()`. Поток выполнения, связанный с `*this` до вызова, завершился.

Синхронизация

Завершение потока выполнения, связанного с `*this` до вызова, «происходит до» возвращения из вызова `join()`.

Выдает

Исключение `std::system_error`, если результаты не могут быть достигнуты, или `this->joinable()` возвращает `false`.

Компонентная функция std::thread::detach

Отсоединяет поток выполнения, связанный с `*this` для завершения.

Объявление

```
void detach();
```

Предусловия

Функция `this->joinable()` возвращает `true`.

Результат

Отсоединяет поток выполнения, связанный с `*this`.

Постусловия

`this->get_id()==id()`, `this->joinable()==false`. Поток выполнения, связанный с `*this` до вызова, отсоединяется и больше не связан с объектом `std::thread`.

Выдает

Исключение `std::system_error`, если результаты не могут быть достигнуты, или при вызове `this->joinable()` возвращает `false`.

Компонентная функция std::thread::get_id

Возвращает значение типа `std::thread::id`, которое идентифицирует поток выполнения, связанный с `*this`.

Объявление

```
thread::id get_id() const noexcept;
```

Возвращает

Если у `*this` имеется связанный с ним поток выполнения, возвращает экземпляр `std::thread::id`, идентифицирующий этот поток. В противном случае возвращает изначально созданный объект `std::thread::id`.

Выдает

Ничего не выдает.

Статическая компонентная функция std::thread::hardware_concurrency

Возвращает подсказку о количестве потоков, способных конкурентно запускаться на текущем оборудовании.

Объявление

```
unsigned hardware_concurrency() noexcept;
```

Возвращает

Количество потоков, способных конкурентно запускаться на текущем оборудовании. Это, к примеру, может быть число процессоров, имеющихся в системе. Если такая информация недоступна или не поддается точному определению, эта функция возвращает `0`.

Выдает

Ничего не выдает.

Г.7.2. Пространство имен this_thread

Функции в пространстве имен `std::this_thread` работают с вызывающим потоком.

Некомпонентная функция std::this_thread::get_id

Возвращает значение типа `std::thread::id`, идентифицирующее текущий поток выполнения.

Объявление

```
thread::id get_id() noexcept;
```

Возвращает

Экземпляр `std::thread::id`, идентифицирующий текущий поток.

Выдает

Ничего не выдает.

Некомпонентная функция std::this_thread::yield

Используется для информирования библиотеки о том, что поток, вызвавший функцию, не нуждается в выполнении в точке вызова. Зачастую используется в коротких циклах во избежание потребления излишнего времени центрального процессора.

Объявление

```
void yield() noexcept;
```

Результат

Предоставляет библиотеке возможность спланировать вместо текущего потока что-либо другое.

Выдает

Ничего не выдает.

Некомпонентная функция std::this_thread::sleep_for

Приостанавливает выполнение текущего потока на указанный период времени.

Объявление

```
template<typename Rep,typename Period>
void sleep_for(std::chrono::duration<Rep,Period> const& relative_time);
```

Результат

Блокирует текущий поток до истечения времени, заданного параметром `relative_time`.

ПРИМЕЧАНИЕ

Поток может быть заблокирован на время, дольше указанного. По возможности истекшее время нужно измерять по стабильным часам.

Выдает

Ничего не выдает.

Некомпонентная функция std::this_thread::sleep_until

Приостанавливает выполнение текущего потока до достижения заданного момента времени.

Объявление

```
template<typename Clock,typename Duration>
void sleep_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Результат

Блокирует текущий поток до достижения указанного на часах `Clock` момента времени, заданного параметром `absolute_time`.

ПРИМЕЧАНИЕ

По продолжительности блокировки вызывающего потока не дается никаких гарантий, гарантируется только то, что на момент разблокировки потока функцией Clock::now() будет возвращено время, равное времени, заданному параметром absolute_time, или более позднее.

Выдает

Ничего не выдает.