Райнер Гримм

Параллельное программирование на современном C++

Что должен знать каждый профессионал о параллельном программировании

Райнер Гримм

Параллельное программирование на современном языке C++

Concurrency with Modern C++

What every professional C++ programmer should know about concurrency

Rainer Grimm

Leanpub

Параллельное программирование на современном языке C++

Что каждый профессионал должен знать о параллельном программировании

Райнер Гримм

Москва, 2022

УДК 004.4
ББК 32.973.202
Г84

Гримм Р.

Г84 Параллельное программирование на современном языке C++ / пер. с англ. В. Ю. Винника. – М.: ДМК Пресс, 2022. – 616 с.: ил.

ISBN 978-5-97060-957-6

Книга во всех подробностях освещает параллельное программирование на современном языке C++. Особое внимание уделено опасностям и трудностям параллельного программирования (например, гонке данных и мертвой блокировке) и способам борьбы с ними. Приводятся многочисленные примеры кода, позволяющие читателю легко закрепить теорию на практических примерах.

Издание адресовано читателям, которые хотят освоить параллельное программирование на одном из наиболее распространенных языков.

УДК 004.4
ББК 32.973.202

Copyright Concurrency with Modern C++ published by Rainer Grimm. Copyright @2020 Rainer Grimm

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-5-97060-957-6 (рус.)

© Rainer Grimm, 2020
© Перевод, оформление, издание, ДМК Пресс, 2022

Дизайн обложки разработан с использованием ресурса freepik.com.

Содержание

От издательства 17

Введение 18

КРАТКИЙ ОБЗОР 22

1. Параллельное программирование и современный язык C++ 23

1.1. Стандарты C++ 11 и C++ 14: закладка фундамента 24
  1.1.1. Модели памяти 24
    1.1.1.1. Атомарные переменные 25
  1.1.2. Управление потоками 25
    1.1.2.1. Классы для поддержки потоков 25
    1.1.2.2. Данные в совместном доступе 26
    1.1.2.3. Локальные данные потока 27
    1.1.2.4. Переменные условия 27
    1.1.2.5. Кооперативное прерывание потоков (стандарт C++ 20) 28
    1.1.2.6. Семафоры (стандарт C++ 20) 28
    1.1.2.7. Защёлки и барьеры (стандарт C++ 20) 28
    1.1.2.8. Задания 28
    1.1.2.9. Синхронизированные потоки вывода (стандарт C++ 20) 29
1.2. Стандарт C++ 17. Параллельные алгоритмы в стандартной библиотеке 29
  1.2.1. Политики выполнения 30
  1.2.2. Новые параллельные алгоритмы 30
1.3. Сопрограммы в стандарте C++ 20 30
1.4. Учебные примеры 31
  1.4.1. Вычисление суммы элементов вектора 31
  1.4.2. Потокобезопасное создание объекта-одиночки 31
  1.4.3. Поэтапная оптимизация с использованием инструмента CppMem 31
  1.4.4. Быстрая синхронизация потоков 31
  1.4.5. Вариации на тему фьючерсов 31
  1.4.6. Модификации и обобщения генераторов 32
  1.4.7. Способы управления заданиями 32
1.5. Будущее языка C++ 32
  1.5.1. Исполнители 32
  1.5.2. Расширенные фьючерсы 33
  1.5.3. Транзакционная память 33
  1.5.4. Блоки заданий 33
  1.5.5. Библиотека для векторных вычислений 34
1.6. Шаблоны и эмпирические правила 34

1.6.1. Шаблоны синхронизации 34
1.6.2. Шаблоны параллельной архитектуры 34
1.6.3. Эмпирические правила....35
1.7. Структуры данных....35
1.8. Сложности параллельного программирования 35
1.9. Библиотека для работы со временем....35
1.10. Обзор инструментального средства CppMem....35
1.11. Пояснение некоторых терминов 36

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В ПОДРОБНОСТЯХ....37

2. Модель памяти 38
2.1. Начальное представление о модели памяти 38
2.1.1. Что такое область памяти? 39
2.1.2. Что происходит, когда два потока обращаются к одной области памяти 39
2.2. Модель памяти как контракт 40
2.2.1. Основы....42
2.2.2. Трудности....42
2.3. Атомарные переменные 43
2.3.1. Отличие сильной модели памяти от слабой....44
2.3.1.1. Сильная модель памяти 44
2.3.1.2. Слабая модель памяти 46
2.3.2. Атомарный флаг....47
2.3.2.1. Циклическая блокировка 48
2.3.2.2. Сравнение циклической блокировки с мьютексом 50
2.3.2.3. Синхронизация потоков....53
2.3.3. Шаблон `std::atomic`....54
2.3.3.1. Фундаментальный атомарный интерфейс 55
2.3.3.2. Атомарные типы с плавающей точкой в стандарте C++ 20....66
2.3.3.3. Атомарный тип указателя 67
2.3.3.4. Атомарные целочисленные типы 67
2.3.3.5. Псевдонимы типов....70
2.3.4. Функции-члены атомарных типов 71
2.3.5. Свободные функции над атомарными типами 73
2.3.5.1. Особенности типа `std::shared_ptr` (до стандарта C++ 20)....74
2.3.6. Шаблон класса `std::atomic_ref` в стандарте C++ 20 76
2.3.6.1. Мотивация....76
2.3.6.2. Специализации шаблона `std::atomic_ref` 80
2.3.6.3. Полный список атомарных операций....82
2.4. Синхронизация и порядок доступа к памяти....83
2.4.1. Шесть вариантов модели памяти в языке C++ 83
2.4.1.1. Виды атомарных операций....84
2.4.1.2. Ограничения на синхронизацию и порядок доступа 85
2.4.2. Последовательно-согласованное выполнение....86

2.4.3. Семантика захвата и освобождения 88
2.4.3.1. Транзитивность 90
2.4.3.2. Типичное недоразумение 93
2.4.3.3. Последовательность освобождений 97
2.4.4. Модель памяти std::memory_order_consume....99
2.4.4.1. Порядок захвата и освобождения 100
2.4.4.2. Порядок освобождения и потребления....101
2.4.4.3. Различие порядков «освобождение-захват» и «освобождение-потребление»....102
2.4.4.4. Зависимости данных в модели std::memory_order_consume 102
2.4.5. Ослабленная семантика....104
2.4.5.1. Отсутствие ограничений на синхронизацию и порядок операций....104
2.5. Барьеры....106
2.5.1. Барьер std::atomic_thread_fence 106
2.5.1.1. Что такое барьеры памяти 106
2.5.1.2. Три барьера 107
2.5.1.3. Барьеры захвата и освобождения 109
2.5.1.4. Синхронизация с использованием атомарных переменных и барьеров....111
2.5.2. Барьер std::atomic_signal_fence 116

3. Управление потоками 117
3.1. Базовые потоки: класс std::thread 117
3.1.1. Создание потока 118
3.1.2. Время жизни потоков....119
3.1.2.1. Функции join и detach....120
3.1.3. Передача аргументов при создании потока 122
3.1.3.1. Передача по значению и по ссылке....122
3.1.4. Перечень функций-членов....125
3.2. Усовершенствованные потоки: класс std::jthread (стандарт C++ 20)....129
3.2.1. Автоматическое присоединение к потоку....129
3.2.2. Прерывание по запросу в классе std::jthread....131
3.3. Данные в совместном доступе 133
3.3.1. Мьютексы....134
3.3.1.1. Затруднения с мьютексами....138
3.3.2. Блокировщики 141
3.3.2.1. Тип std::lock_guard....141
3.3.2.2. Тип std::scoped_lock 142
3.3.2.3. Тип std::unique_lock 143
3.3.2.4. Блокировщик std::shared_lock 144
3.3.3. Функция std::lock 148
3.3.4. Потокобезопасная инициализация 151
3.3.4.1. Константные выражения....151
3.3.4.2. Функция std::call_once и флаг std::once_flag 152
3.3.4.3. Локальные статические переменные....156
3.4. Данные с потоковой длительностью хранения....157

3.5. Переменные условия 160
3.5.1. Использование предиката в функции ожидания 163
3.5.2. Утерянные и ложные пробуждения 164
3.5.3. Процедура ожидания 165
3.6. Кооперативное прерывание потоков (стандарт C++ 20) 166
3.6.1. Класс `std::stop_source` 167
3.6.2. Класс `std::stop_token` 168
3.6.3. Класс `std::stop_callback` 169
3.6.4. Общий механизм посылки сигналов 172
3.6.5. Особенности класса `std::jthread` 175
3.6.6. Новые перегрузки функции `wait` в классе `std::condition_variable_any` ... 175
3.7. Семафоры (стандарт C++ 20) 178
3.8. Защёлки и барьеры (стандарт C++ 20) 182
3.8.1. Класс `std::latch` 182
3.8.2. Класс `std::barrier` 187
3.9. Асинхронные задания 190
3.9.1. Отличие заданий от потоков 191
3.9.2. Функция `std::async` 192
3.9.2.1. Политика запуска 193
3.9.2.2. Запустить и забыть 195
3.9.2.3. Параллельное вычисление скалярного произведения 196
3.9.3. Тип `std::packaged_task` 198
3.9.4. Типы `std::promise` и `std::future` 203
3.9.4.1. Тип `std::promise` 205
3.9.4.2. Тип `std::future` 205
3.9.5. Тип `std::shared_future` 207
3.9.6. Обработка исключений в асинхронных заданиях 211
3.9.7. Оповещения 214
3.10. Синхронизированные потоки вывода (стандарт C++ 20) 216
3.11. Краткие итоги 223

4. Параллельные алгоритмы в стандартной библиотеке 225
4.1. Политики выполнения 226
4.1.1. Параллельное и векторизованное выполнение 227
4.1.1.1. Код без оптимизации 228
4.1.1.2. Максимальная оптимизация 228
4.1.2. Обработка исключений 228
4.1.3. Опасность гонок данных и мёртвых блокировок 230
4.2. Алгоритмы стандартной библиотеки 231
4.3. Новые параллельные алгоритмы 232
4.3.1. Новые перегрузки 237
4.3.2. Наследие функционального программирования 237
4.4. Поддержка в различных компиляторах 239
4.4.1. Компилятор Microsoft Visual Compiler 239
4.4.2. Компилятор GCC 240
4.4.3. Будущие реализации параллельных стандартных алгоритмов 240

4.5. Вопросы производительности ... 241
4.5.1. Компилятор Microsoft Visual Compiler ... 243
4.5.2. Компилятор GCC ... 244
4.6. Краткие итоги ... 244

5. Сопрограммы в стандарте C++ 20 ... 245
5.1. Функция-генератор ... 247
5.2. Особенности сопрограмм ... 249
5.2.1. Типичные сценарии использования ... 249
5.2.2. Разновидности сопрограмм ... 249
5.2.3. Требования к сопрограммам ... 250
5.2.4. Преобразование функции в сопрограмму ... 250
5.2.4.1. Ограничения ... 251
5.3. Концептуальная модель ... 251
5.3.1. Объект-обещание ... 252
5.3.2. Дескриптор сопрограммы ... 252
5.3.3. Кадр сопрограммы ... 254
5.4. Ожидание отложенного вычисления ... 254
5.4.1. Прообраз ожидания ... 254
5.4.2. Общие требования к контроллерам ожидания ... 255
5.4.3. Стандартные контроллеры ожидания ... 255
5.4.4. Функция `initial_suspend` ... 256
5.4.5. Функция `final_suspend` ... 256
5.4.6. Получение контроллера ожидания ... 257
5.5. Процесс функционирования сопрограммы ... 258
5.5.1. Управление обещанием ... 258
5.5.2. Управление ожиданием ... 259
5.6. Оператор `co_return` и жадный фьючерс ... 261
5.7. Оператор `co_yield` и бесконечный поток данных ... 263
5.8. Оператор `co_await` ... 266
5.8.1. Запуск задания по запросу ... 266
5.9. Синхронизация потоков ... 268
5.10. Краткие итоги ... 273

6. Учебные примеры ... 274
6.1. Вычисление суммы элементов вектора ... 274
6.1.1. Суммирование элементов вектора в одном потоке ... 274
6.1.1.1. Суммирование в цикле по диапазону ... 275
6.1.1.2. Суммирование алгоритмом `std::accumulate` ... 276
6.1.1.3. Использование блокировщика ... 277
6.1.1.4. Использование атомарной переменной ... 278
6.1.1.5. Сводные данные по однопоточным алгоритмам ... 280
6.1.2. Многопоточное суммирование с общей переменной ... 281
6.1.2.1. Использование блокировщика ... 281
6.1.2.2. Использование атомарной переменной ... 283
6.1.2.3. Использование атомарной переменной с функцией `fetch_add` ... 285

6.1.2.4. Использование ослабленной семантики 286
6.1.2.5. Сводные данные по алгоритмам с общей переменной...............287
6.1.3. Раздельное суммирование в потоках 287
6.1.3.1. Использование локальной переменной 287
6.1.3.2. Использование переменных с потоковым временем жизни 292
6.1.3.3. Использование асинхронных заданий 294
6.1.3.4. Сводные данные 296
6.1.4. Суммирование вектора: подведение итогов 297
6.1.4.1. Однопоточные алгоритмы 297
6.1.4.2. Многопоточные алгоритмы с общей переменной.......................297
6.1.4.3. Многопоточные алгоритмы с локальными переменными..........297
6.2. Потокобезопасное создание объекта-одиночки 299
6.2.1. Шаблон «Блокировка с двойной проверкой» 300
6.2.2. Измерение производительности...301
6.2.3. Потокобезопасный вариант реализации Мейерса 304
6.2.4. Реализации на основе блокировщика 305
6.2.5. Реализация на основе функции `std::call_once` 307
6.2.6. Решение на основе атомарных переменных......................................308
6.2.6.1. Семантика последовательной согласованности..........................308
6.2.6.2. Семантика захвата и освобождения...310
6.2.7. Сводные данные 312
6.3. Поэтапная оптимизация с использованием инструмента CppMem 312
6.3.1. Неатомарные переменные..314
6.3.1.1. Анализ программы...315
6.3.2. Анализ программы с блокировкой 320
6.3.3. Атомарные переменные с последовательной согласованностью 321
6.3.3.1. Анализ программы инструментом CppMem................................322
6.3.3.2. Последовательность операций...326
6.3.4. Атомарные переменные с семантикой захвата и освобождения 327
6.3.4.1. Анализ программы инструментом CppMem................................329
6.3.5. Смесь атомарных и неатомарных переменных..................................331
6.3.5.1. Анализ программы инструментом CppMem................................332
6.3.6. Атомарные переменные с ослабленной семантикой..........................333
6.3.6.1. Анализ инструментом CppMem 334
6.3.7. Итоги 335
6.4. Быстрая синхронизация потоков 335
6.4.1. Переменные условия..336
6.4.2. Решение на основе атомарного флага..338
6.4.2.1. Решение с двумя флагами...338
6.4.2.2. Решение с одним атомарным флагом..340
6.4.3. Решение на основе атомарной логической переменной 341
6.4.4. Реализация на семафорах...343
6.4.5. Сравнительный анализ 345
6.5. Вариации на тему фьючерсов 345
6.5.1. Ленивый фьючерс...348
6.5.2. Выполнение сопрограммы в отдельном потоке 351
6.6. Модификации и обобщения генераторов 355

6.6.1. Модификации программы ... 358
6.6.1.1. Если сопрограмму не пробуждать ... 358
6.6.1.2. Сопрограмма не приостанавливается на старте ... 359
6.6.1.3. Сопрограмма не приостанавливается при выдаче значения ... 360
6.6.2. Обобщение ... 361
6.7. Способы управления заданиями ... 364
6.7.1. Функционирование контроллера ожидания ... 364
6.7.2. Автоматическое возобновление работы ... 367
6.7.3. Автоматическое пробуждение сопрограммы в отдельном потоке ... 370
6.8. Краткие итоги ... 373

7. Будущее языка C++ ... 374
7.1 Исполнители ... 374
7.1.1. Долгий путь исполнителя ... 375
7.1.2. Что такое исполнитель ... 376
7.1.2.1. Свойства исполнителя ... 376
7.1.3. Первые примеры ... 377
7.1.3.1. Использование исполнителя ... 377
7.1.3.2. Получение исполнителя ... 378
7.1.4. Цели разработки исполнителей ... 379
7.1.5. Терминология ... 380
7.1.6. Функции выполнения ... 381
7.1.6.1. Единичная кардинальность ... 382
7.1.6.2. Множественная кардинальность ... 382
7.1.6.3. Проверка требований к исполнителю ... 382
7.1.7. Простой пример использования ... 383
7.2. Расширенные фьючерсы ... 386
7.2.1. Техническая спецификация ... 386
7.2.1.1. Обновлённое понятие фьючерса ... 386
7.2.1.2. Средства асинхронного выполнения ... 388
7.2.1.3. Создание новых фьючерсов ... 388
7.2.2. Унифицированные фьючерсы ... 391
7.2.2.1. Недостатки фьючерсов ... 391
7.2.2.2. Пять новых концептов ... 394
7.2.2.3. Направления дальнейшей работы ... 395
7.3. Транзакционная память ... 396
7.3.1. Требования ACI(D) ... 396
7.3.2. Синхронизированные и атомарные блоки ... 397
7.3.2.1. Синхронизированные блоки ... 397
7.3.2.2. Атомарные блоки ... 400
7.3.3. Транзакционно-безопасный и транзакционно-небезопасный код ... 401
7.4. Блоки заданий ... 401
7.4.1. Разветвление и слияние ... 402
7.4.2. Две функции для создания блоков заданий ... 403
7.4.3. Интерфейс ... 404
7.4.4. Планировщик заданий ... 404
7.5. Библиотека для векторных вычислений ... 405

7.5.1. Векторные типы данных 406
7.5.2. Интерфейс векторизированных данных 406
7.5.2.1. Вспомогательные типы-признаки 406
7.5.2.2. Выражения над значениями векторного типа 407
7.5.2.3. Приведение типов 407
7.5.2.4. Алгоритмы над векторизированными значениями 407
7.5.2.5. Свёртка по операции 408
7.5.2.6. Свёртка с маской 408
7.5.2.7. Классы свойств 408
7.6. Итоги 409

8. Шаблоны и эмпирические правила 410
8.1. История понятия 410
8.2. Неоценимая польза шаблонов 412
8.3. Шаблоны или эмпирические правила 413
8.4. Антишаблоны 413
8.5. Итоги 414

9. Шаблоны синхронизации 415
9.1. Управление общим доступом 415
9.1.1. Копирование значения 416
9.1.1.1. Гонка данных при передаче по ссылке 416
9.1.1.2. Проблемы со временем жизни объектов, передаваемых по ссылке 419
9.1.1.3. Материал для дальнейшего изучения 421
9.1.2. Потоковая область хранения 421
9.1.2.1. Материал для дальнейшего изучения 422
9.1.3. Использование фьючерсов 422
9.1.3.1. Материал для дальнейшего изучения 423
9.2. Управление изменяемым состоянием 423
9.2.1. Локальные блокировщики 424
9.2.1.1. Материал для дальнейшего изучения 426
9.2.2. Параметризованные блокировщики 426
9.2.2.1. Шаблон «Стратегия» 426
9.2.2.2. Реализация параметризованных блокировщиков 428
9.2.2.3. Материал для дальнейшего изучения 434
9.2.3. Потокобезопасный интерфейс 434
9.2.3.1. Тонкости потокобезопасных интерфейсов 437
9.2.3.2. Материал для дальнейшего изучения 440
9.2.4. Охраняемая приостановка 440
9.2.4.1. Принцип вталкивания и принцип втягивания 441
9.2.4.2. Ограниченное и неограниченное ожидания 442
9.2.4.3. Оповещение одного или всех ожидающих потоков 443
9.2.4.4. Материал для дальнейшего изучения 446
9.3. Краткие итоги 446

10. Шаблоны параллельной архитектуры 447
10.1. Активный объект 448
10.1.1. Компоненты шаблона 448
10.1.2. Преимущества и недостатки активных объектов 450
10.1.3. Реализация 451
10.1.3.1. Материал для дальнейшего изучения 457
10.2. Объект-монитор 457
10.2.1. Требования 458
10.2.2. Компоненты 458
10.2.3. Принцип действия монитора 459
10.2.3.1. Преимущества и недостатки мониторов 459
10.2.3.2. Реализация монитора 460
10.2.3.3. Материал для дальнейшего изучения 464
10.3. Полусинхронная архитектура 464
10.3.1. Преимущества и недостатки 466
10.3.2. Шаблон «Реактор» 466
10.3.2.1. Требования 466
10.3.2.2. Решение 467
10.3.2.3. Компоненты 467
10.3.2.4. Преимущества и недостатки 468
10.3.3. Проактор 469
10.3.3.1. Требования 469
10.3.3.2. Решение 469
10.3.3.3. Компоненты 470
10.3.3.4. Преимущества и недостатки 471
10.3.4. Материал для дальнейшего изучения 472
10.4. Краткие итоги 472

11. Эмпирические правила 473
11.1. Общие правила 473
11.1.1. Рецензирование кода 473
11.1.2. Сведение к минимуму совместного доступа к изменяемым данным 475
11.1.3. Минимизация ожидания 477
11.1.4. Предпочтительное использование неизменяемых данных 478
11.1.4.1. Пользовательские типы данных и константы этапа компиляции 479
11.1.5. Использование чистых функций 481
11.1.6. Отыскание правильных абстракций 482
11.1.7. Использование статических анализаторов кода 483
11.1.8. Использование динамических анализаторов 483
11.2. Работа с потоками 484
11.2.1. Общие вопросы многопоточного программирования 484
11.2.1.1. Создание как можно меньшего числа потоков 484
11.2.1.2. Использование заданий вместо потоков 487
11.2.1.3. Особая осторожность при отсоединении потока 488

11.2.1.4. Предпочтительность потоков с автоматическим присоединением ... 488
11.2.2. Управление доступом к данным ... 489
11.2.2.1. Передача данных по значению ... 489
11.2.2.2. Использование умного указателя для совместного владения данными ... 489
11.2.2.3. Сокращение времени блокировки ... 492
11.2.2.4. Обёртывание мьютекса в блокировщик ... 493
11.2.2.5. Предпочтительный захват одного мьютекса ... 493
11.2.2.6. Необходимость давать блокировщикам имена ... 494
11.2.2.7. Атомарный захват нескольких мьютексов ... 495
11.2.2.8. Не вызывать неизвестный код под блокировкой ... 496
11.2.3. Переменные условия ... 497
11.2.3.1. Обязательное использование предиката ... 497
11.2.3.2. Замена переменных условия обещаниями и фьючерсами ... 499
11.2.4. Обещания и фьючерсы ... 500
11.2.4.1. Предпочтительность асинхронных заданий ... 500
11.3. Модель памяти ... 500
11.3.1. Недопустимость volatile-переменных для синхронизации ... 501
11.3.1.1. Совет избегать неблокирующего программирования ... 501
11.3.2. Использование шаблонов неблокирующего программирования ... 501
11.3.3. Использование гарантий, предоставляемых языком ... 501
11.3.4. Не нужно изобретать велосипед ... 502
11.3.4.1. Библиотека Boost.Lockfree ... 502
11.3.4.2. Библиотека CDS ... 502
11.4. Краткие итоги ... 503

СТРУКТУРЫ ДАННЫХ ... 504

12. Структуры данных с блокировками ... 505

12.1. Общие соображения ... 505
12.1.1. Стратегии блокировки ... 506
12.1.2. Гранулярность интерфейса ... 508
12.1.3. Типовые сценарии использования ... 510
12.1.3.1. Производительность в ОС Linux ... 516
12.1.3.2. Производительность в ОС Windows ... 517
12.1.4. Избегание прорех ... 517
12.1.5. Конкуренция потоков ... 520
12.1.5.1. Суммирование в один поток без синхронизации ... 520
12.1.5.2. Суммирование в один поток с синхронизацией ... 522
12.1.5.3. Анализ результатов измерений ... 523
12.1.6. Масштабируемость ... 523
12.1.7. Инварианты ... 525
12.1.8. Исключения ... 528
12.2. Потокобезопасный стек ... 528
12.2.1. Упрощённая реализация ... 529

12.2.2. Полная реализация ... 531
12.3. Потокобезопасная очередь ... 535
12.3.1. Блокировка очереди целиком ... 536
12.3.2. Раздельная блокировка концов очереди ... 538
12.3.2.1. Некорректная реализация ... 538
12.3.2.2. Простая реализация очереди ... 539
12.3.2.3. Очередь с фиктивным элементом ... 542
12.3.2.4. Окончательная реализация ... 544
12.3.2.5. Ожидание значения из очереди ... 547
12.4. Краткие итоги ... 550

ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ ... 551

13. Сложности параллельного программирования ... 552
13.1. Проблема ABA ... 552
13.1.1. Наглядное объяснение ... 552
13.1.2. Некритические случаи эффекта ABA ... 553
13.1.3. Неблокирующая структура данных ... 554
13.1.4. Эффект ABA в действии ... 554
13.1.5. Исправление эффекта ABA ... 555
13.1.5.1. Ссылка на помеченное состояние ... 555
13.1.5.2. Сборка мусора ... 555
13.1.5.3. Списки опасных указателей ... 555
13.1.5.4. Механизм чтения-копирования-модификации ... 556
13.2. Тонкости блокировок ... 556
13.3. Нарушение инварианта программы ... 558
13.4. Гонка данных ... 560
13.5. Мёртвые блокировки ... 561
13.6. Неявные связи между данными ... 563
13.7. Проблемы со временем жизни объектов ... 566
13.8. Перемещение потоков ... 567
13.9. Состояние гонки ... 569

14. Библиотека для работы со временем ... 570
14.1. Взаимосвязь моментов, промежутков времени и часов ... 570
14.2. Моменты времени ... 571
14.2.1. Перевод моментов времени в календарный формат ... 572
14.2.2. Выход за пределы допустимого диапазона часов ... 573
14.3. Промежутки времени ... 575
14.3.1. Вычисления с промежутками времени ... 577
14.4. Типы часов ... 579
14.4.1. Точность и монотонность часов ... 579
14.4.2. Нахождение точки отсчёта часов ... 582
14.5. Приостановка и ограниченное ожидание ... 584
14.5.1. Соглашения об именовании ... 584
14.5.2. Стратегии ожидания ... 585

15. Обзор инструментального средства CppMem ... 591
15.1. Упрощённое введение ... 591
15.1.1. Выбор модели ... 592
15.1.2. Выбор программы ... 592
15.1.2.1. Отображаемые отношения ... 593
15.1.2.2. Параметры отображения ... 593
15.1.2.3. Предикаты модели ... 594
15.1.3. Примеры программ ... 594
15.1.3.1. Примеры из статьи ... 594
15.1.3.2. Другие категории примеров ... 595

16. Глоссарий ... 598

Предметный указатель ... 606

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Leanpub очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Введение

«Параллельное программирование на современном языке C++» – это путешествие по нынешним и будущим средствам параллельного программирования в языке C++.

- Стандарты C++ 11 и C++ 14 предоставляют основные строительные блоки для создания многопоточных и асинхронных программ.
- В стандартной библиотеке C++ 17 появились параллельные алгоритмы. Теперь большинство алгоритмов из стандартной библиотеки можно выполнять последовательным, параллельным или векторизированным образом.
- Развитие средств параллельного программирования на этом не останавливается. В стандарт C++ 20 вошли сопрограммы, а в будущем стандарте C++ 23 можно ожидать поддержку транзакционной памяти, расширенные фьючерсы и другие полезные новшества.

В этой книге рассказывается о подробностях параллельного программирования на современном языке C++, а также приводятся многочисленные примеры кода. Поэтому читатель может сочетать теорию с практикой, чтобы от обеих получить максимум знаний.

Поскольку книга посвящена параллельному программированию, в ней хочется рассказать также и о многочисленных подводных камнях – а ещё о том, как их обойти.

Соглашения

Соглашений будет немного.

Выделение шрифтом

Курсив используется, чтобы выделить важную мысль. **Жирный** шрифт служит для выделения ещё более важных мыслей. `Моноширинным` шрифтом набраны фрагменты кода, команды, ключевые слова языка программирования, имена переменных, типов, функций, классов.

Особые символы

Стрелкой $\Rightarrow$ обозначается логическое следование в математическом смысле: выражение $a \Rightarrow b$ означает «если a, то b».

Особые блоки текста

В особые блоки вынесены советы, предупреждения и важная информация.

Совет. В таких блоках помещены советы и дополнительная информация к материалу главы.

Предупреждение. В таком блоке размещены предупреждения, помогающие избегать ошибки.

Краткие итоги. В таких блоках, размещённых в конце основных глав, кратко повторяются основные положения для лучшего запоминания.

Исходный код

Все примеры исходного кода полны. Это означает, что читатель, имея подходящий компилятор, может компилировать и запускать их. Имя исходного файла показано в комментарии в первой строке листинга. Директива `using namespace std` используется иногда в случае крайней необходимости.

Запуск программ

Компиляция и запуск примеров, относящихся к стандартам C++ 11 и C++ 14, происходят проще всего. Любой современный компилятор языка C++ поддерживает эти стандарты. Компиляторам GCC[1] и clang[2] версию стандарта и требование подключить многопоточную библиотеку нужно передавать в параметрах командной строки. Например, чтобы с помощью компилятора g++ из пакета GCC создать исполняемую программу с именем thread, нужно подать такую команду:

```
g++ -std=c++14 -pthread thread.cpp -o thread
```

Назначение параметров таково:

- `-std=c++14` – использовать стандарт C++ 14;
- `-pthread` – подключить библиотеку pthread для поддержки потоков;
- `thread.cpp` – имя исходного файла;
- `-o thread` – имя исполняемого файла, который должен быть построен.

Таким же образом передаются параметры и компилятору clang. Компилятор, входящий в состав среды Microsoft Visual Studio 17, также поддерживает стандарт C++ 14.

Если современного компилятора под рукой нет, существует множество интерактивных сайтов, позволяющих компилировать программы удалённо. Заметка в блоге Арне Мертца содержит прекрасный обзор таких систем[3].

---

1 https://gcc.gnu.org/.

2 https://clang.llvm.org/.

3 https://arne-mertz.de/2017/05/online-compilers/.

Со стандартами C++ 17, 20 и 23 дело обстоит сложнее[1]. Автор инсталлировал систему HPX[2] (High Performance ParallelX – высокопроизводительные параллельные вычисления) – систему с широким набором возможностей для разработки на языке C++ распределённых приложений любого масштаба. В системе HPX уже реализована параллельная стандартная библиотека из стандарта C++ 17, а также многие относящиеся к параллельному программированию нововведения стандартов C++ 20 и 23.

Как читать эту книгу

Читателям, не имеющим сколько-нибудь серьёзных знаний параллельного программирования, рекомендуется начать с начала – с части «Краткий обзор», чтобы составить общее представление о предмете.

Имея общую картину, можно переходить к части «Параллельное программирование в подробностях». Главу о моделях памяти можно пропустить при первом прочтении книги – кроме случаев, когда именно эта тема и нужна читателю. Глава «Учебные примеры» поможет читателю применить изученную теорию на практике. Некоторые примеры довольно сложны и требуют хорошего понимания моделей памяти.

Глава «Будущее языка C++» не обязательна для изучения. С другой стороны, заглядывать в будущее – это так увлекательно!

Последняя часть, озаглавленная «Дополнительные сведения», содержит разбор вопросов, позволяющих лучше понять основной материал книги и извлечь из неё как можно больше пользы.

Личные замечания

Благодарности

Впервые поделившись в своём англоязычном блоге www.ModernesCpp.com намерением написать эту книгу[3], автор встретил гораздо больший отклик, чем мог ожидать. Около пятидесяти человек изъявили желание ознакомиться с предварительным вариантом книги. Автор благодарен всем этим коллегам, включая дочь Юлитте, которая помогла с вёрсткой, и сына Мариуса, ставшего первым, кому довелось вычитывать текст.

[1] На момент выхода этого перевода компилятор MSVC и идущая с ним в комплекте библиотека поддерживают стандарт C++ 20 полностью, а в компиляторах clang и GCC с соответствующими библиотеками не хватает отдельных второстепенных элементов этого стандарта. – *Прим. перев.*

[2] http://stellar.cct.lsu.edu/projects/hpx/.

[3] http://www.modernescpp.com/index.php/looking-for-proofreaders-for-my-new-book-concurrency-with-modern-c.

В алфавитном порядке фамилий: Никос Атанасиу, Роберт Бадеа, Дафидд Вальтерс, Барт Вандевустейн, Анджей Варжинский, Вадим Винник, Джо Дас, Йонас Девлигере, Лассе Натвиг, Эрик Ньютон, Иан Рив, Ранди Хорманн, Энрико Цшемиш.

Автор о себе

Я работал архитектором программного обеспечения, руководителем команды разработчиков и инструктором по программированию на протяжении двух десятилетий. Люблю писать статьи о языках программирования C++, Python, Haskell, а в свободное время – выступать на конференциях. В 2016 году я решил работать на себя и с тех пор занимаюсь организацией и проведением семинаров по современным языкам C++ и Python.

Необычная история возникновения книги

Эту книгу я начинал писать в Оберстдорфе параллельно с заменой тазобедренного сустава. Строго говоря, весь левый тазобедренный сустав заменялся эндопротезом. Первую половину книги я написал, лёжа в клинике и затем проходя реабилитацию. Говоря откровенно, написание книги очень помогло мне в этот сложный период.

Краткий обзор

1. Параллельное программирование и современный язык C++

С выходом стандарта C++ 11 язык получил библиотеку для многопоточного программирования и подходящую модель памяти. Эта библиотека содержит такие строительные блоки, как атомарные переменные, классы для потоков, двоичных семафоров и переменных условия. Они составляют фундамент, на котором в будущих версиях стандарта – например, C++ 20 и C++ 23 – станет возможным определить абстракции более высокого уровня. Однако и в стандарте C++ 11 уже присутствует понятие задания, которое обеспечивает более высокий уровень абстрагирования по сравнению с перечисленными базовыми строительными блоками.

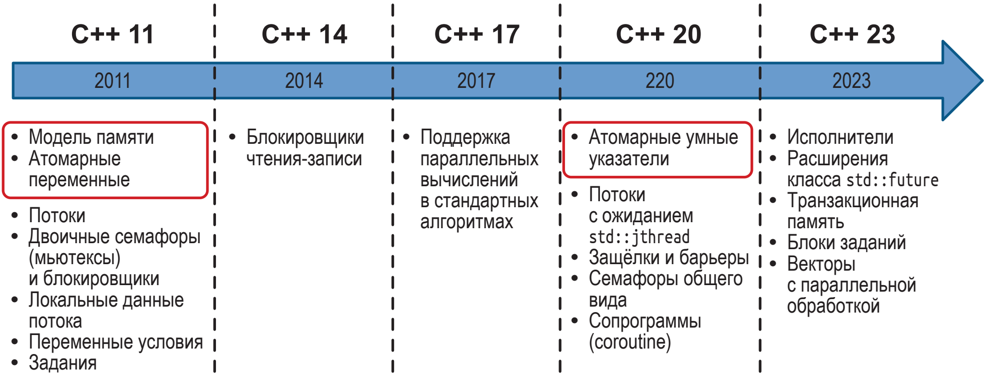
Средства параллельного программирования в языке C++

С некоторым огрублением историю поддержки параллельных вычислений в языке C++ можно разделить на три периода, которые кратко описаны в следующих разделах.

1.1. Стандарты С++ 11 и С++ 14: закладка фундамента

Стандартом С++ 11 поддержка многопоточного программирования впервые добавлена в язык. Она состоит из чётко определённой модели памяти и интерфейса для программирования потоков. С выходом стандарта С++ 14 к этому набору добавлены блокировщики чтения-записи.

1.1.1. Модели памяти



Модели памяти в языке С++

В основе многопоточного программирования лежит чётко определённая *модель памяти*. В ней должны быть отражены следующие аспекты:

- Атомарные операции – операции, выполнение которых не может быть прервано до их полного завершения.
- Частичное упорядочение операций – последовательности операций, порядок выполнения которых не должен нарушаться.
- Видимые эффекты операций – гарантия того, что результат операций над переменными, находящимися в общем доступе, будет заметен из других потоков.

Модель памяти, разработанная для языка С++, испытала заметное влияние предшественника – модели памяти для языка Java. В отличие от последней, однако, язык С++ позволяет снимать требование *временной согласованности* операций, по умолчанию накладываемое на атомарные операции.

Временная согласованность состоит из двух гарантий:

1. Инструкции программы выполняются в том порядке, в котором они записаны в её исходном коде.

2. Существует глобальный порядок выполнения всех операций во всех параллельных потоках.

Понятие модели памяти, в свою очередь, основывается на понятии атомарной операции, выполняемой над данными атомарных типов – для краткости их называют атомарными переменными.

1.1.1.1. Атомарные переменные

Язык C++ содержит набор простых *атомарных типов данных*. К ним относятся логические, символьные, числовые типы и типы указателей, включая различные их вариации. Программист может определить собственный атомарный тип данных, воспользовавшись шаблоном класса `std::atomic`. Этот шаблон позволяет наложить требования синхронизации и упорядочения операций на типы, сами по себе не являющиеся атомарными.

Стандартизированный интерфейс управления потоками – это ядро всей системы средств параллельного программирования на языке C++.

1.1.2. Управление потоками

Средства для работы с потоками в языке C++

Средства многопоточного программирования в языке C++ включают потоки, примитивы синхронизации доступа к общим данным, локальные переменные потоков и задания.

1.1.2.1. Классы для поддержки потоков

В библиотеке языка C++ есть два класса для поддержки потоков: простейший `std::thread`, введённый в стандарте C++ 11, и усовершенствованный `std::jthread`, появившийся в стандарте C++ 20.

1.1.2.1.1. Базовые потоки типа `std::thread`

Класс `std::thread` служит обёрткой для потока – независимой единицы выполнения в составе программы. Выполняемая единица запускается сразу после своего создания, для этого она должна получить на вход вызываемый объект, который и задаёт, что должно быть сделано в потоке. Вызываемый объект может быть именованной функцией, функциональным объектом или лямбда-функцией.

Код, который создаёт поток, отвечает за его дальнейшую судьбу. Единица выполнения, работающая в потоке, завершается с окончанием работы своего вызываемого объекта. Создатель потока может либо присоединить поток, т. е. подождать его завершения посредством вызова `join`, либо отсоединиться от потока, вызвав функцию `detach`. Поток `t` находится в *присоединяемом состоянии*, если над ним не выполнялась ни одна из операций `join` и `detach`. Если в момент вызова деструктора объект-поток находится в присоединяемом состоянии, деструктор вызывает функцию `std::terminate`, которая приводит к аварийному завершению программы.

Поток, отсоединённый от своего создателя, часто называют потоком-демоном, поскольку он выполняется в фоновом режиме.

Под именем `std::thread` скрывается шаблон класса с переменным числом параметров. Это означает, что при создании потока можно передавать любое число аргументов, так как разное число аргументов могут принимать выполняющиеся в потоке вызываемые объекты.

1.1.2.1.2. Усовершенствованные потоки: класс `std::jthread` (стандарт C++ 20)

Название `jthread` означает «присоединяемый» (англ. *joinable*) поток. В дополнение ко всему, что умеет делать класс `std::thread`, введённый в стандарте C++ 11, этот класс ожидает завершения потока в деструкторе и поддерживает кооперативное прерывание. Тем самым класс `std::jthread` расширяет интерфейс класса `std::thread`.

1.1.2.2. Данные в совместном доступе

Программисту необходимо координировать доступ к общей переменной, если более одного потока могут одновременно обращаться к ней и если переменная при этом может менять своё значение (т. е. не является константой). Чтение данных из общей переменной в то время, как другой поток помещает в неё новое значение, называется гонкой данных и представляет собой неопределённое поведение. Координация доступа к общей переменной достигается в языке C++ с помощью мьютексов и блокировщиков.

1.1.2.2.1. Мьютексы

Мьютекс (mutex, от англ. *mutual exclusion* – взаимное исключение) позволяет гарантировать, что только один поток может получить доступ к общей переменной в каждый момент времени. Мьютекс запирает и открывает крити-

ческую секцию, внутри которой происходит работа с общей переменной. В стандартной библиотеке языка C++ определены пять различных видов мьютексов. Они позволяют блокировать выполнение рекурсивно, с запросом состояния блокировки, с ограничением времени ожидания или без такого ограничения. Особый вид мьютекса даёт возможность даже нескольким потокам входить в критическую секцию одновременно.

1.1.2.2.2. Блокировщики

Чтобы гарантировать автоматическое освобождение мьютекса, его следует оборачивать в объект-блокировщик (`lock`). Блокировщики реализуют идиому RAII – время запирания мьютекса ограничивается временем жизни блокировщика. В стандарте языка имеются классы `std::lock_guard` и `std::scoped_lock` для простых сценариев использования и классы `std::unique_lock` и `std::shared_lock` – для более сложных (например, для явного освобождения и повторного запирания мьютекса).

1.1.2.2.3. Потокобезопасная инициализация

Если общие данные используются только для чтения, достаточно обеспечить потокобезопасность их инициализации. Язык C++ предоставляет для этого множество средств, включая константные выражения, статические переменные, видимые в определённом блоке, или функцию `std::call_once` вместе с флагом `std::once_flag`.

1.1.2.3. Локальные данные потока

Объявление переменной как локальной для потока (англ. *thread-local*) означает, что каждый поток получит собственную копию такой переменной. В этом случае переменная уже не будет общей для нескольких потоков. Время жизни локальных данных потока ограничено временем выполнения потока-хозяина.

1.1.2.4. Переменные условия

Переменные условия (англ. *condition variables*) позволяют синхронизировать потоки путём отправки сообщений[1]. Один поток выступает отправителем оповещения, а остальные – получателями. Типичная ситуация, для которой хорошо подходят переменные условия, – это очередь между производителями и потребителями данных. При этом условную переменную может использовать как отправитель, так и получатель сообщения. Использование переменных условия может оказаться непростым делом; зачастую более простых решений можно добиться на основе т. н. заданий.

[1] Можно также сказать, что переменные условия синхронизируют потоки по наступлении некоторого события. – *Прим. перев.*

1.1.2.5. Кооперативное прерывание потоков (стандарт C++ 20)

Полезное дополнение к средствам управления потоками состоит в возможности прерывать их выполнение – при условии, что в коде потока расставлены точки, где его можно прерывать. Такое прерывание называется кооперативным. Кооперативное прерывание поддерживается классами `std::jthread` и `std::condition_variable`, для его реализации служат классы `std::stop_source`, `std::stop_token` и `std::stop_callback`.

1.1.2.6. Семафоры (стандарт C++ 20)

Семафоры представляют собой механизм управления одновременным доступом к общему ресурсу (и в этом отношении отчасти сходны с мьютексами). Семафор снабжён целочисленным счётчиком, который должен быть неотрицательным. Счётчик инициализируется в конструкторе. Каждый захват семафора уменьшает счётчик на единицу, а освобождение – увеличивает. Если поток пытается зайти под семафор (т. е. захватить его), когда счётчик равен нулю, поток блокируется до тех пор, пока какой-то другой поток не освободит семафор, тем самым нарастив счётчик.

1.1.2.7. Защёлки и барьеры (стандарт C++ 20)

Защёлки и барьеры служат для координирования потоков. Они позволяют блокировать поток до тех пор, пока счётчик не достигнет нуля. Начальное значение счётчика задаётся в конструкторе. Несмотря на сходство названий, барьеры, о которых говорится здесь, не имеют ничего общего с барьерами памяти, на которых основана семантика атомарных операций. Для координации потоков через счётчики служат два класса: `std::latch` и `std::barrier`. Одновременный вызов функций-членов объекта такого класса из разных потоков никогда не приводит к гонке данных.

1.1.2.8. Задания

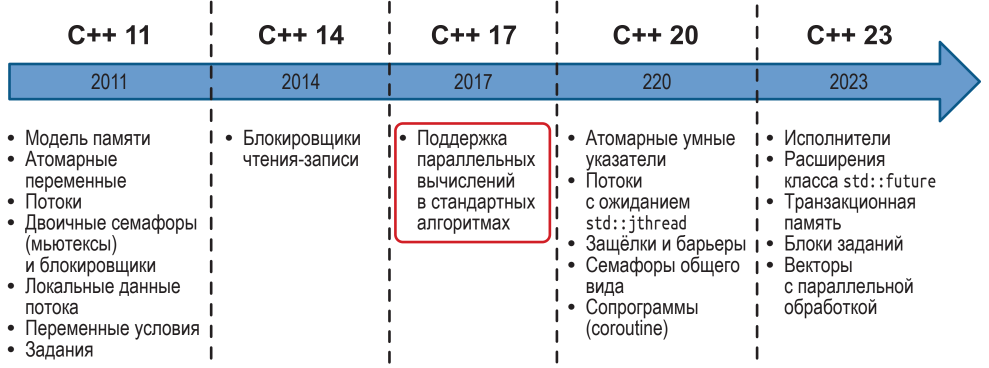
Задания (англ. *tasks*) имеют много общего с потоками. Если поток создаётся и запускается в явном виде, то задание – это нечто, что должно быть рано или поздно выполнено. Реализация стандартной библиотеки C++ сама решит во время выполнения программы, когда задание выполнить, – простым примером может служить функция `std::async`.

Объект-задание похож на канал передачи данных между двумя точками. На основе заданий можно реализовать потокобезопасную коммуникацию между потоками. *Обещание* (англ. *promise*) на одном конце помещает данные в канал, а *фьючерс* на другом конце канала извлекает значение. В роли передаваемых данных могут выступать значения некоторых типов, исключения или просто оповещения о некотором событии. Помимо уже упомянутого `std::async`, стандартная библиотека C++ содержит также шаблоны классов `std::promise` и `std::future`, позволяющие более полно управлять заданиями.

1.1.2.9. Синхронизированные потоки вывода (стандарт C++ 20)

В стандарте C++ 20 появились синхронизированные потоки вывода. Класс `std::basic_syncbuf` представляет собой обёртку над классом `std::basic_streambuf`[1]. Все выводимые символы он накапливает в буфере. Объект-обёртка отправляет накопленные символы в обёрнутый ею поток только в момент своего уничтожения. Следовательно, все сообщения, выведенные в синхронизированный поток, появятся в настоящем потоке вывода в виде единой, цельной последовательности символов. Сообщения из разных потоков при этом перемежаться не могут. Таким образом, из разных потоков можно выводить сообщения в поток `stdout`, не опасаясь путаницы.

1.2. Стандарт C++ 17. Параллельные алгоритмы в стандартной библиотеке



Параллельные алгоритмы в стандарте C++ 17

С появлением стандарта C++ 17 поддержка параллельного программирования языком C++ значительно расширилась, особенно за счёт *параллельных алгоритмов*. Стандарты C++ 11 и C++ 14 содержали лишь простейшие строительные блоки для создания параллельных программ. Эти инструменты были удобны для разработки библиотек или каркасов, но не для разработки приложений. По сравнению со средствами параллельного программирования из стандарта C++ 17, средства стандартов C++ 11 и C++ 14 выглядят словно язык ассемблера.

[1] https://en.cppreference.com/w/cpp/io/basic_streambuf.

1.2.1. Политики выполнения

В стандарте C++ 17 для большинства алгоритмов из стандартной библиотеки стали доступны параллельные версии. Вызывая тот или иной алгоритм, можно передавать ему так называемую *политику выполнения*. Политика определяет, должен ли алгоритм работать последовательно (`std::execution::seq`), параллельно (`std::execution::par`) или параллельно с дополнительной векторизацией (`std::execution::par_unseq`).

1.2.2. Новые параллельные алгоритмы

В дополнение к 69 старым алгоритмам, получившим возможность выполняться параллельным или параллельно-векторизированным способом, библиотека пополнилась восемью новыми алгоритмами. Эти алгоритмы, предназначенные для свёртки, сканирования и преобразования контейнеров, изначально приспособлены для параллельного выполнения.

1.3. Сопрограммы в стандарте C++ 20

Сопрограммы

Сопрограммы можно представить себе как функции, выполнение которых можно приостанавливать, сохраняя текущее состояние, а затем возобновлять. Сопрограммы хорошо подходят для реализации кооперативной многозадачности, как в некоторых операционных системах, циклов обработки событий, бесконечных списков и конвейеров.

1.4. Учебные примеры

После того как модели памяти и интерфейс для управления потоками будут разобраны теоретически, их использование будет продемонстрировано на нескольких учебных примерах.

1.4.1. Вычисление суммы элементов вектора

Сумму элементов вектора можно вычислить различными способами. Это можно делать последовательно или параллельно с большим или меньшим использованием общих данных. Показатели производительности при этом разительно отличаются.

1.4.2. Потокобезопасное создание объекта-одиночки

Потокобезопасное создание объекта-одиночки – классический пример потокобезопасной инициализации переменной в общем доступе. Существует множество способов сделать это, каждый со своей производительностью.

1.4.3. Поэтапная оптимизация с использованием инструмента CppMem

Начав с небольшой программы, будем шаг за шагом её улучшать. Каждый шаг этой непрерывной оптимизации будем проверять на инструменте под названием CppMem. CppMem – это интерактивное инструментальное средство, позволяющее исследовать поведение небольших участков кода с точки зрения моделей памяти, определённых в языке C++.

1.4.4. Быстрая синхронизация потоков

В стандарте C++ 20 имеется много средств для синхронизации потоков. Так, можно пользоваться переменными условия, типами `std::atomic_flag`, `std::atimic<bool>` или семафорами. В отдельной главе изучим показатели производительности этих способов на примере игры в пинг-понг.

1.4.5. Вариации на тему фьючерсов

Благодаря появлению в языке нового ключевого слова `co_return` появится возможность разработать жадный фьючерс, ленивый фьючерс и фьючерс,

работающий в отдельном потоке, – этому посвящена соответствующая глава. Обилие комментариев должно сделать реализацию лёгкой для понимания.

1.4.6. Модификации и обобщения генераторов

Ключевое слово `co_yield` позволяет создавать бесконечные потоки данных. Отдельная глава посвящена тому, как реализацию бесконечного потока сделать конечной и обобщённой.

1.4.7. Способы управления заданиями

В соответствующей главе рассказывается о том, как сделать сопрограмму, способную пробуждаться, когда нужно. В этом поможет ключевое слово `co_await`.

1.5. Будущее языка C++

C++ 11	C++ 14	C++ 17	C++ 20	C++ 23
2011	2014	2017	220	2023
• Модель памяти • Атомарные переменные • Потоки • Двоичные семафоры (мьютексы) и блокировщики • Локальные данные потока • Переменные условия • Задания	• Блокировщики чтения-записи	• Поддержка параллельных вычислений в стандартных алгоритмах	• Атомарные умные указатели • Потоки с ожиданием `std::jthread` • Защёлки и барьеры • Семафоры общего вида • Сопрограммы (coroutine)	• Исполнители • Расширения класса `std::future` • Транзакционная память • Блоки заданий • Векторы с параллельной обработкой

Средства параллельного программирования в стандарте C++ 23

«Очень трудно сделать точный прогноз, особенно о будущем», – говорил Нильс Бор.

1.5.1. Исполнители

Исполнитель (executor) содержит набор правил касательно того, где, когда и как выполнять вызываемый объект. Они представляют собой основные блоки, из которых строится выполнение программы, и определяют, должен ли тот или иной код выполняться в произвольном потоке, в пуле потоков или даже в едином потоке без распараллеливания. От них зависят расшире-

ния фьючерсов, расширения для работы с сетью N4734[1], а также параллельные алгоритмы из стандартной библиотеки; другие средства параллельного программирования из стандартов C++ 20/23, такие как защёлки, барьеры, сопрограммы, транзакционная память и блоки заданий, также смогут использовать исполнители.

1.5.2. Расширенные фьючерсы

Задания, в виде обещаний и фьючерсов появившиеся в стандарте C++ 11, приносят программистам существенную пользу, но у них есть свои недостатки: задания трудно соединять между собой в более крупные единицы. Это ограничение должно исчезнуть благодаря расширениям фьючерсов, появившимся в стандарте C++ 20 и запланированным в стандарте C++ 23. Например, функция-член `then` создаёт фьючерс, который становится готов, когда готово задание-предшественник; функция-член `when_any` – когда готов один из нескольких предшественников, а функция-член `when_all` – когда готовы все предшественники.

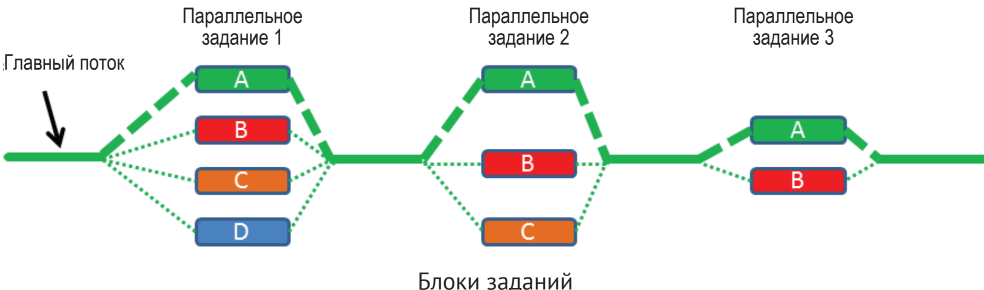
1.5.3. Транзакционная память

Понятие транзакционной памяти основывается на той же идее, что и понятие транзакции в теории баз данных. Транзакция над памятью – это действие, удовлетворяющее первым трем из четырёх требований, предъявляемых к транзакциям в базах данных, известных под названием ACID, а именно требованиям атомарности (atomicity), согласованности (consistency), изолированности (isolation). Требование прочности (durability) имеет смысл для баз данных, но не для разрабатываемой системы транзакционной памяти языка C++. В будущем стандарте ожидаются два вида транзакционной памяти: синхронизированные блоки и атомарные блоки. Оба выполняются с полным упорядочением операций и ведут себя так, будто находятся под глобальной блокировкой. Отличие между ними состоит в том, что в атомарных блоках не может выполняться транзакционно-небезопасный код.

1.5.4. Блоки заданий

Блоки заданий добавляют в язык C++ поддержку идиомы «разветвление–слияние» (fork-join). Следующий рисунок поясняет основную идею блока заданий: на этапе разветвления запускается выполнение ряда заданий, а этап слияния ожидает завершения их всех.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf.



1.5.5. Библиотека для векторных вычислений

Библиотека для векторных вычислений позволяет воспользоваться распараллеливанием по данным (SIMD – англ. *single instructions*, *multiple data*) при работе с векторными типами. Этот принцип вычислений состоит в том, что каждая операция может выполняться параллельно над несколькими значениями.

1.6. Шаблоны и эмпирические правила

Шаблоны – это хорошо документированные методики, лучшие из устоявшихся на практике. Они «выражают отношение между определённым контекстом, проблемой и решением» (Кристофер Александер). Рассматривая трудности параллельного программирования с более фундаментальной точки зрения, можно получить немалую выгоду. В отличие от главы о шаблонах, глава об устоявшихся практиках посвящена более прагматичным советам о том, как преодолевать те или иные затруднения.

1.6.1. Шаблоны синхронизации

Необходимое условие для возникновения гонок данных – наличие общего доступа потоков к изменяемому состоянию. Шаблоны синхронизации сводятся к двум вопросам: что делать с общим доступом и что делать с изменяемым состоянием.

1.6.2. Шаблоны параллельной архитектуры

В главе о параллельных архитектурах представлены три шаблона. Шаблоны «активный объект» и «монитор» основаны на синхронизации и планировании вызовов функций-членов. Третий шаблон, «полусинхронная архитектура» (англ. *Half-Sync/Half-Async*), более сосредоточен на архитектуре системы и позволяет разделить в ней асинхронную и синхронную обработки.

1.6.3. Эмпирические правила

Параллельное программирование сложно по самой своей сути, поэтому имеет смысл пользоваться накопленным практическим опытом – как в области параллельного программирования в целом, так и, в частности, в том, что касается управления потоками и моделей памяти.

1.7. Структуры данных

Структура данных, которая защищает себя сама таким образом, что гонка данных в ней становится невозможной, называется потокобезопасной. Отдельная глава посвящена трудностям, возникающим при разработке потокобезопасных структур данных с блокировками.

1.8. Сложности параллельного программирования

Создание параллельных программ – сложное дело. Особенно сложным оно становится, если использовать только лишь средства из стандартов C++ 11 и C++ 14. Поэтому в отдельной главе рассказывается о наиболее существенных трудностях. Посвятив специальную главу разбору трудностей параллельного программирования, автор надеется, что читатель будет заранее знать, где подстерегает опасность. В частности, речь в главе пойдёт о состоянии гонок, гонке данных и о мёртвых блокировках (англ. *deadlock*).

1.9. Библиотека для работы со временем

Средства для работы со временем тесно связаны со средствами параллельного программирования. Часто возникает необходимость приостановить выполнение потока на определённый промежуток времени или до наступления определённого момента времени. В состав стандартной библиотеки входят: моменты времени, временные интервалы и часы.

1.10. Обзор инструментального средства CppMem

CppMem – это интерактивный инструмент, позволяющий заглянуть глубоко внутрь модели памяти. Он предоставляет две замечательные услуги. Во-

первых, с его помощью можно верифицировать свой безблокировочный код; во-вторых, можно проанализировать безблокировочный код и получить более точное представление о том, как он работает. В этой книге инструмент CppMem используется часто. Поскольку параметры конфигурации и механизмы работы CppMem довольно сложны, глава даёт лишь общее представление об этом инструменте.

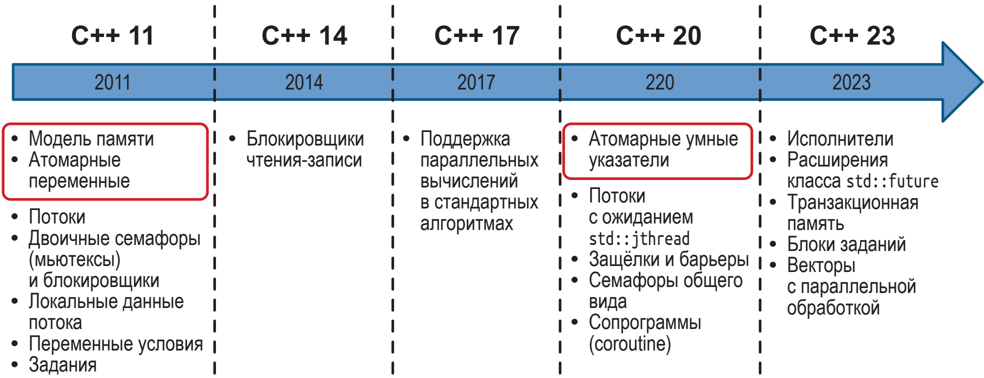
1.11. Пояснение некоторых терминов

Последняя глава книги представляет собой глоссарий важнейших терминов.

Параллельное программирование в подробностях

2. Модель памяти

Основу параллельного программирования составляет *чётко определённая* модель памяти. Читателю она откроется с двух сторон. Во-первых, своей невероятной сложностью и зачастую разительным несоответствием интуитивным представлениям. Во-вторых, тем, как сильно она поможет в глубинном понимании сложностей параллельного программирования.



Модели памяти в языке C++

Разберёмся в первую очередь, что же такое модель памяти.

2.1. Начальное представление о модели памяти

С точки зрения параллельного программирования, модель памяти определяется ответами на два вопроса:

- Что такое область памяти?
- Что происходит, когда два потока одновременно осуществляют доступ к одной области памяти?

Разберём оба этих вопроса в следующих разделах.

2.1.1. Что такое область памяти?

Согласно сайту cppreference.com[1], область памяти – это

- объект скалярного типа (арифметический тип, указатель, перечисление или `std::nullptr_t`),
- наибольшая непрерывная последовательность битовых полей ненулевой длины.

Рассмотрим пример:

```
struct S {
    char a;        // область 1
    int b : 5;     // область 2
    int c : 11,    // область 2 (продолжение)
          : 0,
        d : 8;     // область 3
    int e;         // область 4
    double f;      // область 5
    std::string g; // несколько областей
} obj;
```

В первую очередь бросается в глаза, что объект `obj` состоит из семи подобъектов, однако два битовых поля, `b` и `c`, делят между собой одну область памяти.

Отметим ряд важных наблюдений:

- Каждая переменная есть объект.
- Объект скалярного типа занимает одну область памяти.
- Подряд расположенные битовые поля (в данном примере – поля `b` и `c`) занимают одну область памяти.
- Всякая переменная занимает по меньшей мере одну область памяти.

Теперь можно перейти к самому важному вопросу многопоточного программирования.

2.1.2. Что происходит, когда два потока обращаются к одной области памяти

Если два потока одновременно обращаются к одной области памяти (напомним, что подряд расположенные битовые поля также образуют единую область) и хотя бы один из потоков её модифицирует, в программе имеет место гонка данных, за исключением следующих случаев:

1. Область памяти модифицируется атомарной операцией.
2. Доступ одного потока к области памяти завершается до начала доступа другого.

Второй случай особенно интересен потому, что примитивы синхронизации – такие как мьютексы – существуют именно для того, чтобы обеспечи-

[1] http://en.cppreference.com/w/cpp/language/memory_model.

вать отношение «раньше–позже» между операциями. Отношения «раньше–позже» имеют место между атомарными операциями, а также могут быть распространены на операции над неатомарными типами. Правила упорядочения операций определяют подробности того, как отношения «раньше–позже» выстраиваются в каждом конкретном случае, и поэтому составляют существенную часть модели памяти.

Таково первоначальное формальное объяснение понятия «модель памяти». Пора получить наглядное представление о модели памяти. Модель памяти в языке C++ – это контракт.

2.2. Модель памяти как контракт

Контракт, о котором здесь идёт речь, заключается между программистом и вычислительной системой. В состав системы входят: компилятор, генерирующий машинный код; процессор, на котором машинный код выполняется; кроме того, в процессоре имеются разнообразные буферы сверхоперативной памяти, в которые могут помещаться элементы состояния программы для ускорения доступа. Каждый компонент системы старается, насколько возможно, внести свой вклад в ускорение работы программы. Например, компилятор может использовать регистры процессора для часто меняющихся переменных или разворачивать циклы; в процессоре может происходить внеочередное выполнение[1] или предсказание переходов[2]; буферизация позволяет наперёд загружать машинные инструкции и сохранять в сверхбыстрой памяти данные, к которым обращения происходят особенно часто. Результатом этих оптимизаций – если они выполняются правильно – становится хорошо определённый исполняемый код, полностью оптимизированный для определённой аппаратной платформы. Для большей точности следовало бы говорить не об одном контракте, а о детально проработанном наборе контрактов. Это можно сформулировать иначе: чем менее жёсткие ограничения накладываются на программиста, тем больше остаётся возможностей для оптимизации кода и процесса его выполнения.

Эмпирическое правило таково: чем жёстче условия контракта, тем меньше у системы свободы в выборе способов генерации и выполнения кода

[1] Внеочередное выполнение (англ. *out-of-order execution*) – аппаратная оптимизация, при которой машинные инструкции выполняются не в порядке их следования в машинном коде, а в порядке готовности к выполнению. Ожидая из (более медленной, чем процессор) оперативной памяти поступления данных для очередной инструкции, процессор может «забегая наперёд» выполнить инструкцию, операнды для которой уже находятся в регистрах. – *Прим. перев.*

[2] Предсказание перехода (также прогнозирование ветвлений, англ. *branch prediction*) – аппаратная оптимизация, состоящая в упреждающей загрузке в конвейер и выполнении инструкций, которые с достаточно большой вероятностью должны будут выполниться после команды условного перехода. Если после проверки условия оказывается, что предсказание неверно, результат упреждающего выполнения отбрасывается и в конвейер загружается правильный участок машинного кода; если проверка условия подтверждает правильность прогноза, выполнение программы продолжается с уже наперёд вычисленного состояния. – *Прим. перев.*

с высокой степенью оптимизации. Увы, обратной закономерностью воспользоваться трудно. Когда программист пользуется чрезвычайно слабым контрактом с системой (иначе говоря, очень слабой моделью памяти), он оставляет системе самый большой выбор оптимизаций. Однако в результате их получится программа, разобраться в функционировании которой сможет лишь горстка светил мирового уровня, к которым ни автор, ни читатель этой книги не относятся.

Говоря упрощённо, в стандарте C++ 11 определены три уровня контрактов, см. рисунок:

Три уровня контрактов

До появления стандарта C++ 11 существовал только один контракт. Спецификация языка C++ не содержала контрактов для многопоточного программирования или атомарных операций. В системе предполагался только один поток управления, поэтому возможности оптимизировать исполняемый код оказывались крайне ограниченными. Система непременно должна была дать программисту гарантию, что наблюдаемое поведение программы соответствует последовательности инструкций в исходном коде. Конечно, это означало фактическое отсутствие какой-либо модели памяти. Вместо этого существовало понятие точки следования (англ. *sequence point*). Точкой следования называется такая точка в программе, в которой результаты всех предшествующих инструкций должны стать наблюдаемыми. Начало и конец выполнения функции – это точки следования. При вызове функции с двумя или более аргументами старый стандарт языка C++ не давал никаких гарантий относительно того, какой из них будет вычисляться первым, т. е. порядок

вычисления аргументов оставался неопределённым. Это вполне объяснимо: запятая, которой разделяются аргументы, не является точкой следования.

Всё изменилось в стандарте C++ 11. Это первый стандарт языка C++, в котором учитывается возможность параллельного выполнения нескольких потоков. В основу подробно определённого поведения потоков в языке C++ была положена модель памяти, испытавшая значительное влияние модели памяти для языка Java. Однако модель памяти в языке C++ идёт – как всегда – на несколько шагов впереди. Программисту достаточно соблюдать несколько правил при работе с переменными в совместном доступе, чтобы получить программу с хорошо определённым поведением. Поведение программы не определено, если в ней присутствует хотя бы одна гонка данных. Как уже отмечалось выше, программисту нужно иметь в виду опасность гонки данных, если потоки имеют совместный доступ к изменяемым данным.

Проще всего программировать в терминах заданий, заметно сложнее – в терминах потоков и переменных условия. Атомарные переменные и операции – вотчина профессионалов. Это тем более очевидно, чем более ослабляется контракт, налагаемый моделью памяти. В контексте атомарных переменных часто говорят о неблокирующем (англ. *lock-free*) программировании – столь же интересном, сколь и сложном.

В этом разделе речь шла о сильных и слабых гарантиях. Разумеется, последовательная согласованность представляет собой более сильную модель памяти, а ослабленная семантика – более слабую.

2.2.1. Основы

Модель памяти в языке C++ должна охватывать следующие вопросы:

- Атомарные операции: выполнение каких операций гарантированно не может быть прервано до полного завершения.
- Частичное упорядочение операций: какие последовательности выполнения операций не должны нарушаться.
- Видимые результаты операций: каковы гарантии того, что результаты операций над переменными в общем доступе становятся видимыми в других потоках.

В основе контракта лежат операции над атомарными значениями. Эти операции обладают двумя особенностями: они по определению неделимы и не могут быть прерваны; они накладывают на выполнение программы ограничения, связанные с синхронизацией и порядком выполнения. Эти ограничения на порядок и синхронизацию также сохраняются и для операций над неатомарными значениями. В то время как операции над атомарными переменными всегда атомарны, синхронизацию и упорядочение можно подстроить под свои нужды.

2.2.2. Трудности

Чем более ослабляется модель памяти, тем более программисту необходимо концентрировать внимание на следующих обстоятельствах:

- Доступно больше возможностей оптимизации программы.
- Возможное число путей выполнения программы растёт экспоненциально.
- Всё большее мастерство требуется от программиста.
- Поведение программы противоречит интуитивным ожиданиям.
- Программист погружается в микрооптимизацию.

Чтобы заниматься многопоточным программированием, нужно быть мастером. Чтобы иметь дело с атомарными переменными на уровне последовательной согласованности, нужно взойти на более высокую ступень мастерства. Что же можно сказать о семантике захвата и освобождения или об ослабленной семантике? Только одно: нужно подняться ещё выше по лестнице мастерства (или, если угодно, погрузиться ещё глубже в бездну знаний).

Уровни мастерства

Теперь пора погрузиться глубже в модели памяти языка C++ и начать знакомство с безблокировочным программированием. В последующих разделах речь пойдёт об атомарных переменных и операциях. Затем, после знакомства с основами, последуют остальные уровни модели памяти. Отправной точкой послужит вполне простая согласованность временной последовательности операций, затем разберём семантику захвата-освобождения, а завершим уже не столь интуитивно очевидной ослабленной семантикой.

Начнём же знакомство с атомарными операциями.

2.3. Атомарные переменные

Атомарные переменные лежат в основе модели памяти языка C++. По умолчанию к атомарным переменным применяется наиболее сильная модель

памяти. Поэтому имеет смысл начать изучение с особенностей сильной модели памяти.

2.3.1. Отличие сильной модели памяти от слабой

Как читатель помнит из предыдущих разделов, под сильной моделью памяти понимается последовательная согласованность операций, тогда как наиболее слабой моделью является ослабленная семантика.

2.3.1.1. Сильная модель памяти

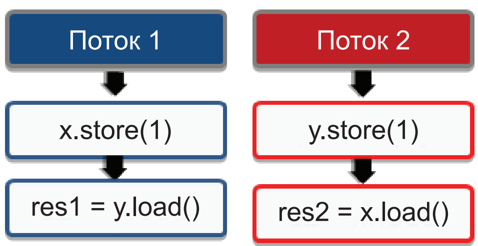
Язык Java 5.0 получил свою нынешнюю модель памяти в 2004 году, а язык C++ – в 2011 году. До этого модель памяти языка Java содержала ошибки, а язык C++ её вообще не имел. Однако было бы в корне неверно думать, что история вопроса начинается с указанных дат. Основы многопоточного программирования закладывались 40 или 50 лет назад. Лесли Лэмпорт в 1979 году дал такое определение последовательной согласованности.

Последовательная согласованность состоит из двух гарантий:

- инструкции программы выполняются в том порядке, в котором они записаны в её тексте;
- существует глобальный порядок операций для всех потоков программы.

Прежде чем погружаться в анализ этих гарантий, необходимо подчеркнуть, что эти гарантии в полной мере относятся к атомарным переменным и в некоторой степени затрагивают неатомарные.

На следующем рисунке показаны два потока. Каждый поток присваивает значение своей атомарной переменной (соответственно *x* и *y*), затем читает значение другой переменной (соответственно *y* и *x*), а затем сохраняет полученное значение в переменные *res1* и *res2*.



Две атомарные переменные

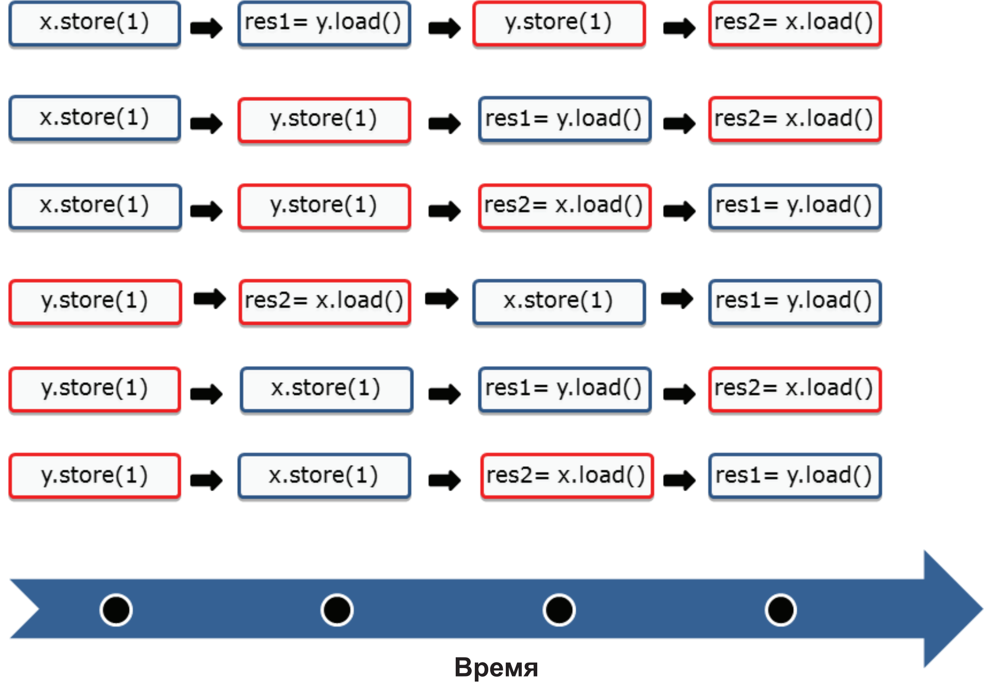
Поскольку переменные *x* и *y* атомарны, операции над ними выполняются атомарным образом, т. е. не могут быть прерваны на середине. По умолчанию действует модель последовательной согласованности. Попытаемся выяснить, в каком порядке должны выполняться эти операции.

Первая гарантия последовательной согласованности означает, что инструкции каждого потока выполняются в том порядке, в котором записаны

в исходном коде. Это понять просто: ни в одном из двух потоков операция `load` не может выполниться раньше, чем операция `store`.

Вторая гарантия, входящая в понятие последовательной согласованности, состоит в том, что инструкции всех потоков вместе взятые должны образовывать глобально упорядоченную последовательность. В данном примере это означает, что поток 2 видит операции, выполняемые потоком 1, в том же порядке, в котором поток 1 их выполняет. Это обстоятельство исключительно важно. Поток 2 видит операции потока 1 в том же порядке, в котором они указаны в исходном коде потока 1. Однако это же верно и для потока 1, который видит операции потока 2. Таким образом, вторую гарантию можно понимать как наличие глобальных часов, ход которых одинаков для всех потоков. Глобальные часы предполагают глобальную упорядоченность операций. Всякий раз, когда глобальные часы продвигаются на один шаг, выполняется одна атомарная операция – хотя невозможно заранее сказать, какая именно.

Головоломка пока не разгадана до конца. Необходимо рассмотреть различные последовательности выполнения операций из двух потоков. Всего существует шесть способов перемежающегося выполнения двух потоков, они показаны на следующем рисунке.



Шесть возможных вариантов перемежающегося выполнения потоков

Теперь всё стало просто. В этом и состоит последовательная согласованность, также известная как сильная модель памяти.

2.3.1.2. Слабая модель памяти

Обратимся ещё раз к понятию контракта между программистом и системой. В этом примере программист использовал атомарные переменные. Тем самым он выполнил свою часть контракта. Система должна гарантировать хорошо определённое поведение программы без гонок данных. Во всём остальном система вольна выполнять четыре операции в любой последовательности. И если программист пользуется ослабленной семантикой, смысл контракта разительно меняется. С одной стороны, программисту становится гораздо труднее понять все возможные способы взаимодействия двух потоков. С другой же – у системы появляется гораздо больше возможностей для оптимизации программы.

В рамках ослабленной семантики, также известной как слабая модель памяти, существует гораздо больше комбинаций, в которые могут складываться эти четыре операции. Так, возможно противоречащее здравому смыслу поведение, когда поток 1 видит операции потока 2 не в том порядке, в котором тот их выполняет, – таким образом, система в целом уже не обладает глобальными часами. Поток 1 вполне может увидеть результат операции `res2= x.load()` раньше, чем результат операции `y.store(1)`. Более того, поток 1 или 2 может и сам выполнять свои операции не в том порядке, в котором они записаны в исходном коде. Например, поток 2 может выполнить операцию `res2= x.load()` раньше, чем операцию `y.store(1)`.

Между последовательной согласованностью и ослабленной семантикой находятся ещё другие модели. Важнейшая из них – семантика захвата-освобождения. Семантика захвата-освобождения предоставляет программисту менее жёсткие гарантии, нежели модель последовательной согласованности. Как следствие у системы появляется больше возможностей для оптимизации. Семантика захвата-освобождения служит ключом к глубокому пониманию синхронизации и частичной упорядоченности операций в параллельном программировании. Потоки синхронизируются в особых точках кода. Без этих точек синхронизации невозможным было бы хорошо определённое поведение потоков или работа переменных условия.

В предыдущем разделе говорилось, что последовательная согласованность – это поведение, которым атомарные переменные обладают по умолчанию. Что это значит? Программист может задать способ упорядочения для каждой отдельно взятой атомарной операции. Если же модель памяти не задать явно, будет применена последовательная согласованность – это означает, что флаг `std::memory_order_seq_cst` неявно применяется к каждой операции над атомарной переменной. Таким образом, фрагмент кода

```
x.store(1);
res = x.load();
```

эквивалентен следующему:

```
x.store(1, std::memory_order_seq_cst);
res = x.load(std::memory_order_seq_cst);
```

В целях простоты всюду в этой книге используется первая форма записи.

Пришло время глубоко изучить атомарные переменные и их место в модели памяти языка C++. Начнём с наиболее элементарного типа – `std::atomic_flag`.

2.3.2. Атомарный флаг

Тип `std::atomic_flag` – это логический тип, все операции над которым атомарны. У него два состояния: установлен и сброшен. Для простоты состояние сброшенного флага будем впредь обозначать `false`, а состояние установленного – `true`. Функция-член `clear` устанавливает значение флага в `false`. Функция-член `test_and_set` устанавливает значение в `true` и возвращает предыдущее значение флага. Поначалу отсутствовала функция, которая бы просто возвращала текущее значение. Это исправлено в стандарте C++ 20. В этом стандарте тип `std::atomic_flag` получает функцию-член `test` и даже может использоваться для синхронизации потоков благодаря появлению функций-членов `notify_one`, `notify_all` и `wait`. В следующей таблице показаны все функции-члены этого класса.

Имя функции	Описание
`atomicFlag.clear()`	Сбросить флаг
`atomicFlag.test_and_set()`	Установить флаг и вернуть его старое состояние
`atomicFlag.test()` (C++20)	Вернуть текущее значение флага
`atomicFlag.notify_one()` (C++20)	Разблокировать одну операцию ожидания
`atomicFlag.notify_all()` (C++20)	Разблокировать все операции ожидания
`atomicFlag.wait(boo)` (C++20)	Заблокировать до прихода оповещения

Вызов `atomicFlag.test()` возвращает текущее состояние флага `atomicFlag`, не изменяя его. Более того, типом `std::atomic_flag` можно пользоваться для синхронизации потоков: для этого служат вызовы `atomicFlag.wait(bool)`, `atomicFlag.notify_one()` и `atomicFlag.notify_all()`. Функции-члены `notify_one` и `notify_all` оповещают, соответственно, один или все потоки, заблокированные на вызове `wait`. Вызов `atomicFlag.wait(boo)` требует аргумента `boo` логического типа. Этот вызов блокирует выполнение потока до тех пор, пока поток не будет разбужен оповещением (`notify_one` или `notify_all`) из другого потока, или до ложного пробуждения. Пробудившись, данная функция-член проверяет, равно ли текущее значение флага `atomicFlag` значению аргумента `boo` и, если не равно, разблокирует выполнение. Таким образом, аргумент `boo` служит защитой от ложных пробуждений.

Помимо этого, в отличие от C++ 11, в стандарте C++ 20 конструктор по умолчанию устанавливает объект `std::atomic_flag` в начальное состояние `false`.

Инициализация атомарных флагов в стандарте C++ 11

Стандарт C++ 11 требовал, чтобы переменные типа `std::atomic_flag` инициализировались следующим образом:

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;
```

Никакие иные способы инициализации, например `std::atomic_flag flag(ATOMIC_FLAG_INIT)`, в стандарте не определены.

Тип `std::atomic_flag` обладает двумя замечательными свойствами:

- это единственный атомарный тип, для которого гарантируется отсутствие блокировок;
- он используется для построения связанных с потоками абстракций более высокого уровня.

Почему говорят, что это единственный тип без блокировок? Остальные, более мощные атомарные типы *могут*, в соответствии со стандартом C++, в своих внутренних механизмах использовать мьютексы. Эти прочие атомарные типы обладают функцией-членом `is_lock_free`, позволяющей проверить, используются ли мьютексы в их внутренней реализации. На всех широко распространённых микропроцессорных архитектурах эти функции возвращают `true`. Программисту следует знать об этом и делать такую проверку, если хочется обеспечить в программе отсутствие блокировок.

Независимость от адресации

Атомарные операции, гарантированно свободные от блокировок, должны быть также независимы от адресации (англ. *address-free*). Это означает, что операции, выполняемые двумя разными процессами[1] над одной областью памяти, остаются атомарными.

Функция `std::is_always_lock_free`

У каждого отдельно взятого объекта `obj` атомарного типа или типа `atomic_ref` (появившегося в стандарте C++ 20) можно спросить, гарантирует ли он отсутствие блокировок: для этого служит вызов `obj.is_lock_free()`. Такая проверка проходит на этапе выполнения. С появлением constexpr-функции `atomic<type>::is_always_lock_free` стало возможным для всякого атомарного типа проверять, гарантирует ли он отсутствие блокировок на всех аппаратных платформах, на которых может запускаться исполняемый модуль. Эта функция даёт результат `true`, только если блокировок заведомо не будет на всех поддерживаемых видах аппаратуры. Проверка отрабатывает на этапе компиляции. Функция доступна начиная со стандарта C++ 17.

Следующее контрольное утверждение никогда не нарушается:

```
if (std::atomic<T>::is_always_lock_free)
  assert(std::atomic<T>().is_lock_free();
```

Интерфейс класса `std::atomic_flag` достаточно мощен, чтобы с его помощью реализовать циклическую блокировку. Циклическая блокировка позволяет защищать критические секции таким же образом, как и мьютекс.

2.3.2.1. Циклическая блокировка

Циклическая блокировка (также называемая спин-блокировкой, спинлоком, англ. *spinlock*) – это примитив синхронизации наподобие мьютекса. В отли-

[1] ...а не только разными потоками в пределах одного процесса. То есть атомарность и последовательная согласованность (если используется сильная модель) должна гарантироваться даже тогда, когда физическое расположение атомарной переменной в памяти по-разному отображено на виртуальные адресные пространства разных процессов. – *Прим. перев.*

чие от мьютекса, однако, он не пассивно ожидает возможности продолжить выполнения. Вместо этого он постоянно продолжает запрашивать право на доступ к критической секции. В случае ожидания циклическая блокировка предотвращает дорогостоящую операцию переключения контекста между пользовательским режимом и режимом ядра, однако при этом она активно использует центральный процессор и впустую нагружает его командами. Если потоки блокируются лишь на очень короткое время, циклические блокировки весьма эффективны. Также часто используются комбинации из циклической блокировки и мьютекса. Поначалу в течение ограниченного времени применяется циклическая блокировка. Если за это время блокировка не освободилась, поток переводится в состояние ожидания.

Никогда не следует пользоваться циклическими блокировками на однопроцессорной системе. В лучшем случае цикл будет расходовать ресурсы и замедлять выполнение своего потока. В худшем случае можно получить мёртвую блокировку (deadlock).

Ниже показана реализация цикла блокировки на основе типа `std::atomic_flag`.

Ожидание на основе циклической блокировки

```
1  // spinLock.cpp
2
3  #include <atomic>
4  #include <thread>
5
6  class Spinlock {
7    std::atomic_flag flag = ATOMIC_FLAG_INIT;
8
9  public:
10   void lock() {
11     while( flag.test_and_set() );
12   }
13
14   void unlock() {
15     flag.clear();
16   }
17 };
18
19 Spinlock spin;
20
21 void workOnResource() {
22   spin.lock();
23   // shared resource
24   spin.unlock();
25 }
26
27 int main() {
28   std::thread t1(workOnResource);
29   std::thread t2(workOnResource);
30
31   t.join();
```

```
32   t2.join();
33 }
```

Два потока, t1 и t2, созданные в строках 28 и 29, соревнуются между собой за доступ к критической секции. Для простоты в этом примере критическая секция состоит лишь из строки 23 с комментарием. Как эта программа работает? Класс Spinlock, подобно мьютексу, обладает функциями-членами lock и unlock. В дополнение к этому переменная типа std::atomic_flag получает начальное значение false (строка 7). Когда поток t1 собирается выполнить функцию workOnResource, возможны следующие два случая:

1. Поток t1 захватывает блокировку, и вызов функции-члена lock сразу завершается. Это может произойти тогда, когда флаг на момент вызова функции lock содержал значение false. В этом случае поток t1 оказывается первым, кто устанавливает значение флага в true. Если теперь поток t2 постарается захватить блокировку, условие в цикле while вернёт значение true. Поток t2 начинает бег по кругу. Он не может сам сбросить флаг в false, поэтому раз за разом проверяет в цикле условие до тех пор, пока поток t1 не выполнит функцию-член unlock и не сбросит флаг – после этого поток t2сможет покинуть цикл и продолжить выполнение.
2. Поток t1 не успевает первым захватить блокировку. Происходит то же самое, что и в предыдущем случае, только потоки t1 и t2 меняются местами.

Читателю стоит обратить внимание на функцию-член test_and_set класса std::atomic_flag. Эта функция выполняет две операции: чтение старого и запись нового значения флага. Однако обе эти операции должны выполняться как одна атомарная операция. Если бы это было не так, два потока осуществляли бы одновременные попытки чтения и записи в общую область памяти. Это по определению является гонкой данных, и программа в целом обладала бы неопределённым поведением.

Было бы интересно сравнить активное ожидание посредством циклической блокировки с пассивным ожиданием, реализованным на основе мьютекса.

2.3.2.2. Сравнение циклической блокировки с мьютексом

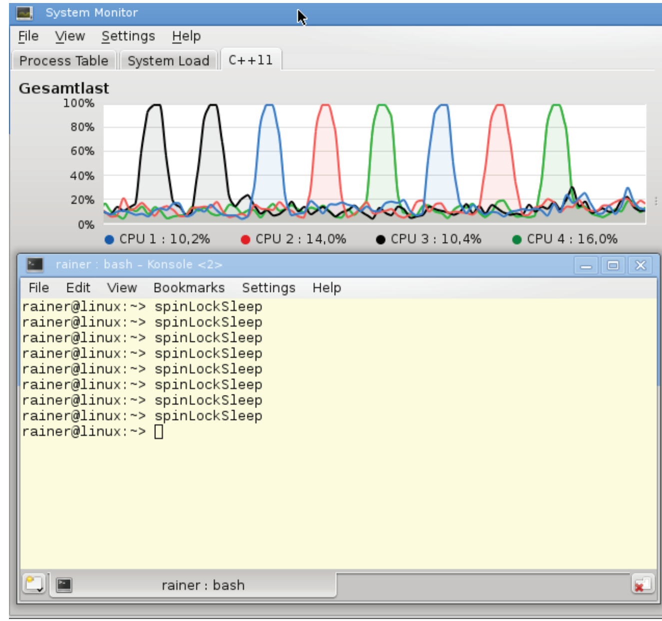
Что произойдёт с нагрузкой на центральный процессор, если функция workOnResource будет захватывать блокировку на 2 секунды?

Ожидание на основе циклической блокировки

```
1  // spinLockSleep.cpp
2
3  #include <atomic>
4  #include <chrono>
5  #include <thread>
6
7  class Spinlock {
8    std::atomic_flag flag = ATOMIC_FLAG_INIT;
9
10 public:
```

```
11   void lock() {
12     while( flag.test_and_set() );
13   }
14
15   void unlock() {
16     flag.clear();
17   }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23   spin.lock();
24   std::this_thread::sleep_for(std::chrono::milliseconds(2000));
25   spin.unlock();
26 }
27
28 int main(){
29   std::thread t1(workOnResource);
30   std::thread t2(workOnResource);
31
32   t.join();
33   t2.join();
34 }
```

Если принцип работы циклической блокировки описан правильно, то одно из процессорных ядер должно быть полностью загружено. Именно это и происходит в действительности, как видно из следующего графика.



Циклическая блокировка на две секунды

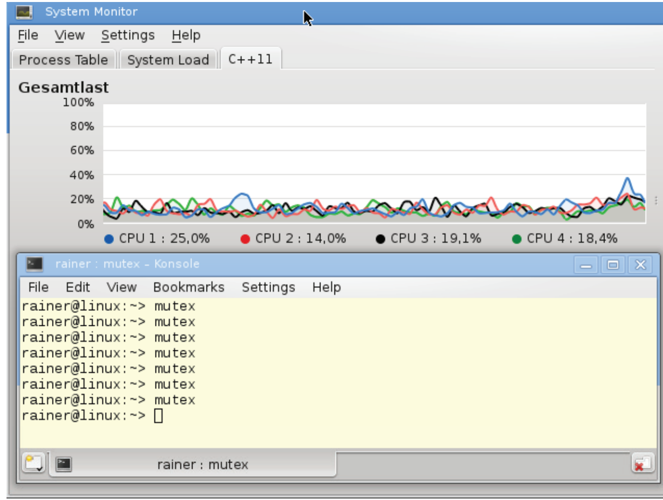
На графике хорошо видно, что при активном ожидании нагрузка на одно из четырёх ядер подскакивает до 100 %. При каждом запуске программы это может оказаться новое ядро.

Теперь вместо циклической блокировки воспользуемся мьютексом и посмотрим, что изменится.

Ожидание на основе мьютекса

```
1  // mutex.cpp
2
3  #include <chrono>
4  #include <mutex>
5  #include <thread>
6
7  std::mutex mut;
8
9  void workOnResource(){
10   mut.lock();
11   std::this_thread::sleep_for(std::chrono::milliseconds(2000));
12   mut.unlock();
13 }
14
15 int main(){
16   std::thread t1(workOnResource);
17   std::thread t2(workOnResource);
18
19   t.join();
20   t2.join();
21 }
```

Сколько бы раз ни запускать эту программу, ни одно из ядер не показывает заметного увеличения нагрузки, о чём свидетельствует следующий график.



Блокировка мьютекса на две секунды

Тем не менее с помощью типа std::atomic_flag синхронизацию потоков можно сделать очевидной и быстрой.

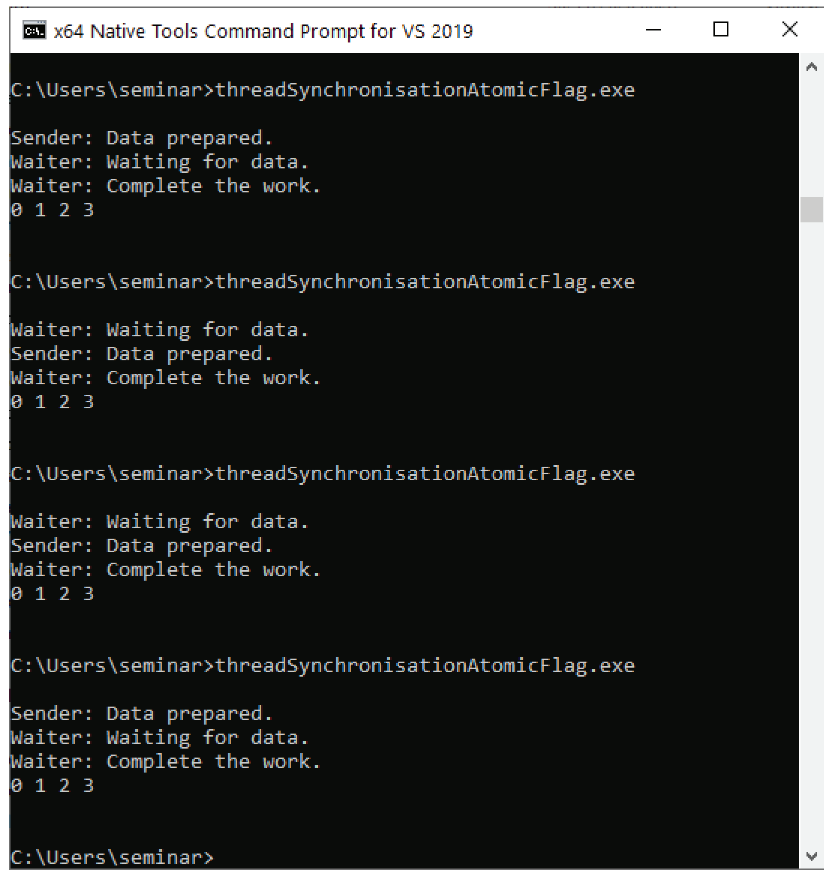
2.3.2.3. Синхронизация потоков

Синхронизация потоков на основе типа std::atomic_flag

```
1  // threadSynchronisationAtomicFlag.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::atomic_flag atomicFlag{};
11
12 void prepareWork() {
13     myVec.insert(myVec.end(), {0, 1, 0, 3});
14     std::cout << "Sender: Data prepared." << '\n';
15     atomicFlag.test_and_set();
16     atomicFlag.notify_one();
17 }
18
19 void completeWork() {
20     std::cout << "Waiter: Waiting for data." << '\n';
21     atomicFlag.wait(false);
22     myVec[2] = 2;
23     std::cout << "Waiter: Complete the work." << '\n';
24     for (auto i: myVec) std::cout << i << " ";
25     std::cout << '\n';
26 }
27
28 int main() {
29     std::cout << '\n';
30
31     std::thread t1(prepareWork);
32     std::thread t2(completeWork);
33
34     t1.join();
35     t2.join();
36
37     std::cout << '\n';
38 }
```

Первый поток готовит исходные данные, а затем устанавливает переменную atomicFlag в значение true (строка 15) и посылает оповещение. Второй поток, который обрабатывает эти данные, ждёт оповещения. Он разблокируется только тогда, когда переменная atomicFlag примет значение true.

Ниже представлен результат нескольких запусков этой программы, откомпилированной с помощью компилятора фирмы Microsoft.



Синхронизация потоков с использованием типа std::atomic_flag

Даже если первый поток отправит своё оповещение до того, как второй поток начнёт его ожидать, оповещение не будет потеряно. Флаги типа std::atomic_flag не подвержены утере пробуждений.

Теперь пора от типа std::atomic_flag как наиболее элементарного механизма перейти к более сложным атомарным типам – к шаблону std::atomic.

2.3.3. Шаблон std::atomic

В библиотеке имеется множество различных вариаций для шаблона класса std::atomic. Так, для типов std::atomic<bool> и std::atomic<пользовательский_тип> используется общее определение шаблона. Частичные специализации доступны для типов указателей std::atomic<T*>, а начиная со стандарта C++ 20 – и для типов умных указателей std::atomic<std::shared_ptr<U>> и std::atomic<std::weak_ptr<U>>. Полные специализации определены для целочисленных типов, а начиная со стандарта C++ 20 – и для действительных типов с плавающей точкой.

Атомарный логический тип, атомарные пользовательские типы и атомарные умные указатели поддерживают один и тот же интерфейс. Будем называть его фундаментальным атомарным интерфейсом. Атомарные указатели расширяют фундаментальный атомарный интерфейс. Ещё более расширен интерфейс атомарных арифметических типов: у атомарных типов с плавающей точкой интерфейс расширен по сравнению с интерфейсом атомарных указателей, а атомарные целочисленные типы обладают интерфейсом, более широким, чем типы с плавающей точкой.

Недостаток различных вариаций шаблона `std::atomic` состоит в том, что они не дают гарантии отсутствия блокировок. В следующих подразделах будут рассмотрены различные атомарные типы, сгруппированные по интерфейсам. Таких подразделов будет четыре. Начнём с атомарных логического, пользовательских типов и типов умных указателей (появившихся в стандарте C++ 20).

2.3.3.1. Фундаментальный атомарный интерфейс

Три специализации шаблона (для типа `bool`, пользовательских типов и типов умных указателей) поддерживают фундаментальный атомарный интерфейс.

Функции-члены	Описание
`is_lock_free`	Проверить отсутствие блокировок
`is_always_lock_free`	Проверить отсутствие блокировок на этапе компиляции
`load`	Атомарным образом вернуть текущее значение атомарной переменной
`operator T`	Эквивалент предыдущей функции
`store`	Атомарным образом присвоить атомарной переменной новое значение (неатомарного типа)
`exchange`	Атомарным образом присвоить атомарной переменной новое значение и вернуть старое
`compare_exchange_strong`, `compare_exchange_weak`	Атомарным образом сравнить и, возможно, обменять значение с неатомарной переменной. Подробно рассматривается ниже
`notify_one` (C++ 20)	Оповестить одну ожидающую операцию
`notify_all` (C++ 20)	Оповестить все ожидающие операции
`wait` (C++ 20)	Заблокировать поток до прихода оповещения. Получив оповещение, сравнить текущее значение со старым для защиты от ложных пробуждений и утери пробуждения. Если текущее значение отличается от старого, функция завершается, иначе продолжает ожидание

Тип `std::atomic<bool>` заслуживает особого рассмотрения.

2.3.3.1.1. Тип `std::atomic<bool>`

Тип `std::atomic<bool>` предоставляет клиентам гораздо больше возможностей, чем тип `std::atomic_flag`. Так, переменным этого типа можно в явном виде присваивать значения `true` и `false`.

Шаблон atomic – это не спецификатор volatile

Что общего между ключевым словом volatile в языках C# и Java и ключевым словом volatile в языке C++? Ничего! Различие между атомарными и volatile-переменными таковы:

- спецификатор volatile предназначен для особых объектов, операции чтения и записи которых не разрешается оптимизировать;
- шаблон std::atomic позволяет объявлять переменные, предназначенные для безопасного чтения и записи из различных потоков.

В этом кроется причина недоразумения. Ключевое слово volatile в языках C++ и Java имеет тот же смысл, что и шаблон std::atomic в языке C++. С другой стороны, слово volatile в языке C++ не имеет отношения к многопоточности.

В языках C и C++ слово volatile обычно используется при программировании встроенных систем для обозначения переменных, которые могут менять свои значения независимо от хода выполнения программы. Одним из примеров может служить переменная, представляющая внешнее устройство (т. е. ввод-вывод с прямым отображением на память). Поскольку значения этих переменных могут изменяться независимо от выполнения программы, операции над ними должны всегда содержать обращение к основной памяти, поэтому использовать буферную память для оптимизации таких операций нельзя.

Типа std::atomic<bool> уже достаточно для синхронизации потоков; на основе этого типа вполне можно реализовать аналог переменной условия.

2.3.3.1.2. Моделирование переменных условия

Покажем сначала, как для синхронизации двух потоков использовать обычную переменную условия.

Использование переменных условия

```
1  // conditionVariable.cpp
2
3  #include <condition_variable>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> mySharedWork;
9  std::mutex mutex_;
10 std::condition_variable condVar;
11
12 bool dataReady{false};
13
14 void waitingForWork() {
15     std::cout << "Waiting " << std::endl;
16     std::unique_lock<std::mutex> lck(mutex_);
17     condVar.wait(lck, []{ return dataReady; });
18     mySharedWork[1] = 2;
19     std::cout << "Work done " << std::endl;
20 }
21
22 void setDataReady() {
```

```
23     mySharedWork = {1, 0, 3};
24     {
25         std::lock_guard<std::mutex> lck(mutex_);
26         dataReady = true;
27     }
28     std::cout << "Data prepared" << std::endl;
29     condVar.notify_one();
30 }
31
32 int main() {
33     std::cout << std::endl;
34
35     std::thread t1(waitingForWork);
36     std::thread t2(setDataReady);
37
38     t1.join();
39     t2.join();
40
41     for (auto v: mySharedWork) {
42         std::cout << v << " ";
43     }
44
45     std::cout << "\n\n";
46 }
```

Здесь опишем работу этой программы лишь в общих чертах. В подробностях о переменных условия рассказывается в соответствующей главе этой книги.

Поток `t1` в строке 17 ждёт оповещения от потока `t2`. Оба потока пользуются одной переменной условия `condVar` и одним и тем же мьютексом `mutex_`. Как происходит взаимодействие между ними?

Поток `t1`:

- подготавливает исходные данные присваиванием `mySharedWork = {1, 0, 3};`
- устанавливает неатомарную логическую переменную `dataReady` в значение `true`;
- посылает оповещение ожидающему потоку с помощью вызова `condVar.notify_one()`.

Поток `t2`:

- ожидает оповещения с помощью вызова `condVar.wait(lck, []{ return dataReady; });`
- когда оповещение пришло и условие `dataReady` истинно, захватывает блокировщик `lck` и продолжает выполнение с оператора `mySharedWork[1] = 2`.

Переменная `dataReady` логического типа, которую поток `t1` устанавливает в значение `true`, а поток `t2` проверяет с помощью лямбда-функции, играет роль памяти для переменной условия, которая сама по себе состоянием не обладает. Переменные условия бывают подвержены двум неприятным явлениям:

1) ложное пробуждение – ситуация, когда ожидающий поток просыпается, хотя никакого оповещения отправитель не присылал;
2) утерянное пробуждение – ситуация, когда отправитель посылает оповещение до того, как поток-получатель переходит в состояние ожидания.

Наличие условия в операции wait позволяет справиться с обеими проблемами.

Теперь посмотрим, как то же самое поведение реализовать с помощью типа `std::atomic<bool>`.

Реализация переменной условия на основе типа `std::atomic<bool>`

```
1  // atomicCondition.cpp
2
3  #include <atomic>
4  #include <chrono>
5  #include <iostream>
6  #include <thread>
7  #include <vector>
8
9  std::vector<int> mySharedWork;
10 std::atomic<bool> dataReady(false);
11
12 void waitingForWork() {
13     std::cout << "Waiting " << std::endl;
14     while (!dataReady.load()) { // C
15         std::this_thread::sleep_for(std::chrono::milliseconds(5));
16     }
17     mySharedWork[1] = 2; // D
18     std::cout << "Work done " << std::endl;
19 }
20
21 void setDataReady(){
22     mySharedWork = {1, 0, 3}; // A
23     dataReady = true; // B
24     std::cout << "Data prepared" << std::endl;
25 }
26
27 int main(){
28     std::cout << std::endl;
29
30     std::thread t1(waitingForWork);
31     std::thread t2(setDataReady);
32
33     t1.join();
34     t2.join();
35
36     for (auto v: mySharedWork) {
37         std::cout << v << " ";
38     }
39
40     std::cout << "\n\n";
41 }
```

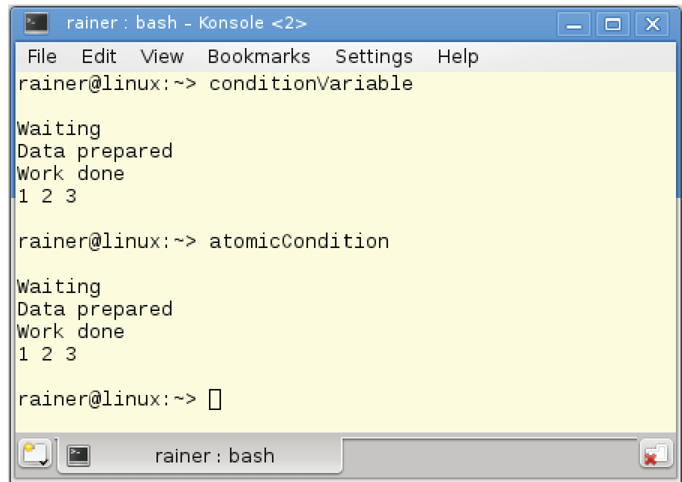
Чем гарантируется, что строка 17 выполнится после строки 14? Или, в более общем плане, что поток `t1` выполнит присваивание `mySharedWork[1] = 2` (строка 17) после того, как поток `t2` выполнит оператор `mySharedWork = {1, 0, 3}` (строка 22). Теперь взаимодействие потоков можно определить более строго:

- строка 22 *предшествует* строке 23;
- строка 14 *предшествует* строке 17;
- строка 23 *синхронизирована* со строкой 14;
- поскольку отношение синхронизации включает в себя предшествование и поскольку отношение предшествования транзитивно, получаем, что оператор `mySharedWork = {1, 0, 3}` выполнится до оператора `mySharedWork[1] = 2`.

Просто, не правда ли? Для простоты в этом рассуждении опущены промежуточные звенья: отношение *синхронизации* влечёт за собой отношение *межпоточного предшествования*, а оно, в свою очередь, влечёт *предшествование*. Интересующийся читатель может найти подробное описание в следующей статье[1].

Следует ещё раз подчеркнуть это исключительно важное обстоятельство: доступ к общей переменной `mySharedWork` синхронизируется посредством переменной условия `condVar` или атомарной переменной `dataReady`. Это так даже несмотря на то, что переменная `mySharedWork` сама по себе не атомарна и не защищена блокировщиком.

Обе представленные программы выдают один и тот же результат, как показано на следующем рисунке.



Синхронизация двух потоков
на основе переменной условия и атомарной переменной

[1] https://en.cppreference.com/w/cpp/atomic/memory_order.

Принцип втягивания и вталкивания

И всё-таки это была не полная правда. Есть одно важное различие между синхронизацией потоков с помощью переменной условия и синхронизацией на основе типа `std::atomic<bool>`. Переменная условия вызовом `condVar.notify()` оповещает ожидающий поток о том, что тот может продолжить свою работу. Во второй программе, напротив, ожидающий поток постоянно проверяет, сделал ли поток-производитель свою работу, т. е. выполнил ли присваивание `dataRead = true`.

Переменная условия оповещает ожидающий поток, реализуя тем самым принцип вталкивания, тогда как реализация на основе атомарной переменной логического типа раз за разом запрашивает значение, пока оно не станет истинным – тем самым реализуя принцип втягивания.

Тип `std::atomic<bool>`, как и все другие полные и частичные специализации шаблона `std::atomic`, поддерживают наиболее важные из всех атомарных операций: операции `compare_exchange_strong` и `compare_exchange_weak`.

2.3.3.1.3. Операции compare_exchange_strong и compare_exchange_weak

Первая операция обладает следующей сигнатурой:

```
bool compare_exchange_strong(T& expected, T& desired)
```

Поскольку эта операция производит сравнение и обмен значений за одну атомарную операцию, в англоязычной литературе её часто называют «compare and swap» (сравнить и поменять местами), или сокращённо CAS. Подобные операции имеются во многих языках программирования и составляют основу неблокирующих алгоритмов. Конечно, поведение подобных операций в разных языках может немного отличаться. Операция, о которой идёт речь здесь, ведёт себя следующим образом:

- если сравнение текущего значения переменной `atomicValue` со значением аргумента `expected` даёт положительный результат, переменной `atomicValue` в рамках этой же атомарной операции присваивается значение аргумента `desired`;
- если же сравнение даёт отрицательный результат, то переменной `expected` присваивается текущее значение переменной `atomicValue`.

Причина, по которой операция названа «сильной» (strong), станет сейчас очевидной. Рядом с ней есть также операция под названием `compare_exchange_weak` (англ. *weak* – слабая). Данная операция может допускать промахи. Это означает, что даже если условие `*atomicValue == expected` выполнено, присваивания переменной `atomicValue` со значения `expected` не происходит, а функция возвращает значение `false`, так что вызов этой операции нужно выполнять в цикле:

```
while (!atomicValue.compare_exchange_weak(expected, desired))
```

Слабая форма существует потому, что некоторые процессоры не поддерживают атомарную инструкцию сравнения и обмена. Поэтому если операцию предполагается использовать в цикле, следует предпочесть слабую форму. На некоторых моделях процессоров она выполняется быстрее.

Для операций семейства CAS характерна так называемая проблема ABA. А именно, если значение переменной прочесть дважды, и оба раза это оказывается одно и то же значение A, можно заключить, будто переменная своего значения не изменяла. Однако может оказаться, что между двумя чтениями она успела изменять своё значение на B и вернуться к прежнему значению.

В следующем подразделе будет показано, сколь просто добиться синхронизации двух потоков с помощью типа `std::atomic<bool>`.

2.3.3.1.4. Синхронизация потоков с помощью типа `std::atomic<bool>`

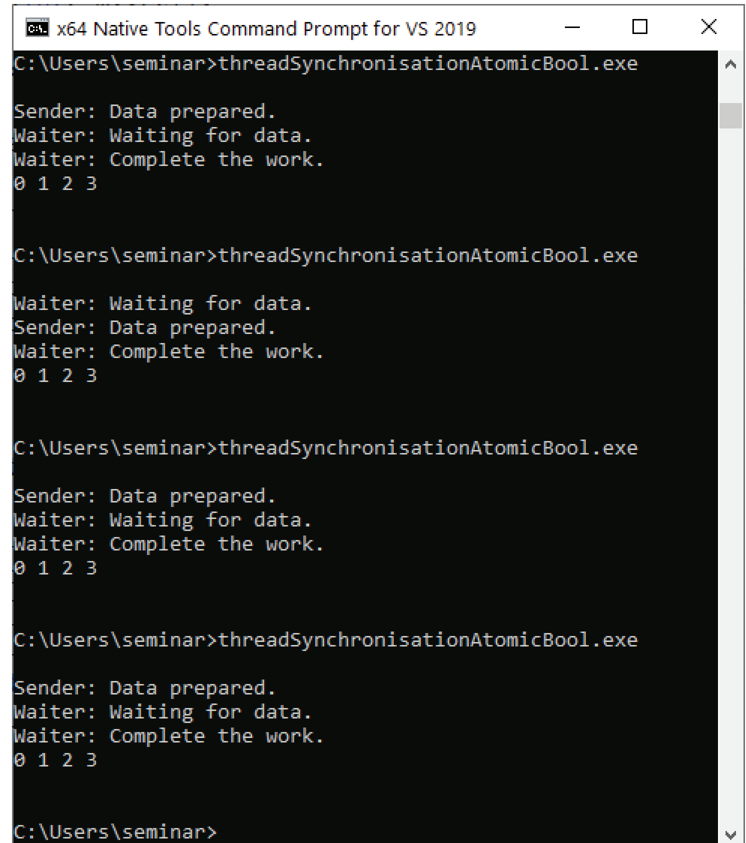
Пример программы с синхронизацией потоков можно легко переписать, используя вместо типа `std::atomic_flag` тип `std::atomic<bool>`.

Синхронизация потоков с помощью типа `std::atomic<bool>`

```
1  // threadSynchronisationAtomicBool.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::atomic<bool> atomicBool{false};
11
12 void prepareWork() {
13     myVec.insert(myVec.end(), {0, 1, 0, 3});
14     std::cout << "Sender: Data prepared." << '\n';
15     atomicBool.store(true);
16     atomicBool.notify_one();
17 }
18
19 void completeWork() {
20     std::cout << "Waiter: Waiting for data." << '\n';
21     atomicBool.wait(false);
22     myVec[2] = 2;
23     std::cout << "Waiter: Complete the work." << '\n';
24     for (auto i: myVec) std::cout << i << " ";
25     std::cout << '\n';
26 }
27
28 int main() {
29     std::cout << '\n';
30
31     std::thread t1(prepareWork);
32     std::thread t2(completeWork);
33
34     t1.join();
35     t2.join();
36
37     std::cout << '\n';
38 }
```

Здесь операция `atomicBool.wait(false)` блокирует выполнение потока до тех пор, пока выполняется условие `atomicBool == false`. Вызов `atomicBool.store(true)` устанавливает этой переменной значение `true`, затем отсылается оповещение, что приводит к разблокировке первого потока.

Так же, как в случае реализации на основе типа `std::atomic_flag`, приведём результаты четырёх запусков программы, собранной компилятором фирмы Microsoft, см. рисунок ниже.



Синхронизация потоков с помощью типа `std::atomic<bool>`

Помимо логического типа, атомарными можно сделать типы указателей, целочисленные и пользовательские типы. Для пользовательских типов действуют особые правила. Однако операции CAS поддерживают все вариации шаблона `std::atomic`.

2.3.3.1.5. Атомарные обёртки для пользовательских типов

Благодаря шаблону `std::atomic` становится возможным делать атомарными типы, созданные программистом. Имеется ряд довольно строгих ограничений, которым должен удовлетворять пользовательский тип, чтобы его мож-

но было подставлять в шаблон std::atomic. Атомарный тип std::atomic<user_defined_type> поддерживает тот же интерфейс, что и тип std::atomic<bool>. Ограничения, накладываемые на атомарные пользовательские типы, таковы:

- копирующие операции присваивания для самого этого типа, всех его базовых классов и всех нестатических членов-данных должны быть тривиальны. Это означает, что в типе нельзя своими руками определять копирующую операцию присваивания, однако о создании такой операции можно попросить компилятор, воспользовавшись ключевым словом default[1];
- пользовательский тип не должен обладать виртуальными функциями-членами или виртуальными базовыми классами;
- пользовательский тип должен допускать побитовое копирование и сравнение, т. е. к нему должны быть применимы функции memcpy[2] и memcmp[3].

На широко распространённых платформах операции над типами вида std::atomic<user_defined_type> могут быть атомарными только тогда, когда размер пользовательского типа не превышает размер типа int.

Проверка свойств типа на этапе компиляции

Свойства пользовательских типов можно проверять на этапе компиляции, используя следующие функции: std::is_trivially_copy_constructible, std::is_polymorphic и std::is_trivial. Эти функции входят в чрезвычайно мощную библиотеку свойств типов[4].

2.3.3.1.6. Атомарные умные указатели в стандарте C++ 20

Объект типа std::shared_ptr содержит в себе указатели на блок служебных данных и на объект-ресурс. Блок служебных данных потокобезопасен, а ресурс – нет. Операции над счётчиком ссылок атомарны, а ресурс уничтожается гарантированно ровно один раз.

Почему так важна потокобезопасность

Стоит сделать небольшое отступление, чтобы подчеркнуть, насколько это важно для типа std::shared_ptr – иметь хорошо определённую семантику в многопоточной среде. На первый взгляд, использование умных указателей типа std::shared_ptr не выглядит разумным выбором для многопоточного программирования. Эти указатели по определению реализуют семантику совместного владения изменяемым объектом данных и потому словно напрашиваются на несинхронизированные операции чтения и записи и, следовательно, на неопределённое поведение. С другой стороны, при программировании на современном языке C++ действует правило: не использовать «сырые» указатели. Поэтому в программах, включая многопоточные, следует пользоваться умными указателями.

---

[1] http://en.cppreference.com/w/cpp/keyword/default.
[2] http://en.cppreference.com/w/cpp/string/byte/memcpy.
[3] http://en.cppreference.com/w/cpp/string/byte/memcmp.
[4] http://en.cppreference.com/w/cpp/header/type_traits.

Предложение N4162[1], касающееся атомарных умных указателей, прямо нацелено на недостатки существующей реализации. Эти недостатки сводятся к трём основным аспектам: согласованности, корректности и производительности.

- **Согласованность**: атомарные операции над типом `std::shared_ptr` – единственные атомарные операции над неатомарным типом, что нарушает единство системы типов.
- **Корректность**: использование глобально-атомарных операций для умных указателей чревато ошибками, так как требует от программиста самодисциплины. Очень просто забыть о необходимости использования атомарной операции – например, написать `ptr = localPtr` вместо `std::atomic_store(&ptr, localPtr)`. Результатом такой ошибки может стать неопределённое поведение по причине гонки данных. Если бы вместо этого использовался тип атомарного умного указателя, система типов сделала бы подобную ошибку невозможной.
- **Производительность**: атомарные умные указатели обладают значительным преимуществом перед функциями семейства `std::atomic_*`. Атомарная версия умных указателей разрабатывается специально для определённой задачи и может, например, иметь в основе своей реализации дешевую циклическую блокировку[2] с типом `std::atomic_flag`. Попытка же сделать неатомарную версию функций над умными указателями потокобезопасной может оказаться напрасным трудом, если эти указатели используются в однопоточной среде. Кроме того, это может привести к потере производительности.

Аргумент, касающийся корректности, вероятно, самый важный. Почему? Ответ дан в документе. В предложении приведена реализация односвязного списка с операциями вставки, удаления и поиска элементов. Причём список реализован без использования блокировок.

2.3.3.1.7. Потокобезопасный односвязный список

Все изменения, которые нужно внести, чтобы программу стало возможно откомпилировать компилятором, поддерживающим только стандарт C++ 11, отмечены красным. Реализация на основе атомарных умных указателей гораздо проще и потому менее подвержена ошибкам. Система типов языка C++ 20 не позволяет применять неатомарные операции к атомарным умным указателям.

В предложении N4162 говорилось о том, чтобы добавить в библиотеку новые типы атомарных умных указателей: `std::atomic_shared_ptr` и `std::atomic_weak_ptr`. При интеграции предложения в действующий международный стандарт языка они превратились в частичные специализации шаблона `std::atomic`, а именно `std::atomic<std::shared_ptr>` и `std::atomic<std::weak_ptr>`.

[1] http://wg21.link/n4162.

[2] https://ru.wikipedia.org/wiki/Спин-блокировка.

```
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
          // in C++11: remove "atomic_" and remember to use the special
          // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
       reference(shared_ptr<Node> p_) : p{p_} { }
       T& operator* () { return p->t; }
       T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load();  // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
       return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;          // in C++11: atomic_load(&head)
      while( !head.compare_exchange_weak(p->next, p) ){ }
      // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
    void pop_front() {
       auto p = head.load();
       while( p && !head.compare_exchange_weak(p, p->next) ){ }
       // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

Потокобезопасный односвязный список

В то же время атомарные операции над типом `std::shared_ptr` в стандарте C++ 20 объявлены устаревшими.

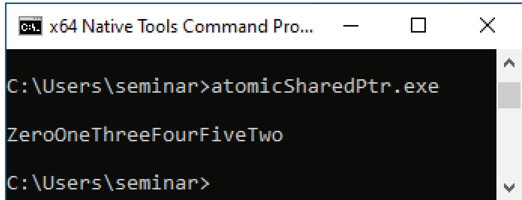
В следующей программе показаны пять потоков, которые одновременно модифицируют переменную типа `std::atomic<std::shared_ptr<std::string>>` без использования явных примитивов синхронизации.

```
1  // atomicSharedPtr.cpp
2
3  #include <iostream>
4  #include <memory>
5  #include <atomic>
6  #include <string>
7  #include <thread>
8
9  int main() {
10     std::cout << '\n';
11
12     std::atomic<std::shared_ptr<std::string>> sharString(
13         std::make_shared<std::string>("Zero"));
14
15     std::thread t1([&sharString]{
16         sharString.store(std::make_shared<std::string>(
```

```
17            *sharString.load() + "One"));
18    });
19    std::thread t2([&sharString]{
20        sharString.store(std::make_shared<std::string>(
21            *sharString.load() + "Two"));
22    });
23    std::thread t3([&sharString]{
24        sharString.store(std::make_shared<std::string>(
25            *sharString.load() +"Three"));
26    });
27    std::thread t4([&sharString]{
28        sharString.store(std::make_shared<std::string>(
29            *sharString.load() +"Four"));
30    });
31    std::thread t5([&sharString]{
32        sharString.store(std::make_shared<std::string>(
33            *sharString.load() +"Five"));
34    });
35
36    t1.join();
37    t2.join();
38    t3.join();
39    t4.join();
40    t5.join();
41    std::cout << *sharString.load() << '\n';
42 }
```

Атомарная переменная `sharString` типа `std::shared_ptr` инициализируется в строке 12 текстом «Zero». Каждый из пяти потоков `t1`–`t5` (строки 15–31) дописывает в конец этой строки свой фрагмент. Если в этой программе подставить тип `std::shared_ptr` вместо типа `std::atomic<std::shared_ptr>`, возникнет гонка данных.

Запуск программы демонстрирует, как перемежается выполнение различных потоков.



Потокобезопасная модификация строки через умный указатель

2.3.3.2. Атомарные типы с плавающей точкой в стандарте C++ 20

Помимо фундаментального атомарного интерфейса, атомарные типы с плавающей точкой поддерживают также сложение и вычитание.

Операции, расширяющие фундаментальный атомарный интерфейс

Функция-член	Описание
fetch_add, +=	Атомарным образом прибавить (соотв. отнять) заданное значение и вернуть старое значение
fetch_sub, -=	

В библиотеке имеются полные специализации шаблона для типов `float`, `double` и `long double`.

2.3.3.3. Атомарный тип указателя

Тип `std::atomic<T*>` – это частичная специализация шаблона `std::atomic`. Она ведёт себя как обычный указатель `T*`. По сравнению с типами с плавающей точкой, этот тип также поддерживает операции пре- и постинкремента и декремента.

Операции, расширяющие интерфейс атомарных типов с плавающей точкой

Функция-член	Описание
++, --	Увеличивает или уменьшает значение атомарной переменной на единицу (с префиксной или постфиксной семантикой)

Рассмотрим небольшой пример.

```
int intArray[5];
std::atomic<int*> p(intArray);
p++;
assert(p.load() == &intArray[1]);
p+=1;
assert(p.load() == &intArray[2]);
--p;
assert(p.load() == &intArray[1]);
```

Кроме рассмотренных, стандарт C++ 11 содержит ещё атомарные обёртки для целочисленных типов.

2.3.3.4. Атомарные целочисленные типы

Для каждого целочисленного типа в библиотеке определена полная специализация шаблона `std::atomic`. Специализации `std::atomic<integral_type>` поддерживают все те же операции, что и типы `std::atomic<T*>` и `std::atomic<floating_point_type>`. Кроме того, они поддерживают ещё побитовые логические операции «И», «ИЛИ» и «исключающее ИЛИ».

Операции над атомарными целочисленными значениями

Функция-член	Описание
fetch_or, |= fetch_and, &= fetch_xor, ^=	Атомарным образом выполняют побитовую операцию над значением атомарной переменной и значением операнда

Есть небольшое различие в поведении между комбинированной операцией с присваиванием и соответствующей функцией из семейства `fetch_`. Комбинированные побитовые операции с присваиванием возвращают новое значение атомарной переменной, тогда как функции `fetch_` – старое.

При внимательном рассмотрении бросается в глаза отсутствие атомарного умножения, атомарного деления, а также атомарных операций побитового сдвига. Это ограничение вряд ли окажется обременительным, так как данные операции бывают нужны нечасто и в случае необходимости их можно реализовать самостоятельно. Ниже представлен пример атомарного умножения.

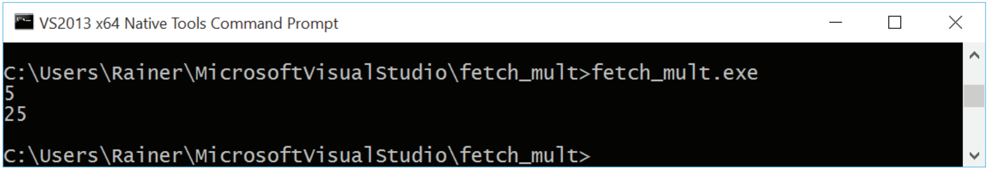
Атомарное умножение с помощью операции `compare_exchange_strong`

```
1  // fetch_mult.cpp
2
3  #include <atomic>
4  #include <iostream>
5
6  template <typename T>
7  T fetch_mult(std::atomic<T>& shared, T mult){
8    T oldValue = shared.load();
9    while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10   return oldValue;
11 }
12
13 int main(){
14   std::atomic<int> myInt{5};
15   std::cout << myInt << std::endl;
16   fetch_mult(myInt,5);
17   std::cout << myInt << std::endl;
18 }
```

Следует обратить внимание, что результат умножения в строке 9 помещается в атомарную переменную только в том случае, когда её текущее значение совпадает со значением переменной `oldValue`. Попытка выполнить умножение помещена в цикл – этим гарантируется, что оно непременно произойдёт[1] –

[1] Напомним, как работает операция `a.compare_exchange_strong(e, d)`. Если текущее значение атомарной переменной `a` совпадает со значением обычной (неатомарной) переменной `e`, то значение `d` помещается в атомарную переменную `a`, а функция возвращает значение `true`. В противном случае, если текущее значение атомарной переменной отличается от ожидаемого, это текущее значение помещается в переменную `e` (для чего та передаётся в функцию по неконстантной ссылке), а функция возвращает значение `false`. Алгоритм атомарного умножения работает следующим образом. Сначала (строка 8) в переменную `oldValue` помещается текущее значение атомарной переменной. К тому времени, как выполнение доходит до строки 9, эту переменную может модифицировать другой поток. Поэтому операция `compare_exchange_strong` проверяет, по-прежнему ли атомарная переменная имеет значение `oldValue`. Если это так, то произведение `oldValue * mult` записывается в атомарную переменную. В противном случае модификации атомарной переменной не происходит, вместо этого в переменную `oldValue` помещается нынешнее значение атомарной переменной и делается новая попытка поместить в неё произведение. Через неизвестное заранее число итераций алгоритму, скорее всего, повезёт – никакой другой поток не успеет модифицировать переменную между итерациями цикла, и алгоритм завершит свою работу. – *Прим. перев.*

ведь над атомарной переменной выполняются две отдельные операции: чтение первоначального значения в строке 8 и последующая попытка модификации в строке 9. Ниже показан результат атомарного умножения.



Атомарное умножение

Атомарное умножение потокобезопасно без блокировок

Алгоритм `fetch_mult` умножит атомарную переменную `shared` на значение `mult`. Важнейшее обстоятельство состоит в том, что между чтением значения атомарной переменной в строке 8 и попыткой присваивания ей нового значения в строке 9 проходит некоторое время, пусть и небольшое. Поэтому другой поток может вклиниться в этот промежуток и изменить значение переменной `shared`. Если теперь представить себе чрезвычайно неудачную очерёдность выполнения потоков, когда другой поток всегда успевает изменить значение атомарной переменной, легко видеть, что алгоритм не гарантирует достижения конечного результата за какое-либо конечное время.
Таким образом, этот алгоритм свободен от блокировок (англ. *lock-free*), но не свободен от ожидания (англ. *wait-free*).

Для каких именно целочисленных типов определены специализации? Ниже приведён подробный список:

- символьные типы: `char`, `char8_t` (C++20), `char16_t`, `char32_t` и `wchar_t`;
- стандартные целочисленные типы со знаком: `signed char`, `short int`, `long` и `long long`;
- стандартные целочисленные типы без знака: `unsigned char`, `unsigned short int`, `unsigned long` и `unsigned long long`;
- дополнительные целочисленные типы, определённые в заголовочном файле `<cstdint>`:
  - `int8_t`, `int16_t`, `int32_t`, `int64_t` (типы со знаком разрядностью ровно 8, 16, 32 и 64 бита соответственно);
  - `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` (беззнаковые типы разрядностью ровно 8, 16, 32 и 64 бита соответственно);
  - `int_fast8_t`, `int_fast16_t`, `int_fast32_t` и `int_fast64_t` (наиболее быстрые типы со знаком разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t` и `uint_fast64_t` (наиболее быстрые типы без знака разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `int_least8_t`, `int_least16_t`, `int_least32_t` и `int_least64_t` (типы со знаком с наименьшей доступной разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `uint_least8_t`, `uint_least16_t`, `uint_least32_t` и `uint_least64_t` (типы без знака с наименьшей доступной разрядностью не менее 8, 16, 32 и 64 бита соответственно);

- intmax_t и uintmax_t (знаковый и беззнаковый целочисленный типы с наибольшей доступной разрядностью);
- intptr_t и uintptr_t (знаковый и беззнаковый целочисленный типы, позволяющие хранить значения указателей).

2.3.3.5. Псевдонимы типов

Для типа std::atomic<bool> и специализаций шаблона std::atomic всеми доступными целочисленными типами в стандартной библиотеке языка C++ определены псевдонимы, как показано в следующей таблице.

Псевдонимы специализаций шаблона std::atomic

Псевдоним	Определение
std::atomic_bool	std::atomic<bool>
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_char8_t (C++20)	std::atomic<char8_t> (C++20)
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>

Псевдоним	Определение
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>
std::atomic_signed_lock_free (C++20)	std::atomic<signed_integral>
std::atomic_unsigned_lock_free (C++20)	std::atomic<unsigned_integral>

Псевдонимы atomic_signed_lock_free и atomic_unsigned_lock_free ссылаются на специализации шаблона std::atomic, соответственно, знаковыми или беззнаковыми целочисленными типами. Эти псевдонимы доступны только для таких целочисленных типов, у которых соответствующие специализации шаблона std::atomic гарантируют отсутствие блокировок. Выбор наиболее эффективной специализации может зависеть от реализации.

2.3.4. Функции-члены атомарных типов

Прежде всего приведём полный список функций-членов из всех атомарных типов.

Функция-член	Описание
test_and_set	Атомарным образом установить значение флага в true и вернуть предыдущее значение
clear	Атомарным образом установить значение флага в false
is_lock_free	Проверить, гарантирует ли объект работу без блокировок
is_always_lock_free	Проверить на этапе компиляции, гарантирует ли тип работу без блокировок
load	Атомарным образом вернуть значение
operator T	Атомарным образом вернуть значение, эквивалентное операции atom.load()
store	Атомарным образом поместить в атомарную переменную новое значение
exchange	Атомарным образом поместить в атомарную переменную новое значение и вернуть её старое значение
compare_exchange_strong compare_exchange_weak	Атомарным образом сравнить значение атомарной переменной с заданным и в зависимости от результата сравнения обменять значение с неатомарной переменной. Подробнее см. раздел 2.3.3.1.3
fetch_add, += fetch_sub, -=	Атомарным образом увеличить (уменьшить) значение атомарной переменной на заданную величину

<table>
<tr><th>Функция-член</th><th>Описание</th></tr>
<tr><td>fetch_or, |=</td><td rowspan="3">Атомарным образом выполнить над атомарной переменной и заданным значением соответствующую побитовую операцию (ИЛИ, И, исключающее ИЛИ)</td></tr>
<tr><td>fetch_and, &=</td></tr>
<tr><td>fetch_xor, ^=</td></tr>
<tr><td>++, --</td><td>Пре- и постинкремент (декремент) атомарной переменной</td></tr>
<tr><td>notify_one (C++20)</td><td>Оповестить один ожидающий поток</td></tr>
<tr><td>notify_all (C++20)</td><td>Оповестить все ожидающие потоки</td></tr>
<tr><td>wait (C++20)</td><td>Заблокировать поток до оповещения. При пробуждении сравнить текущее значение атомарной переменной со старым значением для предотвращения ложных и утерянных пробуждений</td></tr>
</table>

У атомарных типов отсутствует конструктор копирования и копирующая операция присваивания, однако присваивание их значений возможно на основе неявного преобразования к завёрнутому в них обычному типу. Комбинированные операции присваивания возвращают новое значение, тогда как соответствующие им fetch_-функции – старое. Важно отметить, что комбинированные операции присваивания возвращают значение, а не ссылку на объект.

Неявное преобразование к завёрнутому типу:

```
std::atomic<long long> atomObj(2011);
atomObj = 2014;
long long nonAtomObj = atomObj;
```

Каждая функция-член принимает, помимо основного, ещё параметр, отвечающий за упорядочение доступа к памяти. По умолчанию используется значение std::memory_order_seq_cst, однако также можно явно передавать значения std::memory_order_relaxed, std::memory_order_consume, std::memory_order_acquire, std::memory_order_release или std::memory_order_acq_rel. Функции-члены compare_exchange_strong and compare_exchange_weak принимают два параметра, управляющих упорядочением доступа к памяти: один на случай успешного выполнения операции и второй – для неуспешного. Если в явном виде указать лишь один параметр, он будет использован в обоих случаях. Разные порядки доступа будут подробно рассмотрены в разделе 2.4.

Конечно, не каждый атомарный тип поддерживает все эти операции. В следующей таблице показано, какие операции определены для каждой разновидности атомарных типов.

Поддержка операций атомарными типами

Функция-член	atomic_flag	atomic<bool>, atomic<user>, atomic<smart<T>>	atomic <floating>	atomic<T*>	atomic <integral>
test_and_set	да				
clear	да				
is_lock_free		да	да	да	да
is_always_lock_free		да	да	да	да

Функция-член	atomic_flag	atomic<bool>, atomic<user>, atomic<smart<T>>	atomic <floating>	atomic<T*>	atomic <integral>
load		да	да	да	да
operator T		да	да	да	да
store		да	да	да	да
exchange		да	да	да	да
compare_exchange_strong		да	да	да	да
compare_exchange_weak		да	да	да	да
fetch_add, +=			да	да	да
fetch_sub, -=			да	да	да
fetch_or, |=					да
fetch_and, &=					да
fetch_xor, ^=					да
++, --				да	да
notify_one (C++20)	да	да	да	да	да
notify_all (C++20)	да	да	да	да	да
wait (C++20)	да	да	да	да	да

2.3.5. Свободные функции над атомарными типами

К типу std::atomic_flag и типам, получаемым из шаблона std::atomic, можно применять также и ряд свободных (т. е. не являющихся членами класса) функций. Поскольку в этих функциях для передачи аргументов используются указатели, а не ссылки, они совместимы с языком C.

Свободные функции над типом std::atomic_flag – это std::atomic_clear, std::atomic_clear_explicit, std::atomic_flag_test_and_set и std::atomic_flag_test_set_explicit. Первый аргумент у всех этих функций – указатель на объект std::atomic_flag. Помимо того, две функции, имена которых оканчиваются на _explicit, принимают ещё один аргумент, который задаёт упорядочение доступа к памяти.

Для каждой функции-члена каждого типа, получаемого из шаблона std::atomic, существует и соответствующая свободная функция. Имена этих функций строятся по одному и тому же простому шаблону: нужно всего лишь перед именем функции-члена поставить префикс atomic_. Например, функция-член at.store из шаблона std::atomic превращается в свободные функции std::atomic_store и std::atomic_store_explicit. Первая из них принимает один аргумент типа указателя, а вторая – ещё один аргумент, управляющий порядком доступа к памяти. Полный список перегруженных свободных функций над атомарными типами можно найти в справочнике[1].

[1] https://en.cppreference.com/w/cpp/atomic.

За единственным исключением, атомарные свободные функции определены только для атомарных типов. Исключение составляют функции над умными указателями `std::shared_ptr`.

2.3.5.1. Особенности типа std::shared_ptr (до стандарта C++ 20)

Шаблон `std::shared_ptr` представляет собой единственный неатомарный тип данных, для которого определены атомарные функции. Стоит сначала рассказать о причинах, по которым сделано это важное исключение.

Комитет по стандартизации языка C++ счёл необходимым, чтобы умные указатели предоставляли хотя бы минимальные гарантии атомарности при многопоточном программировании. Что подразумевается под минимальными гарантиями атомарности со стороны типов `std::shared_ptr`? Блок служебных данных умного указателя `std::shared_ptr` потокобезопасен. Это означает, что инкремент и декремент счётчика ссылок – атомарные операции. Также гарантируется, что ресурс уничтожается ровно один раз.

Гарантии, предоставляемые типом `std::shared_ptr`, хорошо описаны в документации к библиотеке Boost[1]:

1. Экземпляр `shared_ptr` можно «читать» (т. е. вызывать одни лишь константные операции) одновременно из нескольких потоков.
2. В различные экземпляры `shared_ptr` можно «писать» (т. е. вызывать модифицирующие операции наподобие `operator=` или `reset`) одновременно из нескольких потоков (даже если эти экземпляры представляют собой копии друг друга и имеют под собой общий счётчик ссылок)[2].

Чтобы пояснить смысл этих двух утверждений, стоит привести пример. Снимать указатели `std::shared_ptr` копии и отдавать их различным потокам – вполне нормально.

Потокобезопасное копирование умных указателей std::shared_ptr

```
1  std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3  for (auto i = 0; i < 10; i++) {
4      std::thread([ptr] {
5          std::shared_ptr<int> localPtr(ptr);
6          ptr = std::make_shared<int>(2014);
7        }).detach();
8  }
```

---

[1] http://www.boost.org/doc/libs/1_57_0/libs/smart_ptr/shared_ptr.htm#ThreadSafety.

[2] Может показаться необычным, что для описания свойств умного указателя из стандартной библиотеки C++ привлекается документация по библиотеке Boost. Всё станет на свои места, если вспомнить, что библиотека Boost часто играет роль «испытательного полигона» для различных решений, многие из которых затем включаются в стандарт. Именно это произошло с умными указателями: шаблон `std::shared_ptr` представляет собой почти неизменный `boost::shared_ptr`. – *Прим. перев.*

Посмотрим на строку 5. Вызов конструктора копирования типа `std::shared_ptr` при создании переменной `localPtr` затрагивает лишь блок служебных данных. Это потокобезопасно. Строка 6 несколько интереснее: указателю `ptr` присваивается новое значение. Однако и здесь проблем с точки зрения многопоточности не возникает: лямбда-функция в строке 4 захватывает переменную `ptr` по значению, поэтому модифицирующая операция применяется к копии.

Картина разительно меняется, если захватывать переменную по ссылке.

Гонка данных при работе с типом `std::shared_ptr`

```
1  std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3  for (auto i = 0; i < 10; i++) {
4      std::thread([&ptr] {
5          ptr = std::make_shared<int>(2014);
6        }).detach();
7  }
```

Лямбда-выражение в строке 4 захватывает переменную `ptr` по ссылке. Это означает, что присваивание в строке 5 может привести к одновременному чтению и записи общих данных из нескольких потоков, поэтому программа в целом обладает неопределённым поведением.

Нужно признаться: построить этот последний пример оказалось нелёгким делом. Использование типа `std::shared_ptr` в многопоточной среде требует особого внимания. Тип `std::shared_ptr` – единственный неатомарный тип данных в стандарте языка C++, для которого определены атомарные операции.

2.3.5.1.1. Атомарные операции над типом `std::shared_ptr`

У атомарных операций `load`, `store`, `compare_and_exchange` существуют специализированные версии для типа `std::shared_ptr`. Функции с именами, заканчивающимися на `_explicit`, позволяют дополнительно задавать порядок доступа к памяти. Ниже приведён список всех свободных атомарных функций над типом `std::shared_ptr`.

Атомарные операции над типом `std::shared_ptr`

```
std::atomic_is_lock_free(std::shared_ptr)
std::atomic_load(std::shared_ptr)
std::atomic_load_explicit(std::shared_ptr)
std::atomic_store(std::shared_ptr)
std::atomic_store_explicit(std::shared_ptr)
std::atomic_exchange(std::shared_ptr)
std::atomic_exchange_explicit(std::shared_ptr)
std::atomic_compare_exchange_weak(std::shared_ptr)
std::atomic_compare_exchange_strong(std::shared_ptr)
std::atomic_compare_exchange_weak_explicit(std::shared_ptr)
std::atomic_compare_exchange_strong_explicit(std::shared_ptr)
```

Подробное описание этих функций можно найти в справочнике[1]. С их помощью потокобезопасная модификация умного указателя, переданного по ссылке, оказывается довольно простой.

Устранение гонки данных при работе с типом std::shared_ptr

```
1  std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3  for (auto i = 0; i < 10; i++) {
4      std::thread([&ptr] {
5          auto localPtr= std::make_shared<int>(2014);
6          std::atomic_store(&ptr, localPtr);
7        }).detach();
8  }
```

Модификация переменной `ptr` типа `std::shared_ptr` в выражении `std::atomic_store(&ptr, localPtr)` потокобезопасна. Всё ли теперь хорошо? Нет! Языку C++ нужны настоящие атомарные умные указатели.

Атомарные указатели в стандарте C++ 20

На этом история атомарных умных указателей не заканчивается. В стандарте C++ 20 появились два новых типа умных указателей: `std::atomic<std::shared_ptr>` и `std::atomic<std::weak_ptr>`. О них идёт речь в разделе 2.3.3.1.7. Программисту следует выбирать их всегда, когда это возможно.

Продолжим знакомство со стандартом C++ 20.

2.3.6. Шаблон класса `std::atomic_ref` в стандарте C++ 20

Шаблон класса `std::atomic_ref` позволяет применять атомарные операции через ссылку на объект. Таким образом, параллельно выполняемые над атомарным объектом операции чтения и записи не приводят к гонке данных. Время жизни объекта по ссылке должно превышать время жизни объекта типа `std::atomic_ref`. Доступ к подобъектам объекта, переданного через ссылку `atomic_ref`, однако, не имеет чётко определенного поведения.

2.3.6.1. Мотивация

Здесь следует остановиться и подумать. На первый взгляд может показаться, что использования ссылки на объект атомарного типа было бы достаточно. К сожалению, это не так. В следующем примере объявляется класс `ExpensiveToCopy` с дорогой операцией копирования, в котором есть член `counter` (счётчик). Пусть несколько потоков параллельно наращивают значение счётчика. Тогда переменную-член `counter` нужно защитить от гонки данных.

[1] http://en.cppreference.com/w/cpp/memory/shared_ptr.

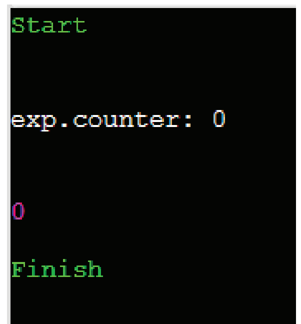
Использование ссылки на атомарный объект

```
1  // atomicReference.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <random>
6  #include <thread>
7  #include <vector>
8
9  struct ExpensiveToCopy {
10     int counter{};
11 };
12
13 int getRandom(int begin, int end) {
14
15     std::random_device seed;        // initial randomness
16     std::mt19937 engine(seed());    // generator
17     std::uniform_int_distribution<> uniformDist(begin, end);
18
19     return uniformDist(engine);
20 }
21
22 void count(ExpensiveToCopy& exp) {
23     std::vector<std::thread> v;
24     std::atomic<int> counter{exp.counter};
25
26     for (int n = 0; n < 10; ++n) {
27         v.emplace_back([&counter] {
28             auto randomNumber = getRandom(100, 200);
29             for (int i = 0; i < randomNumber; ++i) { ++counter; }
30         });
31     }
32
33     for (auto& t : v) t.join();
34 }
35
36 int main() {
37     std::cout << '\n';
38
39     ExpensiveToCopy exp;
40     count(exp);
41     std::cout << "exp.counter: " << exp.counter << '\n';
42
43     std::cout << '\n';
44 }
```

Переменная `exp`, объявленная в строке 39, представляет собой объект, который копировать слишком накладно. Для повышения производительности функция `count` (строка 22) принимает объект `exp` по ссылке. Функция `count` инициализирует переменную типа `std::atomic<int>` из поля `exp.counter` (строка 24). Далее (строка 26) создаются десять потоков, в каждом из которых вы-

полняется лямбда-выражение, принимающее ссылку на переменную `counter`. Это лямбда-выражение берёт случайное число между 100 и 200 (строка 28) и соответствующее число раз увеличивает значение счётчика на единицу. Функция `getRandom` (строка 13) начинается с создания начального значения, затем с помощью генератора случайных чисел, известного как вихрь Мерсенна[1], вычисляется случайное число с равномерным распределением между 100 и 200.

В конце программы (строка 41) поле `exp.counter` должно иметь значение около 1500, поскольку каждый из десяти потоков прибавляет единицу к счётчику примерно по 150 раз. Однако выполнение этой программы с помощью интерактивного компилятора Wandbox[2] даёт неожиданный результат:



Неожиданное поведение ссылки
на атомарный объект

Счётчик равен нулю. Почему? Дело в строке 24. Инициализация переменной в выражении `std::atomic<int> counter{exp.counter}` создаёт копию первоначального счётчика. Следующий небольшой фрагмент иллюстрирует проблему.

Копирование ссылки

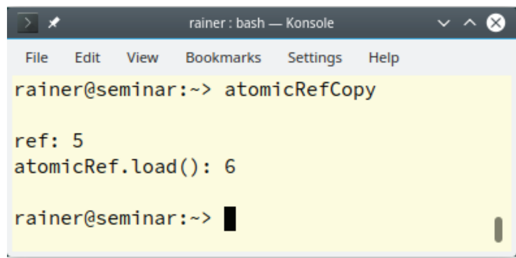
```
1  // atomicRefCopy.cpp
2
3  #include <atomic>
4  #include <iostream>
5
6  int main() {
7      std::cout << '\n';
8
9      int val{5};
10     int& ref = val;
11     std::atomic<int> atomicRef(ref);
12     ++atomicRef;
```

[1] https://en.wikipedia.org/wiki/Mersenne_Twister (имеется статья на русском языке: https://ru.wikipedia.org/wiki/Вихрь_Мерсенна. – *Прим. перев.*).

[2] https://wandbox.org/.

```
13     std::cout << “ref: “ << ref << ‘\n’;
14     std::cout << “atomicRef.load(): “ << atomicRef.load() << ‘\n’;
15
16     std::cout << ‘\n’;
17 }
```

Операция инкремента в строке 12 не имеет отношения к ссылке, объявленной в строке 10. Значение по ссылке не меняется.



Копирование ссылки

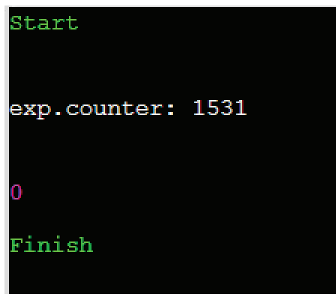
Замена типа переменной `counter` с `std::atomic<int>` на `std::atomic_ref<int>` решает проблему.

Использование типа std::atomic_ref<int>

```
1  // atomicReference.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <random>
6  #include <thread>
7  #include <vector>
8
9  struct ExpensiveToCopy {
10     int counter{};
11 };
12
13 int getRandom(int begin, int end) {
14     std::random_device seed;        // initial randomness
15     std::mt19937 engine(seed());    // generator
16     std::uniform_int_distribution<> uniformDist(begin, end);
17
18     return uniformDist(engine);
19 }
20
21 void count(ExpensiveToCopy& exp) {
22     std::vector<std::thread> v;
23     std::atomic<int> counter{exp.counter};
24
25     for (int n = 0; n < 10; ++n) {
26         v.emplace_back([&counter] {
```

```
27             auto randomNumber = getRandom(100, 200);
28             for (int i = 0; i < randomNumber; ++i) { ++counter; }
29         });
30     }
31 
32     for (auto& t : v) t.join();
33 }
34 
35 int main() {
36     std::cout << '\n';
37 
38     ExpensiveToCopy exp;
39     count(exp);
40     std::cout << "exp.counter: " << exp.counter << '\n';
41 
42     std::cout << '\n';
43 }
```

Теперь программа выдаёт ожидаемый результат:



Работа программы
при использовании типа std::atomic_ref<int>

Подобно шаблону `std::atomic`, определены специализации шаблона `std::atomic_ref` для различных стандартных типов данных.

2.3.6.2. Специализации шаблона std::atomic_ref

Шаблон `std::atomic_ref` позволяет подставлять пользовательские типы данных, а также обладает частичными специализациями для типов указателей и полными специализациями для арифметических типов, в частности целочисленных и вещественных с плавающей точкой.

2.3.6.2.1. Основной вариант шаблона

Основное определение шаблона `std::atomic_ref` позволяет подставить в качестве параметра пользовательский тип `T` с тривиальным копированием, т. е. удовлетворяющий ограничению TriviallyCopyable. Например:

```
struct Counter {
    int a;
    int b;
};

Counter counter;
std::atomic_ref<Counter> cnt(counter);
```

2.3.6.2.2. Частичные специализации для типов указателей

Стандартом определены частичные специализации шаблона для типов указателей: `std::atomic_ref<T*>`.

2.3.6.2.3. Специализации для арифметических типов

В стандарте определены полные специализации для целочисленных типов и типов с плавающей точкой:

- символьные типы: `char`, `char8_t` (C++20), `char16_t`, `char32_t` и `wchar_t`;
- стандартные целочисленные типы со знаком: `signed char`, `short int`, `long` и `long long`;
- стандартные целочисленные типы без знака: `unsigned char`, `unsigned short int`, `unsigned long` и `unsigned long long`;
- дополнительные целочисленные типы, определённые в заголовочном файле `<cstdint>`:
  - `int8_t`, `int16_t`, `int32_t`, `int64_t` (типы со знаком разрядностью ровно 8, 16, 32 и 64 бита соответственно);
  - `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` (беззнаковые типы разрядностью ровно 8, 16, 32 и 64 бита соответственно);
  - `int_fast8_t`, `int_fast16_t`, `int_fast32_t` и `int_fast64_t` (наиболее быстрые типы со знаком разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t` и `uint_fast64_t` (наиболее быстрые типы без знака разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `int_least8_t`, `int_least16_t`, `int_least32_t` и `int_least64_t` (типы со знаком с наименьшей доступной разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `uint_least8_t`, `uint_least16_t`, `uint_least32_t` и `uint_least64_t` (типы без знака с наименьшей доступной разрядностью не менее 8, 16, 32 и 64 бита соответственно);
  - `intmax_t` и `uintmax_t` (знаковый и беззнаковый целочисленный типы с наибольшей доступной разрядностью);
  - `intptr_t` и `uintptr_t` (знаковый и беззнаковый целочисленный типы, позволяющие хранить значения указателей);
- стандартные типы с плавающей точкой: `float`, `double` и `long double`.

2.3.6.3. Полный список атомарных операций

Ниже приведён полный перечень операций, поддерживаемых шаблоном `std::atomic_ref`.

<table>
<tr><th>Функция-член</th><th>Описание</th></tr>
<tr><td><code>is_lock_free</code></td><td>Проверить, гарантирует ли объект работу без блокировок</td></tr>
<tr><td><code>is_always_lock_free</code></td><td>Проверить на этапе компиляции, гарантирует ли тип работу без блокировок</td></tr>
<tr><td><code>load</code></td><td>Атомарным образом вернуть значение</td></tr>
<tr><td><code>operator T</code></td><td>Атомарным образом вернуть значение эквивалентно операции <code>atom.load()</code></td></tr>
<tr><td><code>store</code></td><td>Атомарным образом поместить в атомарную переменную новое значение</td></tr>
<tr><td><code>exchange</code></td><td>Атомарным образом поместить в атомарную переменную новое значение и вернуть её старое значение</td></tr>
<tr><td><code>compare_exchange_strong</code></td><td rowspan="2">Атомарным образом сравнить значение атомарной переменной с заданным и в зависимости от результата сравнения обменять значение с неатомарной переменной. Подробнее см. раздел 2.3.3.1.3</td></tr>
<tr><td><code>compare_exchange_weak</code></td></tr>
<tr><td><code>fetch_add</code>, <code>+=</code></td><td rowspan="2">Атомарным образом увеличить (уменьшить) значение атомарной переменной на заданную величину</td></tr>
<tr><td><code>fetch_sub</code>, <code>-=</code></td></tr>
<tr><td><code>fetch_or</code>, <code>|=</code></td><td rowspan="3">Атомарным образом выполнить над атомарной переменной и заданным значением соответствующую побитовую операцию (ИЛИ, И, исключающее ИЛИ)</td></tr>
<tr><td><code>fetch_and</code>, <code>&=</code></td></tr>
<tr><td><code>fetch_xor</code>, <code>^=</code></td></tr>
<tr><td><code>++</code>, <code>--</code></td><td>Пре- и постинкремент (декремент) атомарной переменной</td></tr>
<tr><td><code>notify_one</code> (C++20)</td><td>Оповестить один ожидающий поток</td></tr>
<tr><td><code>notify_all</code> (C++20)</td><td>Оповестить все ожидающие потоки</td></tr>
<tr><td><code>wait</code> (C++20)</td><td>Заблокировать поток до оповещения. При пробуждении сравнить текущее значение атомарной переменной со старым значением для предотвращения ложных и утерянных пробуждений</td></tr>
</table>

Комбинированные операции присваивания возвращают новое значение переменной по ссылке, тогда как соответствующие им `fetch`-функции возвращают старое значение.

Каждая функция-член принимает, помимо основного, ещё параметр, отвечающий за упорядочение доступа к памяти. По умолчанию используется значение `std::memory_order_seq_cst`, однако также можно явно передавать значения `std::memory_order_relaxed`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release` или `std::memory_order_acq_rel`. Функции-члены `compare_exchange_strong` и `compare_exchange_weak` принимают два параметра, управляющих упорядочением доступа к памяти: один на случай успешного выполнения операции и второй – для неуспешного. Если в явном виде указать лишь один параметр, он будет использован в обоих случаях. Разные порядки доступа будут подробно рассмотрены в разделе 2.4.

Конечно, не каждая из этих операций поддерживается для любых типов данных, которые могут подставляться в качестве параметра в шаблон `std::atomic_ref`. В следующей таблице показана поддержка операций в зависимости от типа-параметра.

Поддержка операций шаблоном std::atomic_ref в зависимости от типа-параметра

Функция	atomic_ref<T>	atomic_ref <floating>	atomic_ref<T*>	atomic_ref <integral>
is_lock_free	да	да	да	да
is_always_lock_free	да	да	да	да
load	да	да	да	да
operator T	да	да	да	да
store	да	да	да	да
exchange	да	да	да	да
compare_exchange_strong	да	да	да	да
compare_exchange_weak	да	да	да	да
fetch_add, +=		да	да	да
fetch_sub, -=		да	да	да
fetch_or, |=				да
fetch_and, &=				да
fetch_xor, ^=				да
++, --			да	да
notify_one (C++20)	да	да	да	да
notify_all (C++20)	да	да	да	да
wait (C++20)	да	да	да	да

Атомарные переменные и атомарные операции играют роль основных строительных блоков, на которых основывается модель памяти. Они позволяют устанавливать требования к синхронизации и упорядочиванию доступа к памяти – ограничения, которые остаются в силе также и для неатомарных переменных. Займёмся теперь подробным рассмотрением синхронизации и порядка доступа к памяти.

2.4. Синхронизация и порядок доступа к памяти

Невозможно каким-либо образом настроить поведение атомарного типа данных, однако у атомарных операций можно настраивать ограничения, связанные с синхронизацией и порядком доступа к памяти. Возможность изменять параметры синхронизации и порядка доступа составляет исключительную особенность языка C++; модель памяти, принятая в языках C# и Java, этого делать не позволяет.

В языке C++ имеется шесть вариантов модели памяти. Чрезвычайно важно понимать их отличительные черты.

2.4.1. Шесть вариантов модели памяти в языке C++

Из предыдущих глав читателю известно, что способов упорядочения доступа к памяти в языке C++ имеется шесть. По умолчанию в атомарных операциях

используется порядок `std::memory_order_seq_cst`. Его название означает «последовательная согласованность» (англ. *sequential consistency*). Кроме того, можно в явном виде указать какой-либо из оставшихся пяти порядков. Какие же варианты существуют в языке C++?

Порядки доступа к памяти

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
}
```

Чтобы разобраться в этих шести способах упорядочивания доступа к памяти, стоит ответить на два вопроса:

1. Какой порядок доступа к памяти следует использовать для тех или иных видов атомарных операций?
2. Какие ограничения на синхронизацию и порядок доступа к памяти накладываются в каждой из шести моделей?

Наш план прост: ответить на оба вопроса.

2.4.1.1. Виды атомарных операций

Существует три вида атомарных операций:

- операции чтения – с ними применяются порядки `memory_order_acquire` и `memory_order_consume`;
- операции записи, с которыми нужно использовать порядок `memory_order_release`;
- операции чтения-модификации-записи, для которых подходят порядки `memory_order_acq_rel` и `memory_order_seq_cst`.

Вариант `memory_order_relaxed` вообще не накладывает ограничений на синхронизацию и порядок доступа к памяти. Он не вписывается в рассматриваемую классификацию.

В следующей таблице атомарные операции разделены по типам в соответствии с тем, осуществляют они чтение или запись.

Функция	Чтение	Запись	Чтение-модификация-запись
test_and_set			да
clear		да	
is_lock_free	да		
load	да		
operator T	да		
store		да	
exchange			да

Функция	Чтение	Запись	Чтение-модификация-запись
compare_exchange_strong			да
compare_exchange_weak			да
fetch_add, +=			да
fetch_sub, -=			
fetch_or, |=			да
fetch_and, &=			
fetch_xor, ^=			
++, --			да
notify_one		да	
notify_all		да	
wait	да		

Операции чтения-модификации-записи предоставляют ещё одну дополнительную гарантию: они всегда работают с самым новым значением переменной. Это означает, например, что многократные вызовы вида atomVar.fetch_sub(1) из различных потоков дают убывающую последовательность чисел без дублирующихся или пропущенных значений.

Если в операции атомарного чтения вида `atomVar.load()` использовать модель памяти, предназначенную для операций записи или чтения-модификации-записи, то часть модели памяти, относящаяся к записи, не сыграет никакой роли. Например, вызов `atomVar.load(std::memory_order_acq_rel)` ведёт себя так же, как и вызов `atomVar.load(std::memory_order_acquire)`. Подобным же образом вызов `atomVar.load(std::memory_order_release)` эквивалентен вызову `atomVar.load(std::memory_order_relaxed)`.

2.4.1.2. Ограничения на синхронизацию и порядок доступа

Говоря упрощённо, в языке C++ определены три различных ограничения на синхронизацию и порядок доступа:

- последовательно-согласованное выполнение (`memory_order_seq_cst`);
- синхронизация захвата и освобождения (`memory_order_consume`, `memory_order_acquire`, `memory_order_release` и `memory_order_acq_rel`);
- ослабленная синхронизация (`memory_order_relaxed`).

Если последовательно-согласованное выполнение устанавливает глобальный порядок для всех операций, выполняемых всеми потоками, то семантика захвата и освобождения требует лишь синхронизации операций чтения с операциями записи, выполняемыми из различных потоков над одной и той же атомарной переменной. Ослабленная семантика гарантирует только упорядоченную модификацию некоторой атомарной переменной `m`. Упорядоченная модификация означает лишь, что все модификации данной атомарной переменной `m` образуют некоторую вполне упорядоченную последовательность. Это значит, что операции чтения атомарного объекта из какого-нибудь потока никогда не увидят значений, более старых, чем те, что поток уже успел увидеть.

Разнообразие моделей памяти и влияний, которые они оказывают на атомарные и неатомарные операции, делает модель памяти языка C++ увлекательным и сложным предметом. Рассмотрим подробнее ограничения на синхронизацию и порядок доступа к памяти, присущие последовательно-согласованной семантике, семантике захвата и освобождения и ослабленной семантике.

2.4.2. Последовательно-согласованное выполнение

Познакомимся глубже с последовательной согласованностью. Ключевую роль здесь играет то обстоятельство, что все операции во всех потоках выполняются как бы по единым часам. Наличие глобальных часов делает эту модель памяти интуитивно понятной.

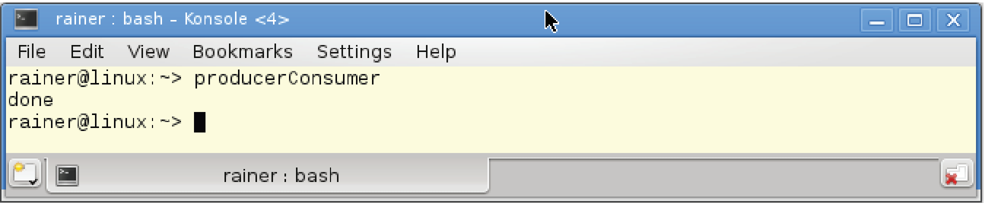
Интуитивная очевидность последовательно-согласованного выполнения имеет, однако, свою цену. Система должна синхронизировать потоки между собой.

Ниже показан пример программы, которая синхронизирует потоки производителя и потребителя, используя для этого семантику последовательной согласованности.

Синхронизация производителя и потребителя на основе последовательной согласованности

```
1  // producerConsumer.cpp
2
3  #include <atomic>
4  #include <
5  std::string work;
6  std::atomic<bool> ready(false);
7
8  void consumer() {
9    while(!ready.load()) {}
10   std::cout<< work << std::endl;
11 }
12
13 void producer() {
14   work= “done”;
15   ready=true;
16 }
17
18 int main() {
19   std::thread prod(producer);
20   std::thread con(consumer);
21   prod.join();
22   con.join();
23 }
```

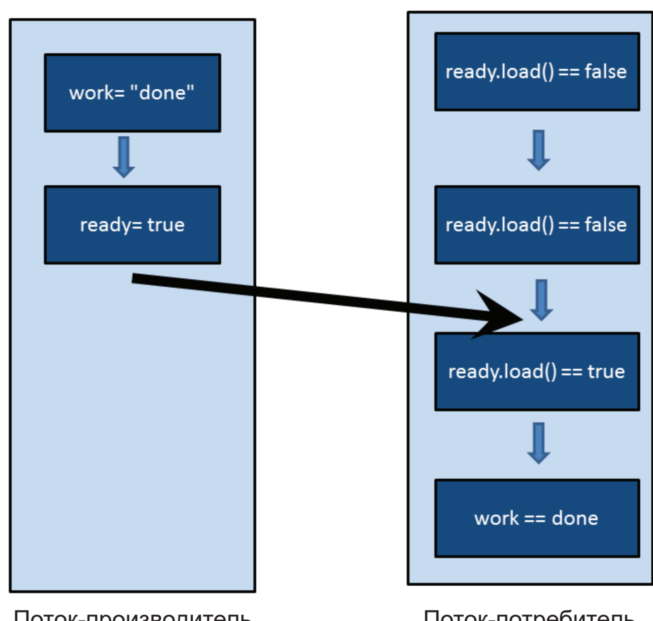
Результат выполнения этой программы может показаться не слишком впечатляющим.



Синхронизация производителя и потребителя
на основе последовательной согласованности

Благодаря последовательно-согласованному выполнению операций поведение программы детерминировано. Она всегда печатает сообщение «done» (с англ. «сделано»).

На следующем рисунке показана последовательность выполнения операций. Поток-потребитель ожидает в цикле до тех пор, пока в атомарной переменной `ready` не окажется значение `true`. После этого поток-потребитель продолжает свою работу.



Порядок выполнения операций
в последовательно-согласованной семантике

Нетрудно понять, что программа всегда печатает сообщение «done». Нужно лишь привлечь две главные черты последовательно-согласованной семантики. С одной стороны, оба потока выполняют свои операции в том порядке, в котором они указаны в исходном коде. С другой стороны, каждый поток видит результат выполнения операций другим потоком в том же порядке, в котором тот их выполнил. Оба потока работают по одним и тем же глобаль-

ным часам. Таким образом, конструкция while(!ready.load()){} обеспечивает синхронизацию потоков производителя и потребителя.

Это рассуждение можно провести более формальным образом, если воспользоваться терминологией, принятой для порядка доступа к памяти.

1. work= «done» располагается прежде ready = true.
   Следовательно, work= «done» происходит прежде ready = true.
2. while(!ready.load()){} располагается прежде std::cout << work << std::endl.
   Следовательно, while(!ready.load()){} происходит прежде std::cout<< work << std::endl.
3. ready= true синхронизировано с while(!ready.load()){}.
   Следовательно, ready= true происходит между потоками прежде while (!ready.load()){}.
   Следовательно, ready= true происходит прежде while (!ready.load()){}.

В силу транзитивности отношения предшествования отсюда следует, что work= «done» происходит прежде, чем ready = true, затем происходит ready= true, далее while (!ready.load()){}, после чего происходит std::cout << work << std::endl.

В модели последовательной согласованности каждый поток видит результаты операций, выполняемых любым другим потоком (а значит, и всех потоков вместе взятых) в одном и том же порядке. Эта важнейшая особенность последовательно согласованного выполнения не имеет места, если атомарные операции выполняются в соответствии с семантикой захвата и освобождения. Семантика захвата и освобождения – это область, в которую языки C# и Java не вторгаются. Также это область, в которой наша интуиция начинает давать сбои.

2.4.3. Семантика захвата и освобождения

При семантике захвата и освобождения нет глобальной синхронизации между всеми потоками; синхронизируются лишь атомарные операции **над одной и той же атомарной переменной**. Операция записи в некоторую атомарную переменную, выполняемая одним потоком, синхронизируется с операцией чтения из этой же переменной в другом потоке.

Семантика захвата и освобождения основывается на одном ключевом правиле: операция освобождения атомарной переменной синхронизируется с операцией захвата этой же переменной – тем самым устанавливается ограничение на порядок их выполнения. Ограничение это состоит в том, что никакие операции чтения и записи атомарной переменной не могут быть перемещены[1] после операции освобождения; никакие операции чтения и записи атомарной переменной не могут быть перемещены ранее операции захвата.

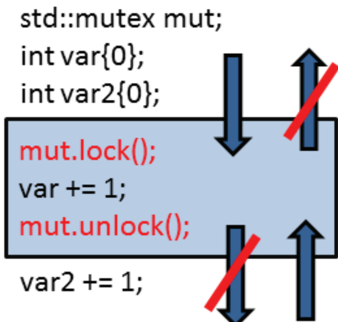
Что такое операции захвата и освобождения? Чтение из атомарной переменной посредством функций load или test_and_set – это операция захвата.

---

[1] Имеется в виду перемещение машинных операций оптимизирующим транслятором. – *Прим. перев.*

Но это ещё не всё: освобождение мьютекса синхронизируется с захватом этого мьютекса другим потоком. Создание потока синхронизируется с запуском выполняемого объекта в этом потоке. Завершение потока синхронизируется с вызовом функции-члена `join` для этого потока. Завершение выполняемого объекта, обёрнутого в задание, синхронизируется с вызовом функции `wait` или `get` этого задания. Операции захвата и освобождения всегда появляются парами.

Будет полезным понять следующую диаграмму.



Критический участок

Модель памяти и глубокое понимание многопоточности

Вот и ещё одна причина, по которой следует помнить модели памяти. Семантика захвата и освобождения помогает лучше понять примитивы синхронизации высокого уровня, например мьютекс. Та же логика подходит и для запуска потока с последующим вызовом функции-члена `join`. В обоих случаях имеют место операции захвата и освобождения. Эту линию продолжают и операции `wait` и `notify_one` для переменной условия: функция-член `wait` представляет собой захват, а функция-член `notify_one` – освобождение. Чем тогда является функция-член `notify_all`? Это тоже операция освобождения.

Рассмотрим ещё раз пример с циклом ожидания из раздела, посвящённого типу `std::atomic_flag`. Его можно реализовать более эффективным образом, так как в основе реализации лежит переменная типа `std::atomic_flag`, к которой применима семантика захвата и освобождения.

Цикл ожидания на основе семантики захвата и освобождения

```
1  // spinlockAcquireRelease.cpp
2
3  #include <atomic>
4  #include <thread>
5
6  class Spinlock{
7    std::atomic_flag flag;
8  public:
9    Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
```

```
11   void lock() {
12     while(flag.test_and_set(std::memory_order_acquire));
13   }
14
15   void unlock() {
16     flag.clear(std::memory_order_release);
17   }
18 };
19
20 Spinlock spin;
21
22 void workOnResource() {
23   spin.lock();
24   // shared resource
25   spin.unlock();
26 }
27
28 int main() {
29   std::thread t(workOnResource);
30   std::thread t2(workOnResource);
31
32   t.join();
33   t2.join();
34 }
```

Вызов `flag.clear` в строке 16 представляет собой операцию освобождения, а вызов `flag.test_and_set` в строке 12 – операцию захвата, а захват синхронизируется с освобождением. Тяжеловесная синхронизация потоков на основе последовательной согласованности (`std::memory_order_seq_cst`) заменяется более лёгкой и быстрой семантикой захвата и освобождения (`std::memory_order_acquire` и `std::memory_order_release`). Поведение программы при этом не изменяется.

Хотя вызов `flag.test_and_set` и представляет собой операцию чтения-модификации-записи, в данном случае для неё довольно семантики захвата. В общем итоге: переменная `flag` атомарна и потому гарантирует строгий порядок модификаций. Это означает, что все модификации над переменной `flag` происходят в определённом порядке, и этот порядок полон.

Порядок выполнения, налагаемый семантикой захвата и освобождения, транзитивен. Это значит, что если между потоками `a` и `b` установлено отношение захвата и освобождения, и если отношение захвата и освобождения существует между потоками `b` и `c`, то это отношение имеет место и между потоками `a` и `c`.

2.4.3.1. Транзитивность

Операция освобождения синхронизируется с операцией захвата этой же атомарной переменной, и между ними устанавливается ограничение на порядок выполнения. Этого хватает, чтобы синхронизировать потоки, если они работают с одной и той же атомарной переменной. Однако как быть,

если у двух потоков нет доступа к общей атомарной переменной? Не хотелось бы использовать семантику последовательной согласованности из-за чрезмерных накладных расходов, хорошо было бы заменить её легковесной семантикой захвата и освобождения.

Ответ вполне очевиден. Синхронизировать не связанные между собой потоки можно, если воспользоваться транзитивностью семантики захвата и освобождения.

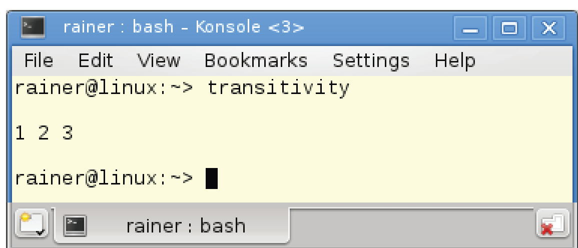
В следующем примере поток `t2`, выполняющий функцию `deliveryBoy` (с англ. «разносчик»), служит связующим звеном между потоками `t1` и `t3`.

Транзитивность в семантике захвата и освобождения

```
1  // transitivity.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> mySharedWork;
9  std::atomic<bool> dataProduced(false);
10 std::atomic<bool> dataConsumed(false);
11
12 void dataProducer() {
13     mySharedWork = {1,0,3};
14     dataProduced.store(true, std::memory_order_release);
15 }
16
17 void deliveryBoy() {
18     while(!dataProduced.load(std::memory_order_acquire));
19     dataConsumed.store(true, std::memory_order_release);
20 }
21
22 void dataConsumer() {
23     while(!dataConsumed.load(std::memory_order_acquire));
24     mySharedWork[1] = 2;
25 }
26
27 int main() {
28   std::cout << std::endl;
29
30   std::thread t1(dataConsumer);
31   std::thread t2(deliveryBoy);
32   std::thread t3(dataProducer);
33
34   t1.join();
35   t2.join();
36   t3.join();
37
38   for (auto v: mySharedWork) {
39       std::cout << v << " ";
```

```
40    }
41
42    std::cout << "\n\n";
43  }
```

Результат работы этой программы детерминирован: переменная-контейнер `mySharedWork` содержит значения 1, 2 и 3.



Результат работы программы `transitivity.cpp`

Два обстоятельства играют здесь ключевую роль:

1) поток `t2` в строке 18 ждёт, пока поток `t3` поставит переменную `dataProduced` в значение `true` в строке 14;
2) поток `t1` в строке 23 ждёт, пока поток `t2` поставит переменную `dataConsumed` в значение `true` в строке 19.

Остальное легко пояснить графически.

```
void dataProducer(){
    mySharedWork={1,0,3};
    dataProduced.store(true, std::memory_order_release);
}

void deliveryBoy(){
    while( !dataProduced.load(std::memory_order_acquire) );
    dataConsumed.store(true,std::memory_order_release);
}

void dataConsumer(){
    while( !dataConsumed.load(std::memory_order_acquire) );
    mySharedWork[1]= 2;
}
```

Располагается перед
Синхронизируется

Транзитивность семантики захвата и освобождения

Самое важное на этом рисунке – стрелки.

- Голубые стрелки изображают отношение «располагается перед». Оно означает, что все операции в пределах одного потока выполняются в том порядке, в котором они расположены в исходном коде.
- Красные стрелки – это отношения «синхронизируется с». Это отношение имеет место потому, что им связаны операции над одной и той же

атомарной переменной в семантике захвата и освобождения. Синхронизация действий над атомарными переменными и, следовательно, синхронизация потоков выполняется в особых местах кода.
- Отношение «располагается перед» влечёт за собой отношение «происходит ранее», а отношение «синхронизируется с» влечёт отношение «происходит ранее между потоками».

Остались сущие мелочи. Порядок операций «происходит ранее» и «происходит ранее между потоками» соответствует направлению стрелок сверху вниз. Наконец, мы установили гарантию, что операция `mySharedWork[1] == 2` выполнится последней.

Операция освобождения синхронизируется с операцией захвата той же атомарной переменной, поэтому можно легко синхронизировать между собой потоки, *если*… Именно с этим «если» связано распространённое недоразумение.

2.4.3.2. Типичное недоразумение

Зачем посвящать особый раздел неправильному пониманию захвата и освобождения? Дело в том, что многие читатели и студенты автора уже попадались в эту ловушку. Рассмотрим простой пример.

2.4.3.2.1. Цикл ожидания присутствует

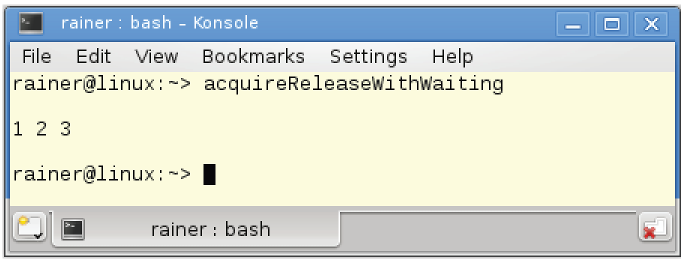
Начнём с простой программы.

Захват и освобождение с ожиданием

```
1  // acquireReleaseWithWaiting.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> mySharedWork;
9  std::atomic<bool> dataProduced(false);
10
11 void dataProducer() {
12     mySharedWork = {1, 0, 3};
13     dataProduced.store(true, std::memory_order_release);
14 }
15
16 void dataConsumer() {
17     while( !dataProduced.load(std::memory_order_acquire) );
18     mySharedWork[1] = 2;
19 }
20
21 int main() {
22   std::cout << std::endl;
23
```

```
24    std::thread t1(dataConsumer);
25    std::thread t2(dataProducer);
26
27    t1.join();
28    t2.join();
29
30    for (auto v: mySharedWork) {
31        std::cout << v << " ";
32    }
33
34    std::cout << "\n\n";
35 }
```

Поток-потребитель `t1` в строке 17 ожидает, пока поток-производитель `t2` в строке 13 установит переменную `dataProduced` в значение `true`. Переменная играет роль стражника, который гарантирует, что доступ к неатомарной переменной `mySharedWork` из разных потоков синхронизирован. Синхронизация в данном примере означает, что сначала поток-производитель `t2` инициализирует переменную `mySharedWork`, а лишь затем поток-потребитель `t1` подхватывает работу с этой переменной, присваивая элементу `mySharedWork[1]` значение 2. В целом поведение программы вполне определено.



Выполнение программы acquireReleaseWithWaiting

На следующем рисунке показаны отношения «происходит ранее» в пределах потока и отношения «синхронизируется с» между потоками. При этом отношение «синхронизируется с» влечёт за собой отношение «происходит ранее между потоками». Оставшаяся часть рассуждения сводится к транзитивности отношения «происходит ранее».

В конечном итоге получаем, что операция `mySharedWork = {1, 0, 3}` происходит ранее, чем операция `mySharedWork[1] = 2`.

Захват и освобождение с ожиданием

Какую деталь часто упускают из виду, рассуждая о поведении подобных программ? Это то самое «если», о котором говорилось выше.

2.4.3.2.2. Если…

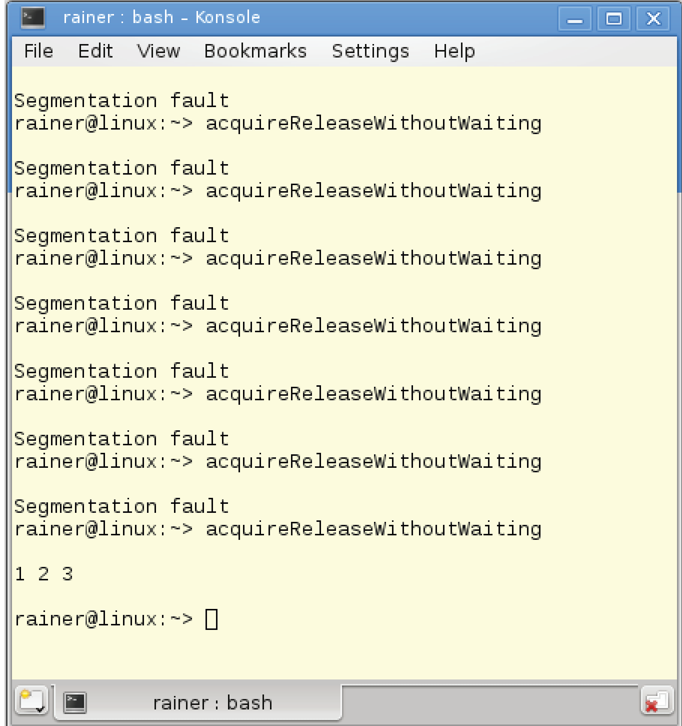
Что произойдёт, если поток-потребитель t1 в строке 17 не станет ждать потока-производителя t2?

Захват и освобождение без ожидания

```
1  // acquireReleaseWithoutWaiting.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> mySharedWork;
9  std::atomic<bool> dataProduced(false);
10
11 void dataProducer() {
12     mySharedWork = {1, 0, 3};
13     dataProduced.store(true, std::memory_order_release);
14 }
15
16 void dataConsumer() {
17     dataProduced.load(std::memory_order_acquire);
18     mySharedWork[1] = 2;
19 }
20
21 int main() {
22   std::cout << std::endl;
23
24   std::thread t1(dataConsumer);
25   std::thread t2(dataProducer);
```

```
26
27   t1.join();
28   t2.join();
29
30   for (auto v: mySharedWork){
31       std::cout << v << " ";
32   }
33
34   std::cout << "\n\n";
35 }
```

Эта программа обладает неопределённым поведением, поскольку при работе с переменной `mySharedWork` возникает гонка данных. Запуская программу, можно убедиться в недетерминированности её поведения.



Неопределённое поведение при семантике захвата и освобождения

В чём проблема? Операция освобождения `dataProduced.store` по-прежнему синхронизируется с операцией захвата `dataProduced.load`, но это отнюдь не означает, что операция захвата станет *ждать* операцию освобождения, и именно эта ситуация изображена на следующем рисунке. Операция `dataProduced.load` происходит до начала выполнения операции `dataProduced.store`, и синхронизация потоков пропадает.

время

```
void dataConsumer(){
  dataProduced.load(memory_order_acquire);
  mySharedWork[1]= 2;
}
```

```
void dataProducer(){
  mySharedWork={1,0,3};
  dataProduced.store(true,memory_order_release);
}
```

Происходит ранее

Захват и освобождение без ожидания

2.4.3.2.3. Решение

Отношение «синхронизируется с» означает в нашем примере следующее: *если* операция `dataProduced.store(true, std::memory_order_release)` происходит раньше, чем операция `dataProduced.load(std::memory_order_acquire)`, то все видимые результаты операций, предшествующих операции `store`, должны стать видимыми другому потоку с момента выполнения операции `load`. Ключевую роль в этой формулировке играет слово «если». В первой версии программы условие выполнялось благодаря оператору `while(!dataProduced.load(std::memory_order_acquire)`.

Повторим это же рассуждение на более высоком уровне строгости.

Все операции, выполняемые ранее операции `dataProduced.store(true, std::memory_order_release)`, происходят ранее, чем все операции другого потока, выполняемые после операции `dataProduced.load(std::memory_order_acquire)`, если соблюдено следующее условие: операция `store` происходит ранее операции `load`.

2.4.3.3. Последовательность освобождений

Последовательность освобождений – довольно сложное понятие, возникающее в связи с семантикой захвата и освобождения. Поэтому в следующем примере начнём с семантики захвата и освобождения.

Цикл ожидания с задержкой

```
1  // releaseSequence.cpp
2
3  #include <atomic>
4  #include <thread>
5  #include <iostream>
6  #include <mutex>
7
8  std::atomic<int> atom{0};
9  int somethingShared{0};
```

```
10
11 using namespace std::chrono_literals;
12
13 void writeShared() {
14     somethingShared = 2011;
15     atom.store(2, std::memory_order_release);
16 }
17
18 void readShared() {
19      while ( !(atom.fetch_sub(1, std::memory_order_acquire) > 0) ){
20          std::this_thread::sleep_for(100ms);
21      }
22
23      std::cout << "somethingShared: " << somethingShared << std::endl;
24 }
25
26 int main() {
27     std::cout << std::endl;
28
29     std::thread t1(writeShared);
30     std::thread t2(readShared);
31     std::thread t3(readShared);
32
33     t1.join();
34     t2.join();
35     t3.join();
36
37     std::cout << "atom: " << atom << std::endl;
38
39     std::cout << std::endl;
40 }
```

Рассмотрим этот пример сперва без потока `t3`. Операция записи в атомарную переменную в строке 15 синхронизируется с операцией чтения этой же переменной в строке 19. Синхронизация гарантирует, что результаты всех операций, выполненных перед записью, будут доступны после чтения. В том числе это означает, что обращение к переменной `somethingShared` не приводит к гонке данных.

Что изменится с появлением потока `t3`? Теперь, по-видимому, возникает гонка данных. Как уже говорилось выше, первый вызов операции `fetch_sub` в строке 19 обладает семантикой захвата и освобождения в паре с вызовом операции `store` в строке 15. Поэтому гонки данных по переменной `something-Shared` и не возникало.

Однако это неверно для второго вызова `fetch_sub`. Это операция чтения-модификации-записи, и вызывается она без флага `std::memory_order_release`. Это означает, что второй вызов операции `fetch_sub` не синхронизируется с первым, и это, казалось бы, может привести к гонке данных при работе с переменной `somethingShared`. Может, но не приводит – благодаря последовательности освобождений. Последовательность освобождений охватывает как первый, так и второй вызов операции `fetch_sub`. Следовательно, первый вызов операции `fetch_sub` связан со вторым отношением «происходит ранее».

В заключение приведём результат работы программы.

Последовательность освобождений

За более формальным изложением вопроса следует обратиться к рабочему варианту стандарта языка C++, документ N4659[1].

Последовательность освобождений

Последовательность освобождений над атомарным объектом M, начинающаяся с операции освобождения A, – это максимальная непрерывная последовательность побочных эффектов в порядке модификации объекта M, в которой первой стоит операция A и каждая последующая операция * либо выполняется тем же потоком, который выполнил операцию A, либо * представляет собой атомарную операцию чтения-модификации-записи.

Читатель, внимательно отнёсшийся к предыдущему материалу, особенно к разделу 2.2.2 о затруднениях, мог бы ожидать, что теперь речь пойдёт об ослабленной семантике, однако сперва познакомимся с моделью памяти `std::memory_order_consume`, которая во многом похожа на модель `std::memory_order_acquire`.

2.4.4. Модель памяти `std::memory_order_consume`

Модель памяти `std::memory_order_consume` из всех шести моделей в наибольшей степени окутана легендами. Для этого есть две причины: во-первых, она чрезвычайно трудна для понимания, а во-вторых, – это может измениться в будущем – сейчас её не поддерживает ни один компилятор. С появлением стандарта C++ 17 положение даже ухудшилось. Вот официальная формулировка: «Определение порядка захвата и потребления в настоящее время подвергается пересмотру, использование модели `std::memory_order_consume` временно не рекомендуется».

Как возможно, чтобы компилятор, реализующий стандарт C++ 11, не поддерживал модель памяти `std::memory_order_consume`? Ответ состоит в том, что компилятор подменяет её моделью `std::memory_order_acquire`. Такая замена

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

допустима, поскольку обе предполагают операцию чтения, или захвата. Модель `std::memory_order_consume` накладывает более мягкие ограничения на синхронизацию и упорядочивание доступа к памяти, чем модель `std::memory_order_acquire`. Поэтому модель захвата и освобождения выходит потенциально менее быстрой, но – и это самое главное – хорошо определённой.

2.4.4.1. Порядок захвата и освобождения

Начнём с примера программы, состоящей из двух потоков: `t1` и `t2`. Поток `t1` играет роль производителя, а поток `t2` – потребителя. Атомарная переменная `ptr` помогает синхронизировать производителя с потребителем.

Порядок захвата и освобождения

```
1  // acquireRelease.cpp
2
3  #include <atomic>
4  #include <thread>
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 atomic<string*> ptr;
11 int data;
12 atomic<int> atoData;
13
14 void producer(){
15     string* p  = new string("C++11");
16     data = 2011;
17     atoData.store(2014, memory_order_relaxed);
18     ptr.store(p, memory_order_release);
19 }
20
21 void consumer(){
22     string* p2;
23     while (!(p2 = ptr.load(memory_order_acquire)));
24     cout << "*p2: " << *p2 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30     cout << endl;
31
32     thread t1(producer);
33     thread t2(consumer);
34
35     t1.join();
36     t2.join();
37
```

```
38     cout << endl;
39 }
```

Прежде чем анализировать эту программу в подробностях, нужно сделать одну небольшую поправку.

2.4.4.2. Порядок освобождения и потребления

Заменим модель памяти `std::memory_order_acquire` в строке 23 порядком `std::memory_order_consume`.

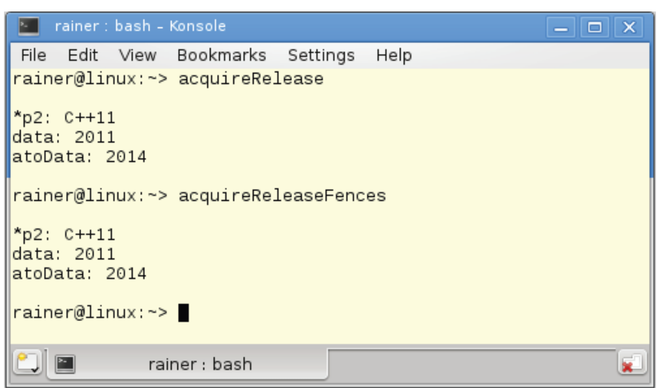
Порядок освобождения и потребления

```
1  // acquireConsume.cpp
2
3  #include <atomic>
4  #include <thread>
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 atomic<string*> ptr;
11 int data;
12 atomic<int> atoData;
13
14 void producer(){
15     string* p  = new string("C++11");
16     data = 2011;
17     atoData.store(2014,memory_order_relaxed);
18     ptr.store(p, memory_order_release);
19 }
20
21 void consumer(){
22     string* p2;
23     while (!(p2 = ptr.load(memory_order_consume)));
24     cout << "*p2: " << *p2 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30     cout << endl;
31
32     thread t1(producer);
33     thread t2(consumer);
34
35     t1.join();
36     t2.join();
37
38     cout << endl;
39 }
```

Теперь программа обладает неопределённым поведением. Это утверждение носит умозрительный характер – например, компилятор GCC 5.4 просто заменяет модель `std::memory_order_consume` на модель `std::memory_order_acquire`, поэтому «под капотом» обе программы работают одинаково.

2.4.4.3. Различие порядков «освобождение-захват» и «освобождение-потребление»

Результат работы обеих программ оказывается одинаковым.



Порядки «освобождение-захват» и «освобождение-потребление»

Рискуя лишний раз повториться, всё же следует сказать несколько слов о том, почему первая программа, `acquireRelease.cpp`, обладает вполне определённым поведением.

Операция `store` в строке 17 синхронизируется с операцией `load` в строке 23. Причина этого в том, что в операции `store` используется порядок `std::memory_order_release`, а в операции `load` – порядок `std::memory_order_acquire`. Отношение между операциями `store` и `load` предполагает синхронизацию. Какие ограничения налагает модель памяти на порядок выполнения операции захвата и освобождения? Порядок освобождения и захвата гарантирует, что результаты всех операций, выполнявшихся ранее операции записи (строка 17), доступны после операции чтения (строка 23). Таким образом, операции освобождения и захвата упорядочивают доступ как к неатомарной переменной `data` (строка 16) и атомарной переменной `atoData` (строка 17). Причём это справедливо даже несмотря на то, что для доступа к переменной `atoData` используется модель `std::memory_order_relaxed`.

Важнейший вопрос теперь звучит так: что произойдёт, если модель `std::memory_order_acquire` заменить моделью `std::memory_order_consume`?

2.4.4.4. Зависимости данных в модели `std::memory_order_consume`

В модели `std::memory_order_consume` учитываются зависимости между атомарными переменными. Эти зависимости бывают двоякого рода. Это отношения

переноса зависимости (англ. *carries-a-dependency-to*) в пределах потока и отношения порядка изменения (англ. *dependency-ordered before*) между операциями разных потоков. Оба вида зависимостей влекут за собой отношение «происходит ранее». Эти отношения имеют для нашей темы основополагающее значение. Что же в точности они означают?

- Если результат операции A используется в качестве операнда операции B, говорят, что операция A *переносит зависимость* на операцию B.
- Операция сохранения `store` A (при использовании моделей `std::memory_order_release`, `std::memory_order_acq_rel` или `std::memory_order_seq_cst`) *предшествует в порядке изменения* операции чтения `load` B (при использовании модели `std::memory_order_consume`), если результат операции B используется в какой-либо последующей операции C в том же потоке. Подчеркнём важность условия, что операции B и C должны находиться в одном и том же потоке.

Автору из личного опыта известно, что эти определения бывает поначалу нелегко осмыслить. Следующая диаграмма поможет представить данные понятия наглядно.

```
std::atomic<std::string*> ptr;
int data;
std::atomic<int> atoData;

void producer(){
    std::string* p  = new std::string("C++11");
    data = 2011;
    atoData.store(2014,std::memory_order_relaxed);
    ptr.store(p, std::memory_order_release);
}

void consumer(){
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)));
    std::cout << "*p2: " << *p2 << std::endl;
    std::cout << "data: " << data << std::endl;
    std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```


Зависимости данных в модели `std::memory_order_consume`

Выражение `ptr.store(p, std::memory_order_release)` в порядке изменения предшествует выражению `while (!(p2 = ptr.load(std::memory_order_consume)))`, поскольку в строке `std::cout << «*p2: « << *p2 << std::endl` используется её результат. Помимо того, оказывается, что строка `while (!(p2 = ptr.load(std::memory_order_consume)))` переносит зависимость на строку `std::cout << «*p2: « << *p2 << std::endl`, поскольку используемая в последней переменная `*p2` есть результат операции `ptr.load`.

Относительно выводимых на печать значений переменных `data` и `atoData` никаких гарантий дать нельзя. В самом деле, операции над этими переменными не связаны переносом зависимости с операцией `ptr.load`. Однако и это ещё не всё: поскольку переменная `data` не атомарна, при работе с ней возникает гонка данных. Это происходит потому, что два потока могут осуществлять доступ к ней одновременно, и один из них – поток `t1` – её модифицирует. Поэтому программа в целом обладает неопределённым поведением.

Наконец, мы добрались до ослабленной семантики.

2.4.5. Ослабленная семантика

Ослабленная семантика занимает крайнее положение в спектре моделей памяти. Это наислабейшая из моделей, она гарантирует лишь порядок модификации отдельно взятой атомарной переменной. Это означает, что все модифицирующие операции над атомарной переменной выполняются вполне упорядоченным образом.

2.4.5.1. Отсутствие ограничений на синхронизацию и порядок операций

Здесь всё довольно просто. Если нет никаких правил, их невозможно нарушить. Однако это даже слишком просто – программа всё же должна обладать хорошо определённым поведением. Хорошо определённое поведение обычно предполагает использование синхронизации и гарантий упорядоченности операций из более сильных моделей памяти. А именно: поскольку один поток может видеть результаты операций, выполняемых другим потоком, в произвольном порядке, необходимо обеспечить в программе такие точки, в которых операции из разных потоков синхронизируются между собой.

Типичный пример атомарной операции, в которой порядок выполнения операций несуществен, – это счётчик. Главное для счётчика – не то, как различные потоки его наращивают, а то, что каждая операция приращения атомарна и все приращения в конечном счёте будут выполнены. Рассмотрим следующий пример.

Счётчик с ослабленной семантикой

```
1  // relaxed.cpp
2
3  #include <vector>
4  #include <iostream>
5  #include <thread>
6  #include <atomic>
7
8  std::atomic<int> count = {0};
9
10 void add()
11 {
12     for (int n = 0; n < 1000; ++n) {
13         count.fetch_add(1, std::memory_order_relaxed);
14     }
15 }
16
17 int main()
18 {
19     std::vector<std::thread> v;
```

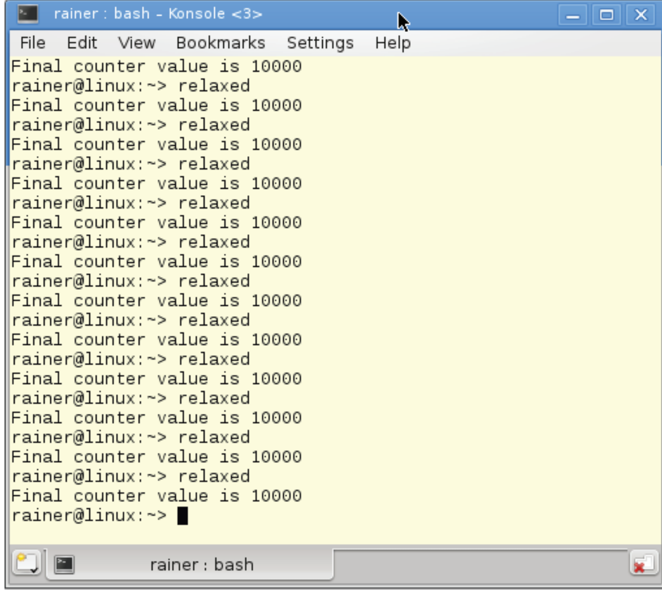
```
20     for (int n = 0; n < 10; ++n) {
21         v.emplace_back(add);
22     }
23     for (auto& t : v) {
24         t.join();
25     }
26     std::cout << “Final counter value is “ << count << ‘\n’;
27 }
```

Три самые замечательные строки в этой программе – это строки 13, 24 и 26. В строке 13 атомарная переменная `count` наращивается с ослабленной семантикой – таким образом, у нас есть гарантия, что эта операция выполняется атомарно. Все операции `fetch_add` из разных потоков выполняются по порядку. Функция `add` (строки 10–15) выполняется параллельно в каждом потоке. Потоки, выполняющие эту функцию, запускаются в строке 21.

Создание потоков образует одну точку синхронизации; вторую составляет вызов `t.join()` в строке 24. Иными словами, главный поток, создавший ряд дочерних потоков, позднее синхронизируется с ними в строке 24. Посредством вызова `t.join()` он ждёт завершения каждого дочернего потока. Именно вызов `t.join()` делает результаты предшествующих атомарных операций видимыми для других потоков. Говоря более формально, вызов `t.join()` играет роль операции освобождения.

В заключение остаётся сказать, что между операцией приращения переменной `counter` в строке 13 и взятием её значения в строке 26 имеет место отношение «происходит ранее».

В результате выполнения программы на печать всегда выводится число 1000. Однообразно? Нет, обнадёживающе!



Атомарный счётчик с ослабленной семантикой

Типичный пример использования ослабленной семантики с атомарным счётчиком – это счётчик ссылок из внутренней реализации типа `std::shared_ptr`. Ослабленная семантика применяется только для наращивания счётчика. Единственное, что существенно при прибавлении единицы к счётчику ссылок, – это чтобы операция выполнялась атомарным образом. Порядок выполнения этих операций значения не имеет. Однако для операций уменьшения счётчика ослабленная семантика не годится. Для уменьшения счётчика в деструкторе применяется семантика захвата и освобождения.

Наращивание счётчика как алгоритм без ожидания

Рассмотрим подробнее функцию `add`, начинающуюся в строке 10. Операция инкремента в строке 13 никак не синхронизируется с другими операциями – она лишь прибавляет единицу к атомарной переменной `count`. Следовательно, перед нами алгоритм не только без блокировок, но и без ожидания.

Следующее на очереди средство многопоточного программирования – `std::atomic_thread_fence`, позволяющее синхронизировать потоки и устанавливать порядок выполнения операций без использования атомарных переменных.

2.5. Барьеры

В стандартной библиотеке языка C++ определены два вида барьеров: `std::atomic_thread_fence` и `std::atomic_signal_fence`.

- Барьер `std::atomic_thread_fence` служит для синхронизации доступа к памяти между потоками.
- Барьер `std::atomic_signal_fence` служит для синхронизации между обработчиком сигнала и кодом, выполняющимся в том же потоке.

2.5.1. Барьер `std::atomic_thread_fence`

Барьер `std::atomic_thread_fence` непроницаем для некоторых операций. Этому барьеру для работы не нужны никакие атомарные переменные. Последние, однако, часто называют барьерами памяти. Из последующих разделов читатель легко поймёт, что такое барьеры и как с ними работать.

2.5.1.1. Что такое барьеры памяти

Что означает фраза «барьер непроницаем для некоторых операций»? Для каких именно операций? Глядя с высоты птичьего полёта, можно сказать, что есть лишь два вида операций: чтение и запись. Выражение `if (resultRead) return result`, например, состоит из операции чтения, за которой следует операция записи. Всего существует четыре способа соединить между собой две операции:

- LoadLoad: чтение, затем чтение;
- LoadStore: чтение, затем запись;
- StoreLoad: запись, затем чтение;
- StoreStore: запись, затем запись.

Конечно же, бывают и более сложные операции, состоящие из нескольких чтений и записей (например, операция `count++`), однако после разложения на элементарные операции они вписываются в эту классификацию.

Причём здесь барьеры памяти? Если поставить барьер между двумя операциями, можно гарантировать, что порядок операций в парах LoadLoad, LoadStore, StoreLoad или StoreStore останется неизменным. В отсутствие барьера могут быть переупорядочены неатомарные операции или атомарные операции с ослабленной семантикой.

2.5.1.2. Три барьера

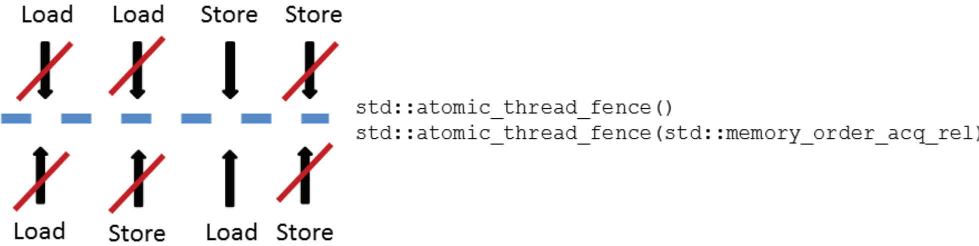
Обычно используются три вида барьеров. Их называют полным барьером, барьером захвата и барьером освобождения. Напомним, что операция чтения соответствует захвату, а запись – освобождению. Что случится, если поместить один из этих трёх барьеров в каждую из четырёх пар операций?

- Полный барьер. Вызов `std::atomic_thread_fence()`, помещённый между двумя произвольными операциями, предотвращает их переупорядочивание с одним исключением: пара StoreLoad может быть переупорядочена.
- Барьер захвата. Вызов `std::atomic_thread_fence(std::memory_order_acquire)` запрещает операцию чтения, расположенную перед барьером, менять местами с операцией чтения или записи после барьера.
- Барьер освобождения. Барьер `std::atomic_thread_fence(std::memory_order_release)` не позволяет операцию записи, расположенную после барьера, менять местами с операцией чтения или записи, расположенной до барьера.

Много сил может уйти на то, чтобы правильно понять определения барьеров захвата и освобождения и значение этих определений для неблокирующего программирования. Особенно трудны для понимания тонкие различия в семантике захвата и освобождения для атомарных операций. Поскольку изложение вплотную подходит к этой точке, стоит пояснить определения графически.

Какие виды операций могут пересекать тот или иной барьер? Посмотрим на следующие три рисунка. Если стрелка перечёркнута красной линией, барьер непроницаем для данного вида операций.

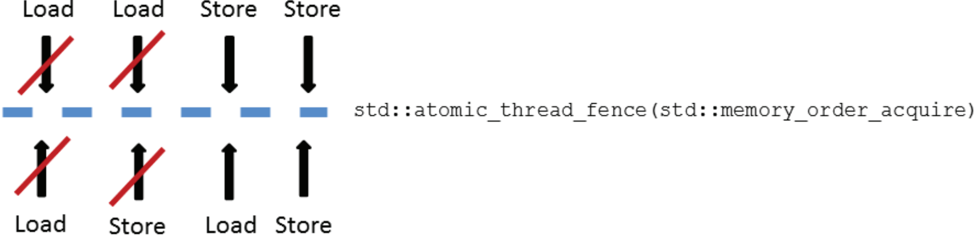
2.5.1.2.1. Полный барьер



Полный барьер

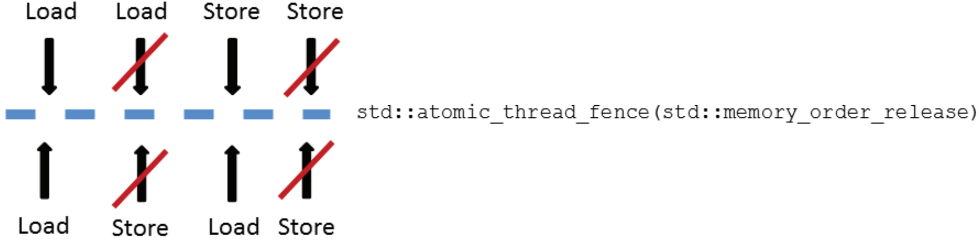
Вместо использования значения по умолчанию, `std::atomic_thread_fence()`, можно, конечно, указать барьер в явном виде: `thread_fence(std::memory_order_seq_cst)`. К барьерам обычно применяются правила последовательной согласованности. Таким образом, барьеры `std::atomic_thread_fence()` выполняются в едином глобальном порядке.

2.5.1.2.2. Барьер захвата



Барьер захвата

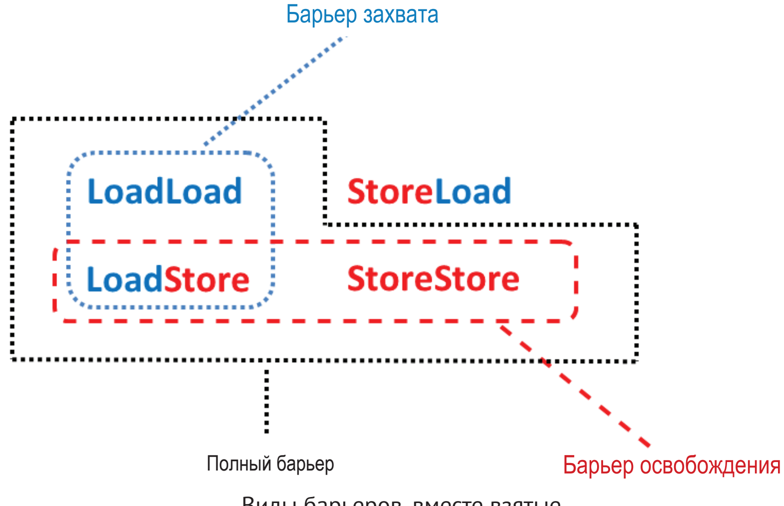
2.5.1.2.3. Барьер освобождения



Барьер освобождения

Три вида барьеров можно изобразить ещё более ясно.

2.5.1.2.4. Виды барьеров, вместе взятые



Виды барьеров, вместе взятые

Барьеры захвата и освобождения дают такие же гарантии синхронизации и упорядочивания доступа, как и атомарные операции с семантикой захвата и освобождения.

2.5.1.3. Барьеры захвата и освобождения

Наиболее очевидное различие между барьерами памяти и атомарными операциями состоит в том, что для барьеров не нужны атомарные переменные. Есть и ещё одно тонкое различие: барьеры захвата и освобождения более тяжеловесны, чем соответствующие атомарные операции.

2.5.1.3.1. Сравнение атомарных операций с барьерами

Для простоты изложения будем называть операциями захвата атомарные операции с семантикой захвата, аналогично для освобождения.

Смысл операций захвата и освобождения состоит в том, что они налагают на потоки ограничения, связанные с синхронизацией и упорядочиванием доступа к памяти. Эти ограничения соблюдаются также для атомарных операций с ослабленной семантикой и для неатомарных операций. Нужно иметь в виду, что операции захвата и освобождения встречаются парами. Кроме того, атомарные операции с семантикой захвата-освобождения должны относиться к одной и той же атомарной переменной. Сделав вводные замечания, можно теперь рассмотреть эти операции по отдельности. Начнём с операций захвата.

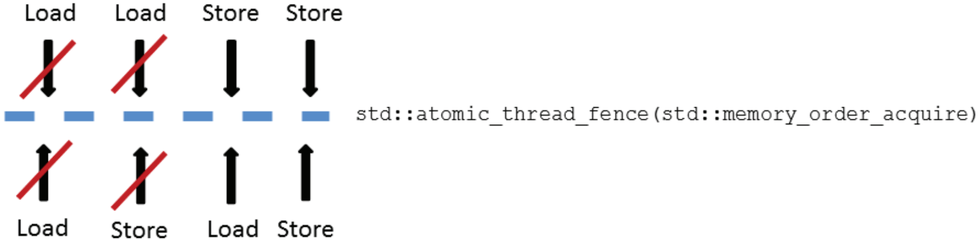
2.5.1.3.2. Операции захвата

Операции чтения над атомарными переменными, выполняемые с параметром `std::memory_order_acquire`, представляют собой операции захвата. Соответствующее ограничение на порядок доступа к памяти показано на следующем рисунке.

```
int ready= var.load(std::memory_order_acquire);
// load and store operations
```

Атомарная операция с семантикой захвата

Барьер `std::atomic_thread_fence` с порядком `std::memory_order_acquire` накладывает более жёсткие ограничения на переупорядочивание операций, как показано на следующем рисунке.



Барьер с семантикой захвата

Из этого сравнения видны два важных обстоятельства:

1) барьер с семантикой захвата накладывает более строгие ограничения на порядок операций. Хотя захват как атомарной переменной, так и барьера одинаково запрещает перемещать последующие операции чтения или записи перед захватом, барьер предоставляет ещё одну гарантию: никакие более ранние операции чтения не будут перемещены после захвата;
2) для чтения атомарной переменной `var` при использовании барьера довольно ослабленной семантики, ведь эта операция не будет перемещена после барьера.

Подобные соображения справедливы и для операций освобождения.

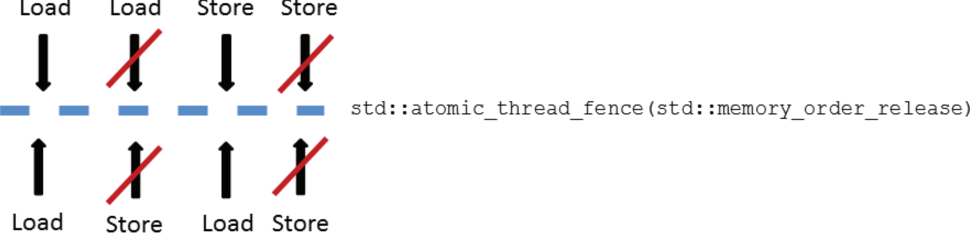
2.5.1.3.3. Операции освобождения

Операции записи над атомарными переменными, выполняемые с параметром `std::memory_order_release`, представляют собой операции освобождения. Ограничение на порядок доступа к памяти показано на следующем рисунке.

```
// load and store operations
var.store(1,std::memory_order_release);
```

Атомарная операция с семантикой освобождения

Ниже показано соответствующее ограничение для барьера.



Барьер с семантикой освобождения

В дополнение к ограничениям, характерным для операции освобождения атомарной переменной, освобождение барьера гарантирует два дополнительных свойства:

1) последующие операции записи не могут быть перемещены перед барьером;
2) для работы с переменной `var` достаточно ослабленной семантики.

Теперь пора сделать следующий шаг и создать программу с использованием барьеров.

2.5.1.4. Синхронизация с использованием атомарных переменных и барьеров

Реализуем на основе семантики захвата и освобождения типичную систему из производителя и потребителя данных. Сначала воспользуемся для этого атомарными операциями, а затем используем вместо них барьеры.

2.5.1.4.1. Реализация на основе атомарных операций

Начнём с атомарных операций, поскольку с ними читатель уже хорошо знаком.

Атомарные операции с семантикой захвата и освобождения

```
1  // acquireRelease.cpp
2
3  #include <atomic>
4  #include <thread>
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
```

```
9
10 atomic<string*> ptr;
11 int data;
12 atomic<int> atoData;
13
14 void producer(){
15     string* p  = new string("C++11");
16     data = 2011;
17     atoData.store(2014, memory_order_relaxed);
18     ptr.store(p, memory_order_release);
19 }
20
21 void consumer(){
22     string* p2;
23     while (!(p2 = ptr.load(memory_order_acquire)));
24     cout << "*p2: " << *p2 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30     cout << endl;
31
32     thread t1(producer);
33     thread t2(consumer);
34
35     t1.join();
36     t2.join();
37
38     cout << endl;
39 }
```

Эта программа должна выглядеть вполне понятной. Это классический пример, уже использованный в разделе, посвящённом модели `std::memory_order_consume`. На следующем рисунке показано, что поток-потребитель `t2` получает все значения от потока-производителя `t1`.

```
void producer(){
  std::string* p  = new std::string("C++11");
  data = 2011;
  atoData.store(2014,std::memory_order_relaxed);
  ptr.store(p, std::memory_order_release);
}

void consumer(){
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_acquire)));
  std::cout << "*p2: " << *p2 << std::endl;
  std::cout << "data: " << data << std::endl;
  std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

Происходит ранее
Синхронизируется с

Атомарные операции с семантикой захвата и освобождения

Поведение программы вполне определено в силу транзитивности отношения «происходит ранее». Нужно лишь объединить в цепочку три таких отношения:

1) операции в строках 15–17 происходят ранее строки 18, где присваивается значение указателю;
2) строка 23 с циклом ожидания происходит ранее строк 24–26;
3) строка 18 синхронизируется со строкой 23. Следовательно, строка 18 происходит между потоками ранее строки 23.

Однако сейчас дело станет ещё интереснее, когда перейдём к использованию барьеров. Их незаслуженно обходят вниманием в литературе по языку C++ и его моделям памяти.

2.5.1.4.2. Барьеры памяти

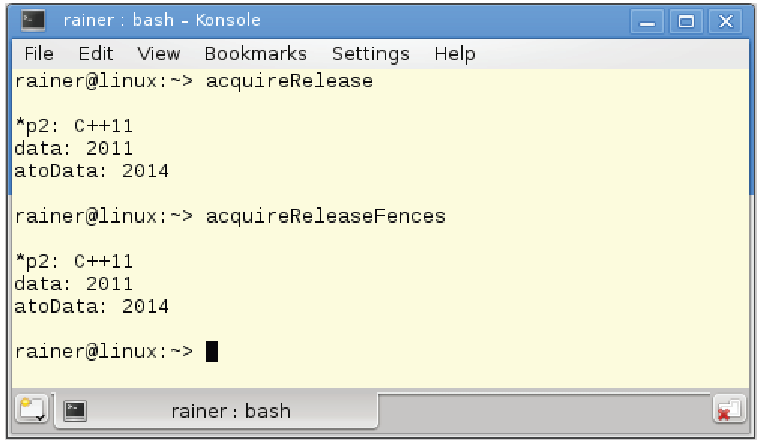
Приведённую выше программу очень просто переделать, применив барьеры.

Порядок захвата и освобождения с использованием барьеров

```
1  // acquireReleaseFences.cpp
2
3  #include <atomic>
4  #include <thread>
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 atomic<string*> ptr;
11 int data;
12 atomic<int> atoData;
13
14 void producer(){
15     string* p  = new string("C++11");
16     data = 2011;
17     atoData.store(2014, memory_order_relaxed);
18     atomic_thread_fence(memory_order_release);
19     ptr.store(p, memory_order_relaxed);
20 }
21
22 void consumer(){
23     string* p2;
24     while (!(p2 = ptr.load(memory_order_relaxed)));
25     atomic_thread_fence(memory_order_acquire);
26     cout << "*p2: " << *p2 << endl;
27     cout << "data: " << data << endl;
28     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
29 }
30
31 int main(){
32     cout << endl;
33
```

```
34      thread t1(producer);
35      thread t2(consumer);
36
37      t1.join();
38      t2.join();
39
40      delete ptr;
41
42      cout << endl;
43  }
```

Первый шаг состоит в том, чтобы вставить в программу барьеры с семантикой освобождения (строка 18) и захвата (строка 25). Затем нужно заменить семантику атомарных операций с освобождения и захвата на ослабленную (строки 19 и 24). Это довольно очевидно. Конечно же, атомарные операции захвата и освобождения можно заменить только на соответствующие барьеры. Ключевой момент состоит в том, что барьер освобождения синхронизируется с барьером захвата, что влечёт межпоточное отношение «происходит ранее». Ниже представлен результат работы программы.



Синхронизация на основе барьеров

Для любителей визуальной формы подачи материала ниже представлено графическое пояснение отношений синхронизации в этой программе.

Ключевой вопрос здесь звучит так: почему операции после барьера захвата видят результаты операций, стоящих перед барьером освобождения? Эта гарантия особенно интересна потому, что переменная `data` неатомарна, а операция `store` над переменной `atoData` выполняется с ослабленной семантикой. Могло бы показаться, что порядок этих операций может быть изменён реализацией, однако – благодаря барьеру освобождения в строке 18 и барьеру захвата в строке 25 – они обязательно выполняются в правильном порядке.

```
void producer(){
  std::string* p  = new std::string("C++11");
  data = 2011;
  atoData.store(2014,std::memory_order_relaxed);
  std::atomic_thread_fence(std::memory_order_release);
  ptr.store(p, std::memory_order_relaxed);
}

void consumer(){
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_relaxed)));
  std::atomic_thread_fence(std::memory_order_acquire);
  std::cout << "*p2: " << *p2 << std::endl;
  std::cout << "data: " << data << std::endl;
  std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

Происходит ранее
Синхронизируется с

Барьеры с семантикой захвата и освобождения

Для большей ясности приведём рассуждение полностью.

1. Предотвращено изменение порядка атомарных и неатомарных операций относительно барьеров захвата и освобождения.
2. Поток-потребитель `t2` ждёт в цикле (строка 24), пока переменной-указателю `ptr` не будет присвоено значение в потоке-производителе `t1`.
3. Барьер освобождения синхронизируется с барьером захвата.
4. В конечном счёте результаты всех операций с ослабленной семантикой и неатомарных операций, происходящих ранее барьера освобождения, видны после барьера захвата.

Синхронизация между барьерами освобождения и захвата

Следующее место из документа N4659[1] (рабочий вариант стандарта языка C++) может оказаться довольно трудным для понимания: «Барьер освобождения A синхронизируется с барьером захвата B, если существуют атомарные операции X и Y, работающие с одним и тем же атомарным объектом M, такие, что барьер A расположен перед операцией X, операция X модифицирует объект M, операция Y расположена перед барьером B и операция Y читает значение, записанное объектом X, или значение, записанное побочным эффектом какой-либо гипотетической последовательностью освобождения, которая начиналась бы с операции X, если бы X была операцией освобождения».

Это предложение стоит пояснить на примере только что разобранной программы:

- строка 18 представляет собой барьер освобождения A;
- строка 25 – это барьер захвата B;
- переменная `ptr`, объявленная в строке 10, есть атомарный объект M;
- операция `store` над объектом `ptr` в строке 19 есть операция записи X;
- операция `load` над объектом `ptr` в строке 24 – это операция чтения Y.

Напоследок отметим, что можно даже использовать операции захвата и освобождения, как в программе `acquireRelease.cpp`, с барьерами захвата

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

и освобождения, как в программе acquireReleaseFence.cpp, без ущерба для отношений синхронизации.

2.5.2. Барьер `std::atomic_signal_fence`

Барьер `std::atomic_signal_fence` позволяет установить порядок синхронизации неатомарных операций и атомарных операций с ослабленной семантикой между потоком и обработчиком сигнала, выполняемым в этом же потоке. Пример использования такого барьера показан в следующей программе.

Синхронизация с обработчиком сигнала

```
1  // atomicSignal.cpp
2
3  #include <atomic>
4  #include <cassert>
5  #include <csignal>
6
7  std::atomic<bool> a{false};
8  std::atomic<bool> b{false};
9
10 extern "C" void handler(int) {
11     if (a.load(std::memory_order_relaxed)) {
12         std::atomic_signal_fence(std::memory_order_acquire);
13         assert(b.load(std::memory_order_relaxed));
14     }
15 }
16
17 int main() {
18     std::signal(SIGTERM, handler);
19
20     b.store(true, std::memory_order_relaxed);
21     std::atomic_signal_fence(std::memory_order_release);
22     a.store(true, std::memory_order_relaxed);
23 }
```

Программа начинается с установки (в строке 18) обработчика для одного определённого сигнала – `SIGTERM`. Сигнал `SIGTERM` означает требование завершить программу. В программе расположены два барьера `std::atomic_signal_fence`: с семантикой освобождения в строке 21 и с семантикой захвата в строке 12. Они запрещают переупорядочивание операций: операции освобождения не могут перемещаться через барьер освобождения, а операции захвата – через барьер захвата. Следовательно, контрольное утверждение в строке 13 никогда не будет нарушено: ведь если операция `store` над переменной `a` (строка 22) выполнилась, то операция `store` над переменной `b` (строка 20) должна была выполниться ещё раньше.

3. Управление потоками

Язык C++ обладает интерфейсом управления потоками начиная со стандарта C++ 11, и этот интерфейс содержит всё необходимое для создания многопоточных программ. В нём есть собственно потоки, примитивы синхронизации для доступа к общим данным – мьютексы и блокировщики, локальные данные потоков, более сложные средства синхронизации потоков, такие как переменные условия, а также задания. Задания, также называемые обещаниями и фьючерсами, составляют более высокий уровень абстракции по сравнению с собственно потоками.

Средства многопоточного программирования в языке C++

Библиотека языка C++ поддерживает два вида потоков: базовые потоки типа `std::thread` (стандарт C++ 11) и усовершенствованные потоки типа `std::jthread` (стандарт C++ 20). Сначала займёмся базовыми потоками, а в следующем разделе главы рассмотрим усовершенствованные.

3.1. Базовые потоки: класс std::thread

Чтобы создать поток в программе на языке C++, нужно подключить заголовочный файл `<thread>`.

3.1.1. Создание потока

Объект типа `std::thread` представляет единицу выполнения программы. То, что должно быть выполнено в потоке, передаётся объекту `std::thread` при его создании в виде вызываемого объекта.

Вызываемый объект ведёт себя подобно функции. Конечно, он может быть в том числе и функцией, но также может быть функциональным объектом или лямбда-функцией. Значение, возвращаемое вызываемым объектом, игнорируется.

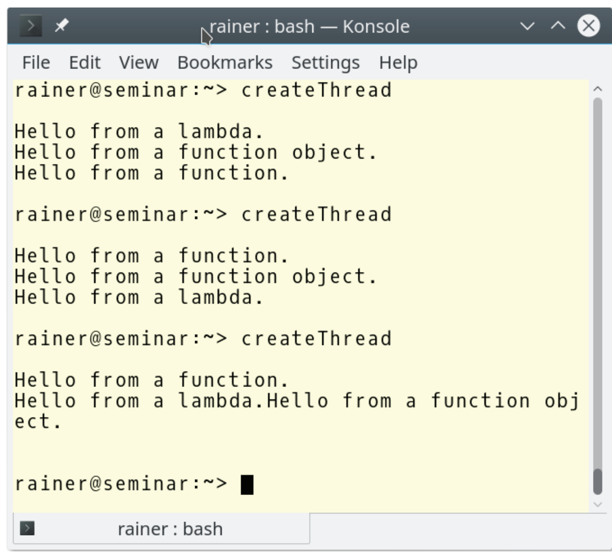
Сделав эти вводные замечания, можно рассмотреть пример.

Создание потоков с различными вызываемыми объектами

```
1  // createThread.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  void helloFunction(){
7    std::cout << "Hello from a function." << std::endl;
8  }
9
10 class HelloFunctionObject{
11   public:
12     void operator()() const {
13       std::cout << "Hello from a function object." << std::endl;
14     }
15 };
16
17 int main(){
18   std::cout << std::endl;
19
20   std::thread t1(helloFunction);
21
22   HelloFunctionObject helloFunctionObject;
23   std::thread t2(helloFunctionObject);
24
25   std::thread t3([]{std::cout << "Hello from a lambda." << std::endl;});
26
27   t1.join();
28   t2.join();
29   t3.join();
30
31   std::cout << std::endl;
32 };
```

Все три потока, `t1`, `t2` и `t3`, печатают сообщения на консоль. При этом в потоке `t1` выполняется обычная функция, в потоке `t2` выполняется функциональный объект (его класс объявлен в строках 10–15), а в потоке `t2` – лямбда-функция (строка 25). В строках 27–29 главный поток ожидает завершения всех дочерних потоков.

Посмотрим, как работает эта программа. Результаты её запуска довольно интересны.



Создание потоков с различными вызываемыми объектами

Три потока могут выполняться в произвольном порядке. Даже тексты, выводимые из разных потоков, могут перемешиваться между собой.

В этом примере главный поток, создающий дочерние потоки, отвечает за время жизни этих объектов.

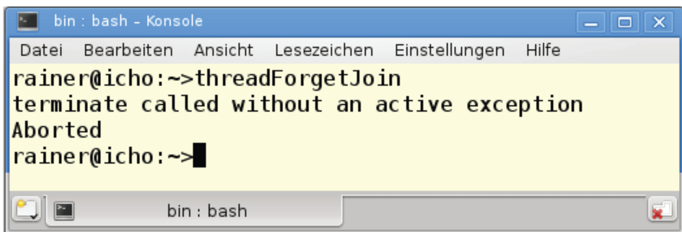
3.1.2. Время жизни потоков

Родители должны заботиться о своих детях. Из этого простого принципа вытекают далекоидущие следствия касательно времени жизни потоков. Рассмотрим пример: следующая программа должна запускать поток и выводить его идентификатор.

Забытое ожидание завершения потока

```
1  // threadWithoutJoin.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  int main(){
7    std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});
8  }
```

Однако программа этого не делает.



В чём причина аварийного завершения?

3.1.2.1. Функции join и detach

Жизнь потока t завершается с окончанием выполнения его вызываемого объекта. У создателя потока есть на выбор два варианта:

- подождать завершения дочернего потока с помощью вызова `t.join()`;
- оборвать связь с дочерним потоком с помощью вызова `t.detach()`.

Вызов `t.join()` полезен в случаях, когда последующий код каким-то образом зависит от результатов работы дочернего потока. Вызов `t.detach()` позволяет дочернему потоку выполняться независимо от объекта t. Это значит, что поток, отсоединённый от объекта, будет выполняться до тех пор, пока не завершится выполняющийся в нём код. Чаще всего отсоединяют потоки, в которых выполняется долгая фоновая работа, например потоки-серверы.

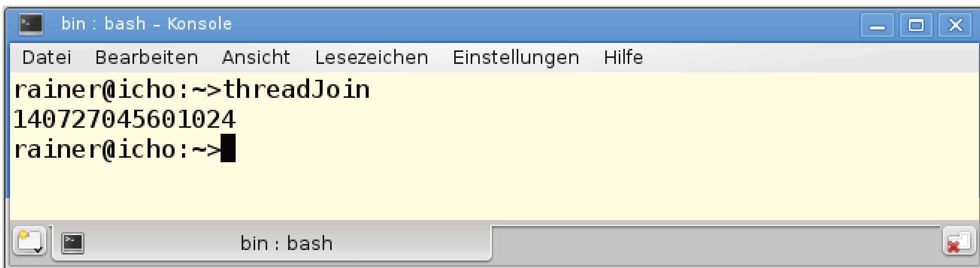
Объект-поток с завёрнутым в него вызываемым объектом (объекты типа `std::thread` можно создавать и пустыми, без вызываемых объектов) называется присоединяемым (англ. *joinable*), если для него не вызывалась ни одна из функций-членов `join` и `detach`. Деструктор объекта-потока в присоединяемом состоянии вызывает аварийное завершение программы посредством `std::terminate`. Именно в этом причина аварийного завершения программы `threadWithoutJoin.cpp`. Если применить функцию `join` или `detach` к объекту, для которого какая-либо из этих функций-членов уже была вызвана ранее, произойдёт исключение типа `std::system_error`.

Исправить предыдущую программу довольно легко – для объекта-потока t вызвать функцию-член `join`.

Ожидание завершения потока

```
1  // threadWithJoin.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  int main(){
7    std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});
8
9    t.join();
10 }
```

Теперь программа работает так, как ожидалось.



Ожидание завершения потока

Число, которое программа выводит на печать, – это уникальный идентификатор потока в системе.

Трудность с функцией detach

Конечно же, в последней программе вместо функции `join` можно было бы применить функцию `detach`. Объект-поток `t` перешёл бы в неприсоединяемое состояние, и его деструктор не стал бы аварийно завершать программу. Но в этом случае возникло бы иное затруднение. Поведение такой программы не определено, поскольку главный поток может завершиться раньше дочернего, и тогда последнему просто не останется времени, чтобы вывести на печать свой результат. Подробнее об этом пойдёт речь в разделе 13.7.

Класс scoped_thread Энтони Уильямса

Если своими руками управлять временем жизни потоков слишком обременительно, можно инкапсулировать класс `std::thread` в собственный класс-обёртку. Деструктор этого класса должен автоматически вызывать функцию `join` потока, если тот всё ещё находится в присоединяемом состоянии. Конечно, можно поступить и противоположным образом, автоматически вызывая для потока функцию `detach` – однако читатель уже знает, что это может быть сопряжено с определёнными трудностями.

Энтони Уильямс создал такой полезный класс и описал его в своей превосходной книге «Параллельное программирование на C++ в действии»[1]. Класс-обёртку он назвал `scoped_thread`. Конструктор этого класса принимает в качестве аргумента поток `t` и проверяет, находится ли он по-прежнему в присоединяемом состоянии. Если поток-аргумент `t` не поддерживает присоединение, нет и необходимости в классе `scoped_thread`. Если же поток `t` можно присоединить, деструктор класса вызывает `t.join()`. Поскольку конструктор копирования и копирующая операция присваивания в явном виде удалены, экземпляры класса `scoped_thread` не могут копироваться никаким способом.

```
// scoped_thread.cpp

#include <iostream>
#include <thread>
#include <utility>
```

---

[1] *Уильямс Э.* Параллельное программирование на C++ в действии: Практика разработки многопоточных программ. М.: ДМК Пресс, 2012. 672 с.

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

3.1.3. Передача аргументов при создании потока

Вызываемый объект, выполняющийся в потоке (в частности, обычная функция), может получать свои аргументы посредством копирования значения, по перемещению или по ссылке. Класс `std::thread` представляет собой шаблон с переменным числом аргументов, или, как их ещё называют, вариадический[1] шаблон.

Если поток получает на вход данные по ссылке, нужно быть чрезвычайно осторожным в том, что касается времени жизни передаваемых объектов и совместного доступа к данным.

3.1.3.1. Передача по значению и по ссылке

Рассмотрим следующий небольшой фрагмент кода:

```
std::string s{"C++11"}

std::thread t1([=]{ std::cout << s << std::endl; });
t1.join();

std::thread t2([&]{ std::cout << s << std::endl; });
t2.detach();
```

Поток `t1` получает данные по значению (т. е. путём создания копии), а поток `t2` – по ссылке.

Передача аргументов в поток по ссылке

По правде говоря, в этом примере допущено некоторое упрощение. Там сказано, что поток `t2` получает аргумент по ссылке, однако на самом деле это лямбда-функция захватывает ссылку на локальные данные. Чтобы передать по ссылке аргумент в функцию потока, нужно завернуть его в класс-обёртку ссылок[2]. Это просто сделать, воспользовавшись вспомогательной функцией[3] `std::ref`, определённой в заголовочном файле `<functional>`. Например:

[1] http://en.cppreference.com/w/cpp/language/parameter_pack.

[2] http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper.

[3] http://en.cppreference.com/w/cpp/utility/functional/ref.

```
void transferMoney(int amount, Account& from, Account& to){
...
}
...
std::thread thr1(transferMoney, 50, std::ref(account1), std::ref(account2));
```

Здесь поток `thr1` выполняет функцию `transferMoney`, которая получает аргументы по ссылке. Благодаря использованию функции `std::ref` в функцию будут переданы ссылки на объекты `account1` и `account2`.

Какие тонкости могут таиться в этих строчках кода? Поток `t2` получает строку `s` по ссылке, а затем отсоединяется от родительского потока. Время жизни строки `s` ограничено временем выполнения объемлющей функции. Время жизни глобального объекта `std::cout` ограничивается только временем выполнения главного потока программы. Поэтому время жизни объектов `s` и `std::cout` может оказаться короче, чем время выполнения потока `t2`. Таким образом мы оказываемся по уши в неопределённом поведении.

Если эти рассуждения читателя не убедили, ниже представлен пример того, как может выглядеть неопределённое поведение.

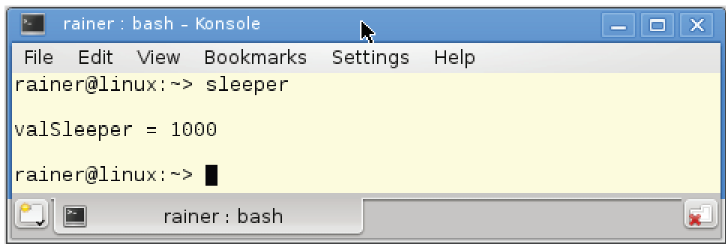
Передача в поток аргументов по ссылке

```
1  // threadArguments.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  class Sleeper{
8    public:
9      Sleeper(int& i_):i{i_}{};
10     void operator() (int k){
11       for (unsigned int j= 0; j <= 5; ++j){
12         std::this_thread::sleep_for(std::chrono::milliseconds(100));
13         i += k;
14       }
15     }
16   private:
17     int& i;
18 };
19
20
21 int main(){
22   std::cout << std::endl;
23
24   int valSleeper = 1000;
25 `  std::thread t(Sleeper(valSleeper), 5);
26   t.detach();
27   std::cout << “valSleeper = “ << valSleeper << std::endl;
28
29   std::cout << std::endl;
30 }
```

Какое значение имеет переменная `valSleeper` в строке 27? Переменная `valSleeper` живёт, пока выполняется функция `main`. Поток `t` получает для работы функциональный объект, в который завёрнута ссылка на переменную `valSleeper`, и число 5 (строка 25). Главное в этом примере то, что поток `t` получает переменную `valSleeper` по ссылке (строка 9) и отсоединяется от главного потока (строка 26). В дочернем потоке выполняется перегруженная операция вызова функционального объекта (строки 10–15). В этой функции цикл увеличивает счётчик от 0 до 5 включительно, на каждой итерации ждёт 1/10 секунды и увеличивает переменную `i` на величину `k`. В конце дочерний поток должен напечатать свой идентификатор. Считая, как говорят в Германии, «по Адаму Ризе»[1], результат должен составлять

1000 + 6 * 5 = 1030.

Однако как же программа работает на самом деле? Что-то в ней идёт не так.



Неопределённое поведение при передаче аргумента по ссылке

Программа обнаруживает два странных свойства. Во-первых, значение переменной `valSleeper` в конце оказывается равным 1000, а во-вторых, идентификатор дочернего потока на печать не выводится. Программа страдает по меньшей мере двумя изъянами.

1. Разные потоки имеют ничем не ограниченный доступ к переменной `valSleeper`. Здесь имеет место гонка данных, поскольку потоки могут одновременно производить чтение и запись этой переменной.
2. Время жизни главного потока завершается раньше, чем дочерний поток успевает выполнить все свои вычисления и вывести свой идентификатор в поток[2] `std::cout`.

---

[1] Адам Ризе (1492–1559) – выдающийся германский математик и учитель арифметики, учебники которого, издававшиеся на немецком языке, пользовались столь широкой популярностью, что имя автора вошло в поговорку. Выражение «по Адаму Ризе» в применении к каким-либо подсчётам означает «с высочайшей точностью». – *Прим. перев.*

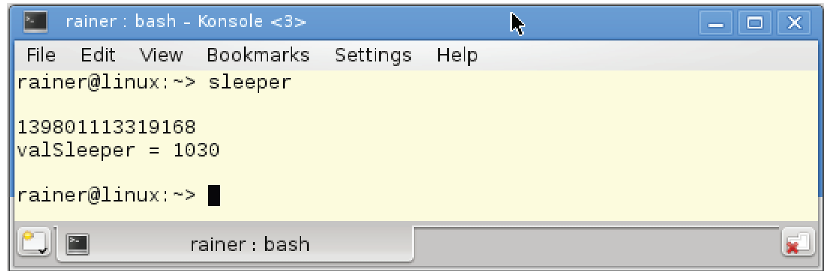
[2] Напомним, что в русской терминологии английским словам «thread» и «stream» соответствует один термин: «поток». В первом случае речь идёт о составной части параллельной программы, а во втором – о последовательности байтов или символов, связанной с устройством ввода-вывода. Смысл термина «поток» нужно определять из контекста. – *Прим. перев.*

Оба названных обстоятельства переводят программу в состояние гонки, поскольку результат работы программы зависит от относительной скорости выполнения операций в разных потоках. Состояние гонок выступает причиной гонки данных.

Исправить гонку данных довольно просто. Для этого достаточно защитить переменную `valSleeper` мьютексом или сделать её атомарной. Чтобы преодолеть ошибку, связанную со временем жизни объектов `valSleeper` и `std::cout`, нужно не отсоединять дочерний поток, а, напротив, присоединить его выполнение (функция `join`). Ниже представлена исправленная функция `main`.

```
int main(){
  std::cout << std::endl;

  int valSleeper= 1000;
  std::thread t(Sleeper(valSleeper),5);
  t.join();
  std::cout << "valSleeper = " << valSleeper << std::endl;

  std::cout << std::endl;
}
```

Теперь программа выдаёт правильный результат, хотя, конечно, и выполняется медленнее.



Исправленное неопределённое поведение

Завершая рассказ о классе `std::thread`, приведём полный список функций-членов.

3.1.4. Перечень функций-членов

Интерфейс класса `std::thread` показан ниже в виде таблицы. Более подробные сведения можно найти на сайте cppreference.com[1].

[1] http://de.cppreference.com/w/cpp/thread/thread.

Функции-члены класса std::thread

Функция	Описание
`join`	Ждать завершения потока
`detach`	Разорвать связь с потоком и оставить его выполняться независимо
`joinable`	Узнать, находится ли поток в присоединяемом состоянии
`get_id`, `std::this_thread::get_id`	Узнать идентификатор потока
`hardware_concurrency`	Узнать, сколько потоков может выполняться параллельно
`std::this_thread::sleep_until`	Погрузить поток в сон до определённого абсолютного момента времени
`std::this_thread::sleep_for`	Погрузить поток в сон на заданное время относительно текущего момента
`std::this_thread::yield`	Уступить выполнение другому потоку
`t1.swap(t2)` `std::swap(t1, t2)`	Обменять местами два объекта

Доступ к системно-зависимой реализации

Классы из стандартной библиотеки представляют собой обёртку над реализацией потоков в конкретной платформе. Для доступа к системному дескриптору потока можно воспользоваться функцией `native_handle`. Такие системные дескрипторы можно получать для потоков, мьютексов и переменных условия.

Объекты класса `std::thread` нельзя копировать, но можно перемещать. Функция `swap` обменивает местами содержимое двух объектов, пользуясь для этого перемещением.

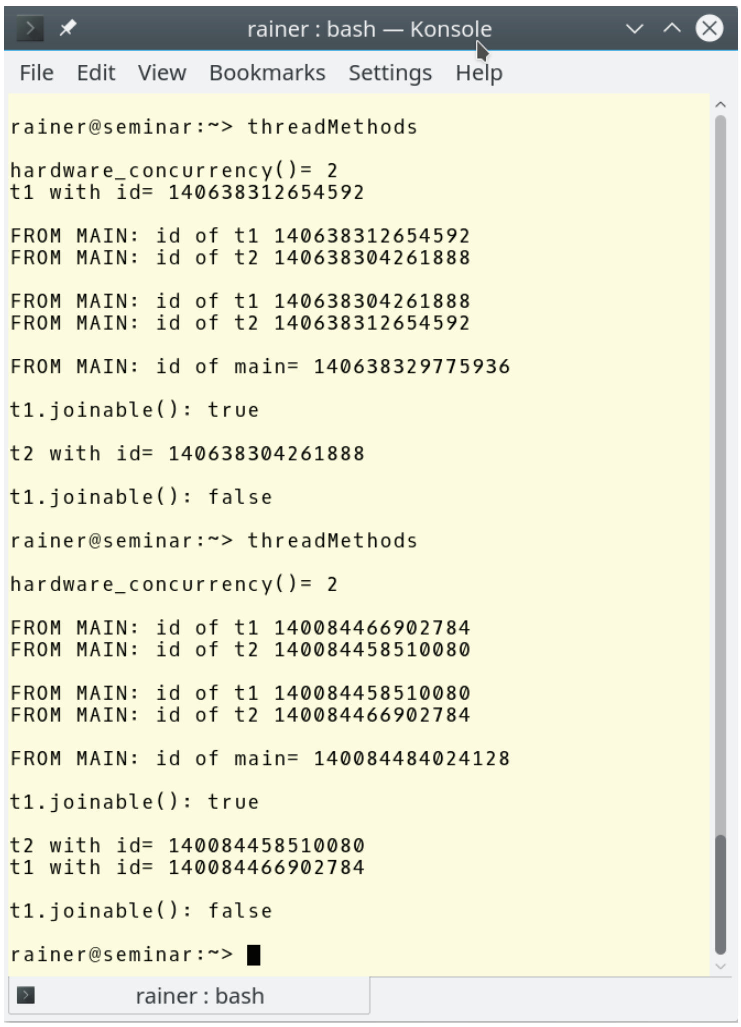
Завершим раздел примером применения некоторых из разобранных здесь функций-членов.

Функции-члены класса std::thread

```
1  // threadMethods.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  using namespace std;
7
8  int main(){
```

```
9     cout << boolalpha << endl;
10
11    cout << "hardware_concurrency()= "<< thread::hardware_concurrency() << endl;
12
13    thread t1([]{cout << "t1 with id= " << this_thread::get_id() << endl;});
14    thread t2([]{cout << "t2 with id= " << this_thread::get_id() << endl;});
15
16    cout << endl;
17
18    cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
19    cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;
20
21    cout << endl;
22    swap(t1,t2);
23
24    cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
25    cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;
26
27    cout << endl;
28
29    cout << "FROM MAIN: id of main= " << this_thread::get_id() << endl;
30
31    cout << endl;
32
33    cout << "t1.joinable(): " << t1.joinable() << endl;
34
35    cout << endl;
36
37    t1.join();
38    t2.join();
39
40    cout << endl;
41
42    cout << "t1.joinable(): " << t1.joinable() << endl;
43
44    cout << endl;
45  }
```

Эту программу должно быть нетрудно понять, особенно если держать перед глазами результат её работы.



Функции-члены класса `std::thread`

Может показаться странным, что потоки `t1` и `t2` (строки 13 и 14) отрабатывают в столь различные моменты времени. Однако это нормально: у программиста нет никаких гарантий относительно того, когда и с какой скоростью выполняются потоки. С уверенностью можно сказать лишь, что выполнение потоков завершится к моменту завершения функции `join`, вызванной в строках 37 и 38.

Управление параллельными потоками тем труднее, чем больше изменяемых (т. е. неконстантных) данных находится у них в общем доступе.

3.2. Усовершенствованные потоки: класс `std::jthread` (стандарт C++ 20)

Название класса `std::jthread` означает присоединяющийся поток (joining thread). В дополнение к функциональности класса `std::thread` из стандарта C++ 11 этот новый класс автоматически вызывает функцию `join` в своём деструкторе и поддерживает прерывание потока на кооперативной основе. Для этого класс `std::jthread` расширяет интерфейс класса `std::thread`.

В следующей таблице представлен обзор функциональных возможностей, добавленных в классе `std::jthread`.

Дополнительные функции-члены класса std::jthread

Имя	Описание
`get_stop_source`	Возвращает объект типа `std::stop_source`, связанный с флагом завершения
`get_stop_token`	Возвращает объект типа `std::stop_token`, связанный с флагом завершения
`request_stop`	Запрашивает завершение выполнения через флаг

3.2.1. Автоматическое присоединение к потоку

Поведение класса `std::thread` противоречит ожиданиям: если на момент вызова деструктора поток находится в присоединяемом состоянии, выполнение всей программы аварийно завершается вызовом `std::terminate`. Напомним, что поток находится в присоединяемом состоянии, если он был запущен и для него не вызывалась ни одна из функций `join` или `detach`.

Аварийно завершающаяся программа с присоединяемым потоком

```
// threadJoinable.cpp

#include <iostream>
#include <thread>

int main() {
    std::cout << std::endl;
    std::cout << std::boolalpha;
    std::thread thr{[]{ std::cout << "Joinable std::thread" << std::endl; }};
    std::cout << "thr.joinable(): " << thr.joinable() << std::endl;
    std::cout << std::endl;
}
```

Если эту программу запустить, она завершится с ошибкой.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@linux:~> threadJoinable

thr.joinable(): true

terminate called without an active exception
Aborted (core dumped)
rainer@linux:~> threadJoinable

thr.joinable(): true

terminate called without an active exception
Joinable std::thread
Aborted (core dumped)
rainer@linux:~>
rainer : bash
```

Аварийно завершающаяся программа с присоединяемым потоком

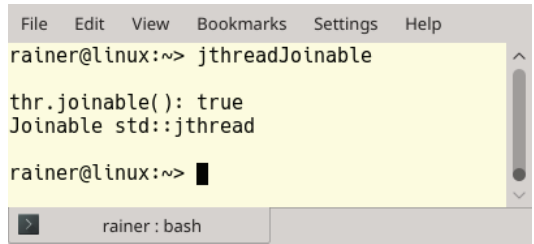
Оба запуска программы завершаются с ошибкой. При этом во второй раз у потока `thr` оказывается вдоволь времени, чтобы вывести своё сообщение: «Joinable std::thread».

Следующий код во всём подобен предыдущему, за исключением замены класса `std::thread` классом `std::jthread` из стандарта C++ 20.

Завершение потока `std::jthread` в присоединяемом состоянии

```
// jthreadJoinable.cpp

#include <iostream>
#include <thread>

int main() {
    std::cout << std::endl;
    std::cout << std::boolalpha;
    std::thread thr{[]{ std::cout << "Joinable std::thread" << std::endl; }};
    std::cout << "thr.joinable(): " << thr.joinable() << std::endl;
    std::cout << std::endl;
}
```

Теперь выполнение дочернего потока автоматически присоединяется к главному потоку при деструкции объекта `thr`, поскольку в момент выхода за область видимости он всё ещё находится в присоединяемом состоянии.



Автоматическое присоединение к потоку

Ниже показано, как мог бы быть реализован деструктор класса `std::jthread`.

Деструктор класса `std::jthread`

```
1  jthread::~jthread() {
2      if(joinable()) {
3          request_stop();
4          join();
5      }
6  }
```

В первую очередь деструктор проверяет, по-прежнему ли поток допускает присоединение (строка 2). Поток находится в присоединяемом состоянии, если он был запущен и для него не вызывалась ни одна из функций `join` или `detach`. Если присоединение имеет смысл, деструктор запрашивает завершение потока (строка 3) и затем вызывает функцию `join` (строка 4). Этот вызов блокирует вызывающий поток до тех пор, пока не закончится выполнение дочернего потока.

3.2.2. Прерывание по запросу в классе `std::jthread`

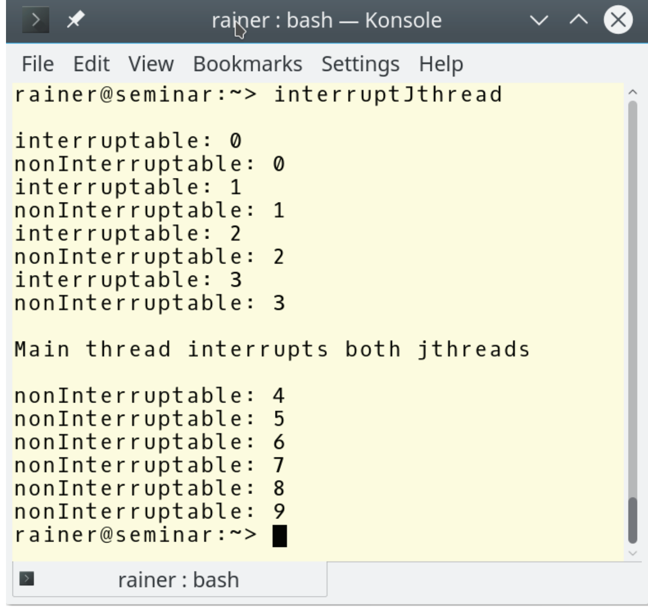
Чтобы проиллюстрировать основную идею, начнём с простого примера.

Прерываемый и непрерываемый потоки

```
1  // interruptJthread.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  using namespace::std::literals;
8
9  int main() {
10     std::cout << ‘\n’;
11
12     std::jthread nonInterruptable([]{
13         int counter{0};
14         while (counter < 10){
15             std::this_thread::sleep_for(0.2s);
16             std::cerr << “nonInterruptable: “ << counter << ‘\n’;
17             ++counter;
18         }
19     });
20
21     std::jthread interruptable([](std::stop_token stoken){
22         int counter{0};
23         while (counter < 10){
24             std::this_thread::sleep_for(0.2s);
```

```
25              if (stoken.stop_requested()) return;
26              std::cerr << "interruptable: " << counter << '\n';
27              ++counter;
28          }
29      });
30
31      std::this_thread::sleep_for(1s);
32
33      std::cerr << '\n';
34      std::cerr << "Main thread interrupts both jthreads" << '\n';
35      nonInterruptable.request_stop();
36      interruptable.request_stop();
37
38      std::cout << '\n';
39  }
```

В главном потоке запускаются два дочерних потока: `nonInterruptible` (с англ. «непрерываемый») в строке 12 и `interruptible` (соотв. «прерываемый») в строке 21. В отличие от непрерываемого потока, прерываемый получает объект `std::stop_token` (флаг завершения) и использует его в строке 25 для проверки того, не запросил ли кто-либо завершение потока (для этого служит функция-член `stop_requested`). Если завершение запрошено, лямбда-функция возвращает управление, и тем самым завершается работа потока. Вызов функции `request_stop` в строке 36 запрашивает завершение потока. Однако на первый поток `nonInterruptible` этот запрос не действует: поток просто не реагирует на него.



Прерываемый и непрерываемый потоки

Подробнее о типе `std::stop_token` будет рассказано в разделе 3.6.

3.3. Данные в совместном доступе

Нужно чётко понимать, что усилия по синхронизации нужны только тогда, когда имеются изменяемые данные, доступные нескольким потокам, поскольку именно в этом случае становится возможной гонка данных. Если в программе происходят одновременные несинхронизированные операции чтения и записи данных, программа в целом обладает неопределённым поведением.

Самый простой способ воочию увидеть результат одновременных несинхронизированных модификаций общего объекта – это из нескольких потоков отправлять текст в поток вывода `std::cout`.

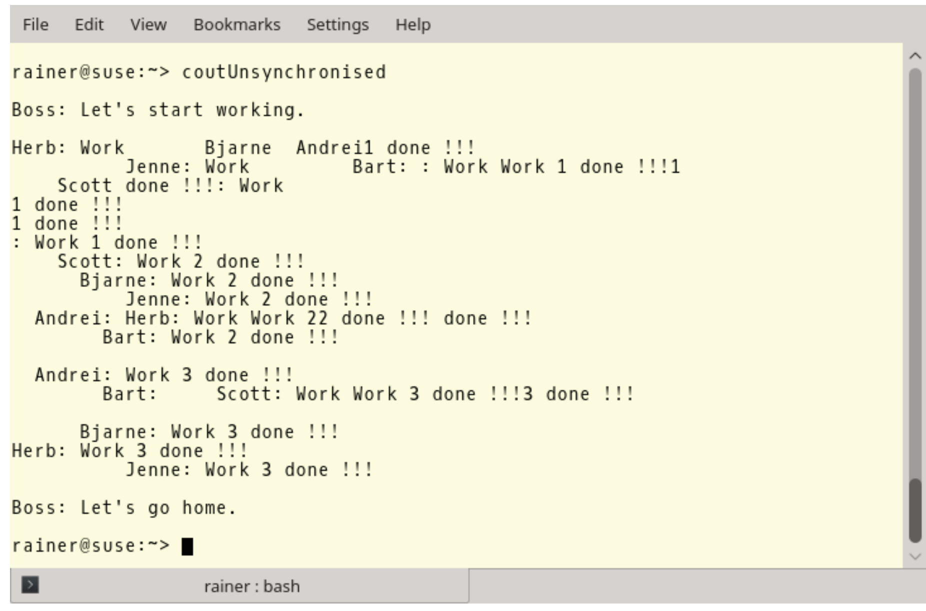
Несинхронизированный вывод в поток `std::cout`

```
1  // coutUnsynchronised.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  class Worker{
8  public:
9    Worker(std::string n):name(n){};
10     void operator() (){
11       for (int i = 1; i <= 3; ++i){
12         // begin work
13         std::this_thread::sleep_for(std::chrono::milliseconds(200));
14         // end work
15         std::cout << name << ": " << "Work " << i << " done !!!" << std::endl;
16       }
17     }
18 private:
19   std::string name;
20 };
21
22 int main(){
23   std::cout << std::endl;
24
25   std::cout << "Boss: Let's start working.\n\n";
26
27   std::thread herb= std::thread(Worker("Herb"));
28   std::thread andrei= std::thread(Worker("  Andrei"));
29   std::thread scott= std::thread(Worker("    Scott"));
30   std::thread bjarne= std::thread(Worker("      Bjarne"));
31   std::thread bart= std::thread(Worker("        Bart"));
32   std::thread jenne= std::thread(Worker("          Jenne"));
33
34   herb.join();
```

```
35    andrei.join();
36    scott.join();
37    bjarne.join();
38    bart.join();
39    jenne.join();
40
41    std::cout << "\n" << "Boss: Let's go home." << std::endl;
42
43    std::cout << std::endl;
44  }
```

Эта программа моделирует рабочий процесс. У начальника есть шесть работников (строки 27–32). Каждому работнику нужно выполнить три задания. Каждое задание занимает 1/5 секунды (строка 13). Закончив задание, работник громко сообщает об этом начальнику (строка 15). Когда начальник получит отчёт от всех работников, он отпускает всех по домам (строка 41).

Сколько же путаницы возникает в столь простой организации!



Несинхронизированный вывод в поток **std::cout**

Самое очевидное решение – воспользоваться мьютексом.

3.3.1. Мьютексы

Мьютекс (англ. *mutex*) сокращённо означает «mutual exclusion» – взаимное исключение. Это примитив синхронизации, который гарантирует, что не более одного потока находится в критической секции в каждый момент времени.

С появлением мьютекса хаос в рабочем коллективе сменяется гармонией.

Синхронизированный вывод в поток std::cout

```
1  // coutSynchronised.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex coutMutex;
9
10 class Worker{
11 public:
12   Worker(std::string n):name(n){};
13
14     void operator() (){
15       for (int i = 1; i <= 3; ++i){
16         // begin work
17         std::this_thread::sleep_for(std::chrono::milliseconds(200));
18         // end work
19         coutMutex.lock();
20         std::cout << name << ": " << "Work " << i << " done !!!" << std::endl;
21         coutMutex.unlock();
22       }
23     }
24 private:
25   std::string name;
26 };
27
28 int main(){
29   std::cout << std::endl;
30
31   std::cout << "Boss: Let's start working." << "\n\n";
32
33   std::thread herb= std::thread(Worker("Herb"));
34   std::thread andrei= std::thread(Worker("  Andrei"));
35   std::thread scott= std::thread(Worker("    Scott"));
36   std::thread bjarne= std::thread(Worker("      Bjarne"));
37   std::thread bart= std::thread(Worker("        Bart"));
38   std::thread jenne= std::thread(Worker("          Jenne"));
39
40   herb.join();
41   andrei.join();
42   scott.join();
43   bjarne.join();
44   bart.join();
45   jenne.join();
46
47   std::cout << "\n" << "Boss: Let's go home." << std::endl;
48
49   std::cout << std::endl;
50 }
```

Теперь поток `std::cout` защищён мьютексом `coutMutex`, объявленным в строке 8. Запирание мьютекса функцией `lock` в строке 19 и отпирание функцией

unlock в строке 21 гарантируют, что работники докладывают о завершении заданий не хором, а по одному.

```
File   Edit   View   Bookmarks   Settings   Help

rainer@suse:~> coutSynchronised

Boss: Let's start working.

Herb: Work 1 done !!!
        Bart: Work 1 done !!!
      Bjarne: Work 1 done !!!
          Jenne: Work 1 done !!!
  Andrei: Work 1 done !!!
    Scott: Work 1 done !!!
        Bart: Work 2 done !!!
Herb: Work 2 done !!!
          Jenne: Work 2 done !!!
  Andrei: Work 2 done !!!
    Scott: Work 2 done !!!
      Bjarne: Work 2 done !!!
        Bart: Work 3 done !!!
Herb: Work 3 done !!!
      Bjarne: Work 3 done !!!
          Jenne: Work 3 done !!!
    Scott: Work 3 done !!!
  Andrei: Work 3 done !!!

Boss: Let's go home.

rainer@suse:~>

rainer : bash
```

Синхронизированный вывод в поток std::cout

Потокобезопасность объекта std::cout

Стандарт C++ 11 гарантирует, что защищать объект std::cout от одновременного доступа из нескольких потоков не нужно. Каждое значение выводится атомарным образом. Однако между соседними операциями вывода может вклиниваться вывод из других потоков. Путаница, однако, здесь исключительно визуальная: поведение программы в целом хорошо определено. Это справедливо для всех глобальных потоков ввода-вывода (std::cout, std::cin, std::cerr, std::clog): отправка данных в эти потоки и получение данных из них потокобезопасны.
Говоря более формальным языком, одновременная запись в поток std::cout из нескольких потоков представляет собой состояние гонки, но не гонку данных. Это означает, что результат вывода программы зависит от порядка выполнения потоков.

В стандарте C++ 11 определены четыре различных мьютекса, которые могут работать рекурсивно, допускать обработку неудач, а также ограничивать время ожидания.

Виды мьютексов в стандартной библиотеке

Функция	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
lock	да	да	да	да
try_lock	да	да	да	да
try_lock_for			да	да
try_lock_until			да	да
unlock	да	да	да	да

Рекурсивный мьютекс (тип `std::recursive_mutex`) можно многократно запирать из одного и того же потока. Мьютекс останется в запертом состоянии до тех пор, пока он не будет открыт ровно столько же раз, сколько и заперт. Максимальное количество раз, которое можно запирать рекурсивный мьютекс, стандартом не определено. Если этот максимум достигнут, выбрасывается исключение `std::system_error`[1].

В стандарте C++ 14 появился тип `std::shared_timed_mutex`, а в стандарте C++ 17 – тип `std::shared_mutex`. Эти типы весьма похожи между собой. Оба могут использоваться как для исключительной, так и для совместной (англ. *shared*) блокировки. Кроме того, тип `std::shared_timed_mutex` позволяет задать предельный момент времени или предельный срок ожидания.

Виды мьютексов с совместной блокировкой

Функция	shared_timed_mutex	shared_ mutex
lock	да	да
try_lock	да	да
try_lock_for	да	
try_lock_until	да	
unlock	да	да
lock_shared	да	да
try_lock_shared	да	да
try_lock_shared_for	да	
try_lock_shared_until	да	
unlock_shared	да	да

Тип `std::shared_timed_mutex` (или `std::shared_mutex`) позволяет реализовать шаблон «читателей и писателей»[2] благодаря наличию двух режимов

[1] http://en.cppreference.com/w/cpp/error/system_error.

[2] Поясним смысл этого часто применяемого на практике шаблона. К данным, находящимся в общем доступе, потоки могут обращаться либо с целью чтения, либо с целью записи. Очевидно, что читать общие данные может одновременно сколь угодно много потоков – при условии что ни один поток в это время не выполняет их запись. С другой стороны, писать данные может только один поток – и лишь в случае, если ни один поток не осуществляет чтение. Таким образом, в любой момент времени должно соблюдаться условие: с данными работает либо один писатель и ни одного читателя, либо ни одного писателя и любое число читателей. Обычно это ограничение дополняют следующим. Если какой-либо поток изъявил желание модифицировать общие данные, он должен получить такую возможность независимо от прихода новых читателей. Это означает, что если один или несколько потоков стоят в очереди на право стать писателями и какой-либо поток желает получить доступ на чтение, он вынужден ждать, пока не закончится обслуживание всех писателей. Иными словами, если поток подаёт заявку на запись, пока данными пользуются несколько читателей, заход новых читателей блокируется; имеющиеся читатели рано или поздно заканчивают свою работу и покидают общие данные; с уходом последнего читателя начинается обслуживание писателей по одному; все потоки, желающие получить доступ на чтение, отправляются в ожидание; наконец, с уходом последнего писателя все такие потоки одновременно приступают к чтению. Легко понять, что доступ в режиме писателя – исключительный, а в режиме читателя – совместный. Они напрямую соответствуют режимам

блокировки: исключительного и совместного. Мьютекс захватывается в исключительном режиме, если к объекту применяется блокировщик типа `std::lock_guard` или `std::unique_lock`, тогда как блокировщик типа `std::shared_lock` захватывает объект в совместном режиме. Функции-члены `try_lock_for` и `try_lock_shared_for` в качестве аргумента принимают промежуток времени, по истечении которого попытка захвата завершается неудачей. Функции же `try_lock_until` и `try_lock_shared_until` принимают абсолютное значение – момент времени, до которого можно ожидать.

Функции `try_lock` и `try_lock_shared` пытаются захватить мьютекс и немедленно возвращают управление. Если захват удался, они возвращают значение `true`, а если мьютекс уже захвачен другими потоками – значение `false`. В отличие от них, функции `try_lock_for`, `try_lock_shared_for`, `try_lock_until` и `try_lock_shared_until` блокируют поток до тех пор, пока мьютекс не станет доступен либо пока не истечёт предельное время ожидания. Задавать предельное время ожидания следует с использованием монотонных часов (`std::chrono::steady_clock`), которые гарантируют неубывающую последовательность показаний.

Мьютексы не стоит использовать в программе напрямую, вместо этого рекомендуется оборачивать их в объекты-блокировщики. В следующем разделе подробно рассматриваются причины этого.

3.3.1.1. Затруднения с мьютексами

Большая часть трудностей, возникающих при использовании мьютексов, сводится к одной главной проблеме – мёртвой блокировке[1].

> **Мёртвая блокировка** – *это состояние, в котором каждый из двух или более потоков заблокирован в ожидании ресурса, занятого другим потоком, и до своей разблокировки не может освободить ресурс, которого ожидает другой поток.*

Результат мёртвой блокировки – полная остановка работы потоков. Все потоки, вовлечённые в мёртвую блокировку, а часто и вся программа, блокируются навечно. Создать такую ситуацию очень просто. Посмотрим, как это может произойти.

3.3.3.1.1. Исключения и неизвестный код

В следующем крошечном фрагменте кода таится множество подводных камней.

```
std::mutex m;
m.lock();
```

работы мьютекса `std::shared_mutex`. Следует отметить, что данный вид синхронизации доступа к данным можно реализовать средствами стандарта C++ 11, где тип `std::shared_timed_mutex` ещё отсутствовал, а именно – через обычный мьютекс и переменную условия. Читателю стоит построить такую реализацию в качестве упражнения. – *Прим. перев.*

[1] В русскоязычной литературе можно встретить ряд других терминов: взаимоблокировка, тупик, дедлок (от англ. *deadlock*). – *Прим. перев.*

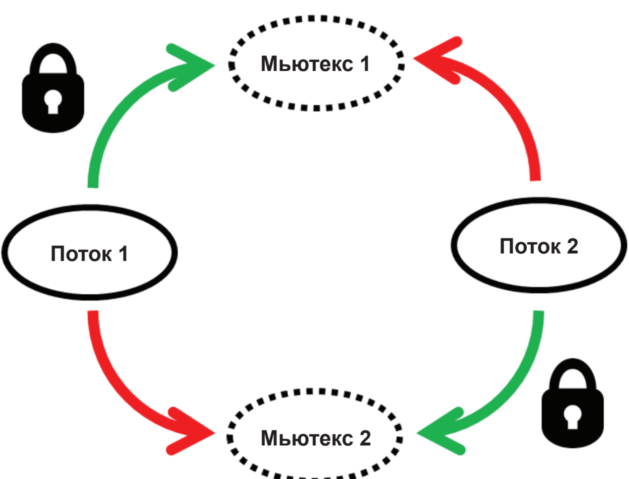
```
sharedVariable = getVar();
m.unlock();
```

1. Если функция `getVar` выбросит исключение, мьютекс `m` никогда не будет освобождён.
2. Никогда, ни в коем случае нельзя вызывать из-под мьютекса функцию, внутреннее устройство которой неизвестно. Если функция `getVar` пытается захватить мьютекс `m`, поведение программы не определено, так как этот мьютекс – не рекурсивный. В большинстве подобных случаев неопределённое поведение выражается в мёртвой блокировке.
3. Вызывать функции из-под блокировки опасно ещё по одной причине. Если функция определена в сторонней библиотеке, в новой её версии реализация и поведение функции могут измениться. Даже если первоначально опасности мёртвой блокировки не было, она может появиться в будущем.

Чем больше в программе блокировок, тем труднее становится уследить за всеми, и эта зависимость весьма нелинейна.

3.3.3.1.2. Захват мьютексов в различном порядке

На следующем рисунке показан типичный пример попадания в мёртвую блокировку по причине захвата мьютексов в различном порядке.



Мёртвая блокировка двух потоков

Каждому из двух потоков требуется для работы доступ к двум ресурсам, для защиты которых используются два отдельных мьютекса. Проблема возникает, когда потоки захватывают их в разном порядке: поток 1 захватывает сначала 1-й мьютекс, затем 2-й, а поток 2 – наоборот: сначала мьютекс 1, потом мьютекс 2. Тогда операции двух потоков перемежаются следующим образом: сначала поток 1 захватывает мьютекс 1, а поток 2 – мьютекс 2,

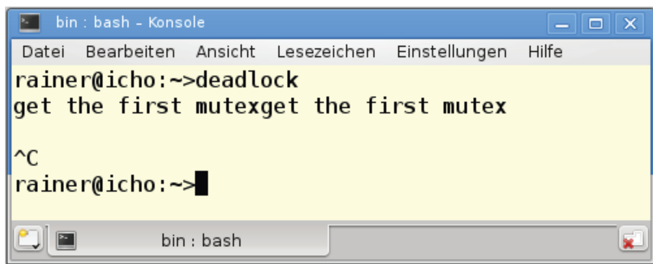
а затем наступает взаимная блокировка. Каждый из двух потоков хочет захватить мьютекс, захваченный другим потоком, но этот последний не может отпустить мьютекс, так как сам заблокирован. Для этой ситуации хорошо подходит выражение «смертельные объятия».

Превратить эту картинку в код довольно просто.

Захват мьютексов в различном порядке

```
1  // deadlock.cpp
2
3  #include <iostream>
4  #include <chrono>
5  #include <mutex>
6  #include <thread>
7
8  struct CriticalData{
9    std::mutex mut;
10 };
11
12 void deadLock(CriticalData& a, CriticalData& b){
13   a.mut.lock();
14   std::cout << "get the first mutex" << std::endl;
15   std::this_thread::sleep_for(std::chrono::milliseconds(1));
16   b.mut.lock();
17   std::cout << "get the second mutex" << std::endl;
18   // do something with a and b
19   a.mut.unlock();
20   b.mut.unlock();
21 }
22
23 int main(){
24   CriticalData c1;
25   CriticalData c2;
26
27   std::thread t1([&]{deadLock(c1,c2);});
28   std::thread t2([&]{deadLock(c2,c1);});
29
30   t1.join();
31   t2.join();
32 }
```

В потоках `t1` и `t2` выполняется функция `deadlock` (строки 12–21). Эта функция получает ссылки на переменные `c1` и `c2` типа `CriticalData` (строки 24 и 25). Поскольку эти объекты нуждаются в защите от одновременного доступа, каждый из них содержит внутри мьютекс (для простоты кода класс `CriticalData` никаких других данных или функций-членов не содержит). Задержки всего на одну миллисекунду в строке 15 оказывается достаточно, чтобы вызвать мёртвую блокировку.



Мёртвая блокировка двух потоков

Единственный способ сдвинуть программу с мёртвой точки – это прервать её выполнение, нажав **CTRL+C**.

Объекты-блокировщики позволяют справиться пусть и не со всеми трудностями, но в большинстве случаев оказываются полезны.

3.3.2. Блокировщики

Блокировщики управляют захватом и освобождением ресурса посредством идиомы RAII. Блокировщик автоматически захватывает мьютекс в конструкторе и освобождает в деструкторе. Это существенно снижает риск мёртвых блокировок, поскольку освобождение мьютекса гарантируется при любом завершении захватившего участка кода.

В стандарте имеется четыре вида мьютексов. Тип `std::lock_guard` предназначен для простейших сценариев[1], тип `std::unique_lock` – для более сложных[2]. Тип `std::shared_lock`, появившийся в стандарте C++ 14, можно использовать для блокировки читателей и писателей[3]. С выходом стандарта C++ 17 в руках у программистов оказался также блокировщик `std::scoped_lock`, который умеет запирать несколько мьютексов за одну атомарную операцию.

Начнём с разбора наиболее простого сценария.

3.3.2.1. Тип std::lock_guard

Посмотрим снова на знакомый фрагмент кода:

```
std::mutex m;
m.lock();
```

---

[1] Однократный захват мьютекса при создании блокировщика и однократное освобождение при уничтожении. – *Прим. перев.*

[2] При создании блокировщика можно с помощью специального параметра отложить захват мьютекса; за время жизни блокировщика мьютекс можно многократно открывать и запирать; освобождение мьютекса в деструкторе гарантируется. – *Прим. перев.*

[3] Блокировщик `std::shared_lock`, который может работать с мьютексами `std::shared_mutex` и `std::shared_timed_mutex`, захватывает их в совместном режиме, тогда как для захвата в исключительном режиме нужно пользоваться блокировщиком `std::lock_guard` или `std::unique_lock`.

```
sharedVariable = getVar();
m.unlock();
```

Мьютекс m должен гарантировать, что доступ к критической секции (строке кода, где присваивается значение переменной sharedVariable) происходит последовательно. Иными словами, если несколько потоков одновременно подходят к этой критической секции, они будут заходить в неё по одному. Тем самым устанавливается глобальный для системы порядок доступа к общим данным. Этот код выглядит просто, но потенциально подвержен мёртвой блокировке, например если изнутри критической секции выбрасывается исключение или если программист забывает разблокировать мьютекс функцией unlock. Блокировщик std::lock_guard позволяет решить данную задачу более элегантно:

```
{
    std::mutex m;
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable = getVar();
}
```

Код выглядит просто. Однако зачем в нём появились открывающая и закрывающая фигурные скобки? Они ограничивают область видимости и время жизни объекта-блокировщика, объявленного внутри этих скобок[1]. Это означает, что как только выполнение достигает закрывающей скобки, время жизни блокировщика заканчивается, отрабатывает его деструктор и – как нетрудно догадаться – освобождает мьютекс. Это происходит совершенно автоматически, даже если функция getVar изнутри критической секции выбрасывает исключение. Тело функции и тело цикла также представляют собой блоки, ограничивающие время жизни объекта.

3.3.2.2. Тип std::scoped_lock

В стандарте C++ 17 библиотека пополнилась типом std::scoped_lock. Он очень похож на тип std::lock_guard, но позволяет управлять одновременно произвольным количеством мьютексов. Нужно иметь в виду несколько обстоятельств.

1. Если в блокировщик std::scoped_lock завернуть только один мьютекс, блокировщик ведёт себя в точности как тип std::lock_guard и в конструкторе вызывает функцию lock мьютекса. Если же в конструктор блокировщика передать несколько мьютексов, для них вызывается функция std::lock.
2. Если текущий поток уже захватил один из мьютексов и если этот мьютекс не рекурсивного типа, поведение не определено. На практике это с большой вероятностью может означать мёртвую блокировку.
3. Блокировщик позволяет также забрать себе управление уже захваченным мьютексом без попытки захватить его[2]. Для этого нужно пере-

[1] https://en.cppreference.com/w/cpp/language/scope#Block_scope.

[2] Эту возможность предоставляют также два других типа блокировщика: std::lock_guard и std::shared_lock. – *Прим. перев.*

дать в конструктор блокировщика[1] дополнительный аргумент типа `std::adopt_lock_t`.

Описанную выше трудность с мёртвой блокировкой можно элегантно решить, воспользовавшись типом `std::scoped_lock`. Это решение будет рассматриваться в следующем разделе.

3.3.2.3. Тип std::unique_lock

Тип блокировщика `std::unique_lock` превосходит возможностями своего младшего брата `std::lock_guard`, но и обходится дороже. В дополнение к возможностям, которые предоставляет тип `std::lock_guard`, тип `std::unique_lock` позволяет также:

- создавать блокировщик, не связанный с каким-либо мьютексом;
- создавать блокировщик, не блокируя переданный ему мьютекс;
- в явном виде многократно захватывать и освобождать мьютекс;
- запирать мьютекс рекурсивно;
- перемещать мьютекс в другой блокировщик;
- *пытаться* захватить мьютекс[2];
- задавать предельное время ожидания при попытке захвата мьютекса.

Функции-члены этого класса показаны в следующей таблице.

Интерфейс класса std::unique_lock

Функция	Описание
`lock`	Захватывает завёрнутый в объект мьютекс
`try_lock`	Пытается захватить мьютекс; если он уже захвачен, возвращает значение `false`
`try_lock_for`, `try_lock_until`	То же с предельным временем ожидания
`unlock`	Освобождает мьютекс
`release`	Отдаёт управление мьютексом, не освобождая его
`swap`	Обменивает мьютексы между двумя блокировщиками; аналог вызова `std::swap(lk1, lk2)`
`mutex`	Возвращает указатель на мьютекс, завёрнутый блокировщик
`owns_lock`, `operator bool`	Проверяет, занят ли мьютекс

Функции-члену `try_lock_for` требуется временной интервал относительно текущего момента, в течение которого можно ожидать занятый мьютекс; функция `try_lock_until` принимает абсолютное значение – момент времени. Функции `try_lock_for` и `try_lock_until` блокировщика вызывают соответствующую функцию мьютекса. При этом мьютекс должен сам под-

[1] Этот способ допустимо использовать тогда и только тогда, когда все мьютексы уже захвачены текущим потоком; в противном случае поведение не определено. Взятие под контроль уже запертого мьютекса подробнее рассматривается в конце раздела 3.3.2.3. – *Прим. перев.*

[2] Такая попытка заканчивается неудачей, если мьютекс уже захвачен. – *Прим. перев.*

держивать попытку захвата с предельным временем ожидания (см. раздел 3.3.1). При указании времени следует пользоваться монотонными часами (std::chrono::steady_clock), которые гарантируют неубывание показателей и которые невозможно переводить.

Функция-член try_lock пытается захватить мьютекс и возвращает управление немедленно. Если захват удался, функция возвращает значение true, в противном случае – false. В отличие от него, функции try_lock_for и try_lock_until в случае занятого мьютекса блокируют выполнение своего потока до тех пор, пока мьютекс не освободится или пока не истечёт предельное время ожидания. Все три функции бросают исключение std::system_error, если с объектом-блокировщиком не связан никакой мьютекс или если мьютекс уже захвачен этим блокировщиком.

Функция release разрывает связь объекта-блокировщика с завёрнутым в него мьютексом и возвращает указатель на этот мьютекс. Код, вызвавший эту функцию, должен отныне сам позаботиться об освобождении мьютекса.

3.3.2.4. Блокировщик std::shared_lock

В стандарте C++ 14 был добавлен блокировщик std::shared_lock. Он обладает таким же интерфейсом, как и тип std::unique_lock, но по-иному ведёт себя, когда применяется к объектам типов std::shared_mutex и std::shared_timed_mutex. Такие мьютексы могут захватываться (в совместном режиме) одновременно несколькими потоками, чем, в частности, реализуется шаблон читателей и писателей. Этот шаблон очень прост и исключительно полезен. Любое число потоков могут одновременно обращаться к критическим данным в режиме чтения, но модифицировать эти данные может только один поток[1].

Блокировка в режиме чтения и записи не устраняет корень проблемы – соперничество потоков за доступ к общим данным, – но помогает заметно расширить «бутылочное горлышко».

Хорошим примером использования блокировок на чтение и запись служит телефонная книга. Обычно в каждый момент времени множество людей желает искать в ней информацию, но лишь изредка кто-либо меняет её содержимое. Рассмотрим следующий код.

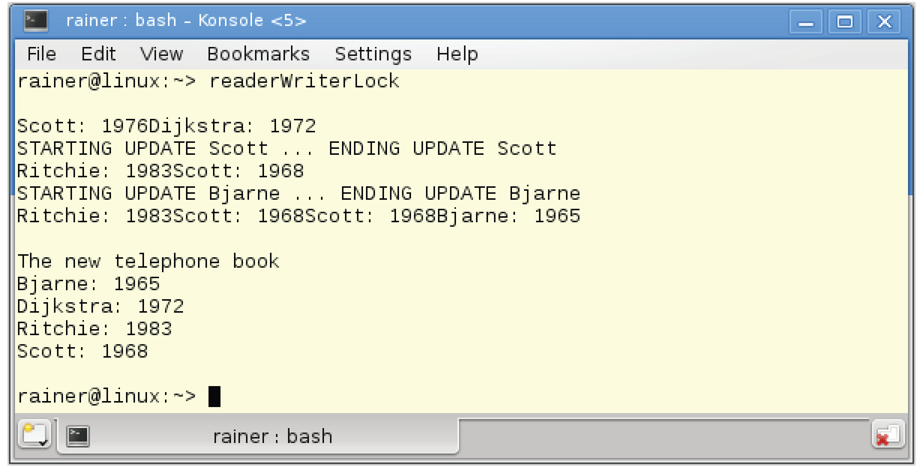
Блокировка на чтение и запись

```
1  // readerWriterLock.cpp
2
3  #include <iostream>
4  #include <map>
5  #include <shared_mutex>
6  #include <string>
7  #include <thread>
8
9  std::map<std::string,int> teleBook{
```

[1] См. также сноску 2 на с. 139. – *Прим. перев.*

```
10     {“Dijkstra”, 1972}, {“Scott”, 1976}, {“Ritchie”, 1983}};
11 
12 std::shared_timed_mutex teleBookMutex;
13 
14 void addToTeleBook(const std::string& na, int tele){
15   std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
16   std::cout << “\nSTARTING UPDATE “ << na;
17   std::this_thread::sleep_for(std::chrono::milliseconds(500));
18   teleBook[na]= tele;
19   std::cout << “ ... ENDING UPDATE “ << na << std::endl;
20 }
21 
22 void printNumber(const std::string& na){
23   std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
24   std::cout << na << “: “ << teleBook[na];
25 }
26 
27 int main(){
28   std::cout << std::endl;
29 
30   std::thread reader1([]{ printNumber(“Scott”); });
31   std::thread reader2([]{ printNumber(“Ritchie”); });
32   std::thread w1([]{ addToTeleBook(“Scott”,1968); });
33   std::thread reader3([]{ printNumber(“Dijkstra”); });
34   std::thread reader4([]{ printNumber(“Scott”); });
35   std::thread w2([]{ addToTeleBook(“Bjarne”,1965); });
36   std::thread reader5([]{ printNumber(“Scott”); });
37   std::thread reader6([]{ printNumber(“Ritchie”); });
38   std::thread reader7([]{ printNumber(“Scott”); });
39   std::thread reader8([]{ printNumber(“Bjarne”); });
40 
41   reader1.join();
42   reader2.join();
43   reader3.join();
44   reader4.join();
45   reader5.join();
46   reader6.join();
47   reader7.join();
48   reader8.join();
49   w1.join();
50   w2.join();
51 
52   std::cout << std::endl;
53 
54   std::cout << “\nThe new telephone book” << std::endl;
55   for (auto teleIt: teleBook){
56     std::cout << teleIt.first << “: “ << teleIt.second << std::endl;
57   }
58 
59   std::cout << std::endl;
60 }
```

Телефонная книга, объявленная в строке 9, находится в общем доступе у всех потоков, поэтому её нужно защищать от несогласованных операций. Восемь потоков изъявляют желание читать из телефонной книги, а два потока пытаются её изменить (строки 30–39). Для синхронизации параллельного доступа к телефонной книге потоки-читатели пользуются в строке 23 блокировщиком `std::shared_lock`. В отличие от них потоки-писатели требуют исключительного доступа к критическим данным. Исключительность достигается использованием блокировщика `std::lock_guard` в строке 15. В конце (строки 54–57) программа отображает обновлённое содержимое телефонной книги.



Доступ к телефонной книге с блокировкой на чтение и запись

Из результата работы программы видно, что вывод из потоков-читателей перемешивается, тогда как потоки-писатели выполняются по одному. Это означает, что операции с блокировкой только на чтение выполняются одновременно.

Решение оказалось удивительно простым. Слишком простым. И неправильным.

3.3.2.4.1. Как побороть неопределённое поведение

Представленная выше программа обладает неопределённым поведением. Говоря конкретнее, в ней присутствует гонка данных. Как это могло случиться? Прежде чем продолжать, читателю рекомендуется подумать несколько минут. Одновременный доступ к потоку `std::cout` здесь, кстати, ни при чём.

Определение гласит, что гонка данных – это ситуация, когда по меньшей мере два потока имеют одновременный доступ к некоторой переменной и хотя бы один из потоков производит её запись. Именно это может произойти в процессе выполнения представленной программы. Интересная особенность ассоциативного контейнера состоит в том, что чтение данных

из контейнера операцией индексирования может на самом деле изменить его. Это происходит, когда элемента с заданным ключом в контейнере не оказывается. Если, скажем, во время чтения в контейнере не обнаруживается элемент с ключом «Bjarne», пара («Bjarne», 0) создаётся и помещается в контейнер. Гонку данных легко сделать очевидной, если печать значения по ключу «Bjarne» поместить перед остальными потоками. Посмотрим, что получится. Первым сообщением программа выдаёт для ключа «Bjarne» значение 0.

```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> readerWriterLocks

Bjarne: 0Ritchie: 1983
STARTING UPDATE Scott ... ENDING UPDATE Scott

STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Ritchie: 1983Scott: 1968Scott: 1968Scott: 1968Dijkstra: 1972Scott: 1968

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@seminar:~>
rainer : bash
```

Возможность гонки данных

Очевидный способ исправить положение состоит в том, чтобы использовать в функции `printNumber` исключительно операции, не модифицирующие контейнер.

Исправленная программа с блокировкой на чтение и запись

```
1  // readerWriterLocksResolved.cpp
2  ...
3  void printNumber(const std::string& na){
4    std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
5    auto searchEntry = teleBook.find(na);
6    if(searchEntry != teleBook.end()){
7      std::cout << searchEntry->first << ": " << searchEntry->second << std::endl;
8    }
9    else {
10     std::cout << na << " not found!" << std::endl;
11   }
12 }
13 ...
```

Если запрошенного ключа в телефонной книге нет, эта программа печатает ключ и текст «not found!» («не найдено»).

```
File Edit View Bookmarks Settings Help
STARTING UPDATE Scott ... ENDING UPDATE Scott

STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Dijkstra: 1972
Ritchie: 1983
ScottScott: : 19681968
ScottScott: : 1968

Bjarne: 1965
1968
Ritchie: 1983


The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@suse:~> readerWriterLocksResolved

ScottRitchie: : 1983
1976
Scott: 1976Ritchie: 1983

DijkstraScott: : 1976
1972
Scott: 1976
Bjarne not found!

STARTING UPDATE Scott ... ENDING UPDATE Scott

STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne


The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@suse:~>
rainer : bash
```

Работа программы с исправленной гонкой данных

При втором запуске программа выдала сообщение «Bjarne not found». При первом запуске поток с добавлением записи в телефонную книгу успел выполниться первым, поэтому данные по ключу «Bjarne» нашлись.

3.3.3. Функция `std::lock`

Функция `std::lock` умеет захватывать сколь угодно много мьютексов, блокировщиков или иных объектов, подпадающих под понятие Lockable[1] (запираемый), за одно атомарное действие. Именованный набор требований Lockable объединяет типы, которые поддерживают функции-члены `lock`, `unlock` и `try_lock`. Функция `std::lock` представляет собой вариадический шаблон, т. е. может принимать любое число аргументов. Функция пытается захватить

[1] https://en.cppreference.com/w/cpp/named_req/Lockable.

все переданные ей запираемые объекты с помощью алгоритма, предотвращающего мёртвые блокировки. Объекты захватываются в неопределённой последовательности вызовом их функций-членов `lock` и `try_lock`. Если вызов операции `lock` для какого-либо мьютекса привёл к исключению, функция `std::lock` вызывает функцию-член `unlock` для всех объектов, которые успела захватить, а затем передаёт это исключение наружу.

Вспомним ещё раз программу `deadlock.cpp` из раздела 3.3.1.1.2 и преобразуем её так, чтобы исключить проблему. Для этого функция `deadLock` должна захватывать оба мьютекса одновременно – именно это делается в следующем примере.

Отложенный захват мьютексов

```
1  // deadlockResolved.cpp
2
3  #include <iostream>
4  #include <chrono>
5  #include <mutex>
6  #include <thread>
7
8  using namespace std;
9
10 struct CriticalData{
11   mutex mut;
12 };
13
14 void deadLock(CriticalData& a, CriticalData& b){
15   unique_lock<mutex> guard1(a.mut, defer_lock);
16   cout << “Thread: “ << this_thread::get_id() << “ first mutex” << endl;
17
18   this_thread::sleep_for(chrono::milliseconds(1));
19
20   unique_lock<mutex> guard2(b.mut, defer_lock);
21   cout << “  Thread: “ << this_thread::get_id() << “ second mutex” <<  endl;
22
23   cout << “    Thread: “ << this_thread::get_id() << “ get both mutexes” << endl;
24   lock(guard1,guard2);
25   // do something with a and b
26 }
27
28 int main(){
29   cout << endl;
30
31   CriticalData c1;
32   CriticalData c2;
33
34   thread t1([&]{deadLock(c1,c2);});
35   thread t2([&]{deadLock(c2,c1);});
36
37   t1.join();
38   t2.join();
```

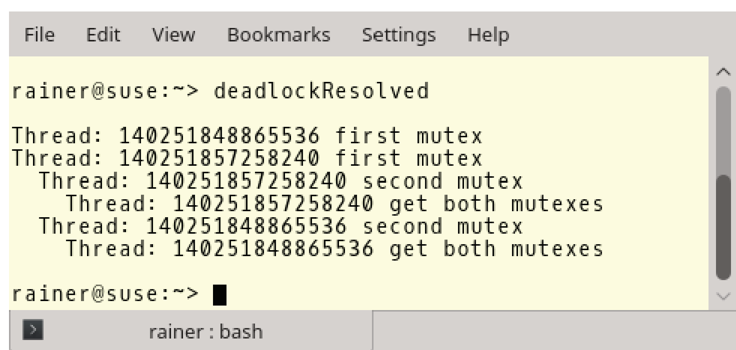
```
39
40   cout << endl;
41 }
```

При вызове конструктора std::unique_lock с параметром std::defer_lock мьютекс не запирается. В момент создания (строки 15 и 20) объекты std::unique_lock всего лишь становятся обёртками для своих мьютексов. Затем (строка 24) шаблонная функция std::lock с переменным числом аргументов запирает два мьютекса одновременно.

В этом примере объекты типа std::unique_lock служат для освобождения мьютексов при окончании времени жизни, а функция std::lock – для захвата этих объектов. Можно было бы сделать и наоборот: сначала захватить мьютексы функцией std::lock, затем взять их под управление блокировщиками std::unique_lock. Пример такого подхода показан ниже.

```
std::lock(a.mut, b.mut);
std::lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
std::lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
```

Оба способа устраняют мёртвую блокировку.



Устранение мёртвой блокировки
с помощью блокировщика std::unique_lock

Использование блокировщика std::scoped_lock

С появлением стандарта C++ 17 решение проблемы мёртвой блокировки становится совсем простым. Теперь есть класс std::scoped_lock, который позволяет захватывать любое число мьютексов в конструкторе. Всё, что нужно, – это просто создать объект типа std::scoped_lock. Ниже показана видоизменённая функция deadlock.

```
1  // deadlockResolvedScopedLock.cpp
2
3  ...
4  void deadLock(CriticalData& a, CriticalData& b){
5    cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;
6    this_thread::sleep_for(chrono::milliseconds(1));
7    cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  endl;
```

```
8
9    cout << "    Thread: " << this_thread::get_id() << " get both mutexes" << endl;
10   std::scoped_lock (a.mut, b.mut);
11   // do something with a and b
12 }
13 ...
```

3.3.4. Потокобезопасная инициализация

Если значение переменной никогда не изменяется, нет нужны синхронизировать доступ к ней с помощью дорогостоящих механизмов блокировки или даже атомарных переменных. Нужно лишь присвоить ей начальное значение потокобезопасным способом.

В языке C++ есть три способа потокобезопасной инициализации:

- константные выражения;
- функция `std::call_once` вместе с флагом `std::once_flag`;
- локальная статическая переменная.

Безопасная инициализация в главном потоке

Самый простой способ инициализировать переменную потокобезопасным образом – инициализировать её в главном потоке до создания всех остальных потоков.

3.3.4.1. Константные выражения

Константные выражения – это выражения, значение которых может быть вычислено на этапе компиляции. Их вычисление неявно потокобезопасно. Ключевое слово `constexpr`, помещённое в начало объявления переменной, делает её константным выражением. Такая переменная должна быть проинициализирована при объявлении, например:

```
constexpr double pi = 3.14;
```

Помимо встроенных в язык типов, константными выражениями могут быть и объекты пользовательских типов. Чтобы объекты можно было вычислять во время компиляции, типы должны удовлетворять некоторым ограничениям[1]:

---

[1] Стоит упомянуть ещё некоторые из перечисленных в стандарте ограничений. Функция `constexpr`, за исключением конструктора, должна содержать ровно один оператор `return`. Все параметры, возвращаемое значение (помимо конструктора) и локальные переменные такой функции должны иметь литеральный тип. В теле функции `constexpr` нельзя использовать операторы перехода `goto`, блоки `try` (до стандарта C++ 20), объявление переменной со статическим временем жизни или временем жизни потока. Конечно же, в теле функции `constexpr` нельзя вызывать функции, не являющиеся `constexpr`, – а таковы функции ввода-вывода и управления динамической памятью. Возможности спецификатора `constexpr` сильно расширены в стандарте C++ 20: появилось даже понятия деструктора `constexpr`. За многочисленными подробностями рекомендуется обратиться к справочнику. – *Прим. перев.*

- тип не должен содержать ни виртуальных функций-членов, ни виртуальных базовых классов;
- конструктор должен быть объявлен с ключевым словом `constexpr`;
- все базовые классы и все нестатические члены-данные должны быть проинициализированы;
- все функции-члены, которые предполагается вычислять на этапе компиляции, должны также быть объявлены с ключевым словом `constexpr`.

Представленный ниже класс `MyDouble` удовлетворяет всем перечисленным условиям. Поэтому его экземпляры можно создавать на этапе компиляции. Инициализация переменных `constexpr` этого типа потокобезопасна.

Вычисление констант пользовательского типа во время компиляции

```
1  // constexpr.cpp
2
3  #include <iostream>
4
5  class MyDouble{
6    private:
7      double myVal1;
8      double myVal2;
9    public:
10     constexpr MyDouble(double v1,double v2):myVal1(v1),myVal2(v2){}
11     constexpr double getSum() const { return myVal1 + myVal2; }
12 };
13
14 int main() {
15   constexpr double myStatVal = 2.0;
16   constexpr MyDouble myStatic(10.5, myStatVal);
17   constexpr double sumStat= myStatic.getSum();
18 }
```

3.3.4.2. Функция std::call_once и флаг std::once_flag

С помощью функции `std::call_once` можно зарегистрировать вызываемый объект, а с помощью флага `std::once_flag` – убедиться, что вызывается ровно один из них, и только один раз. На один флаг `std::once_flag` можно зарегистрировать сколько угодно вызываемых объектов (в частности, функций).

Функция `std::call_once` обладает следующими свойствами:

- выполняется только одна из зарегистрированных функций, и только один раз. Какая именно функция из множества зарегистрированных выбирается для выполнения, не определено. Выбранная функция выполняется в том же потоке, из которого вызвана функция `std::call_once`;
- из всех одновременных вызовов функции `std::call_once` ни один не завершается раньше, чем вызов выбранной для выполнения функции, о которой говорилось в предыдущем пункте;
- если выбранная для выполнения функция завершает работу с исключением, оно выбрасывается наружу из вызова функции `std::call_once`

в вызвавший контекст. В этом случае для выполнения выбирается другая функция[1].

Следующий короткий пример иллюстрирует применение функции `std::call_once` с флагом `std::once_flag`. И функция, и тип объявлены в заголовочном файле `<mutex>`.

Использование однократных действий

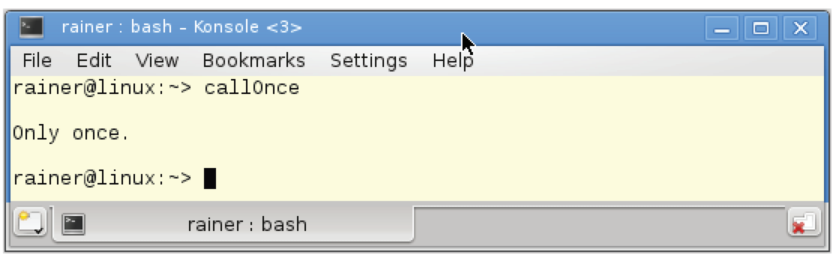
```
1  // callOnce.cpp
2
3  #include <iostream>
4  #include <thread>
5  #include <mutex>
6
7  std::once_flag onceFlag;
8
9  void do_once(){
10   std::call_once(onceFlag, [](){ std::cout << "Only once." << std::endl; });
11 }
12
13 void do_once2(){
14   std::call_once(onceFlag, [](){ std::cout << "Only once2." << std::endl; });
15 }
16
17 int main(){
18   std::cout << std::endl;
19
20   std::thread t1(do_once);
```

---

[1] Предложенная автором метафора *регистрации* вызываемого объекта может создать ложное впечатление, будто регистрация и выполнение разделены во времени. Дадим иную формулировку принципа работы функции `std::call_once`, приближенную к описанию из стандарта. Переменная типа `std::once_flag` может находиться в двух состояниях, показывающих, произошло ли уже некоторое однократное действие. Если в момент вызова функции `std::call_once` значение флага указывает, что действие уже состоялось, вызов немедленно завершается, ничего не делая; такой вызов функции `std::call_once` называется *пассивным*. В противном случае функция `std::call_once` вызывает переданный ей в качестве аргумента вызываемый объект (в частности, функцию) – такой вызов функции `std::call_once` называется *активным*. Если выполнение вызываемого объекта приводит к исключению, оно передаётся наружу из вызова функции `std::call_once`. Такой её вызов называется *исключительным*. Если же выполнение вызываемого объекта завершается нормально (вызов в этом случае называется *возвращающим*), флаг меняет своё состояние, и все последующие вызовы функции `std::call_once` с этим флагом гарантированно будут пассивными. Все вызовы функции `std::call_once` с одним и тем же флагом `std::once_flag` образуют вполне упорядоченную последовательность, в начале которой находится ноль или более исключительных вызовов, затем ровно один возвращающий, после которого следуют только пассивные вызовы. Таким образом, вместо регистрации функции для последующего вызова удобнее говорить об обращении к функции `std::call_once` как о попытке немедленно выполнить некоторое действие с двумя гарантиями: (1) после удачной попытки все остальные попытки игнорируются; (2) при одновременных попытках из различных потоков они обрабатываются последовательно. – *Прим. перев.*

```
21   std::thread t2(do_once);
22   std::thread t3(do_once2);
23   std::thread t4(do_once2);
24
25   t1.join();
26   t2.join();
27   t3.join();
28   t4.join();
29
30   std::cout << std::endl;
31 }
```

В этой программе запускаются четыре потока (строки 20–23). Два из них вызывают функцию `do_once`, а два других – функцию `do_once2`. Ожидаемый результат работы этой программы состоит в том, что на печать будет выведено ровно одно из сообщений: «Only once» (только раз) или «Only once2».



Использование однократных действий

Широко известный шаблон проектирования «Одиночка» (англ. *singleton*) призван гарантировать, что у некоторого класса создаётся только один экземпляр. Реализация этого шаблона в многопоточной среде оказывается непростым делом. Однако при наличии функции `std::call_once` и флага `std::once_flag` задача становится элементарной. Ниже показано, как инициализировать единичный экземпляр потокобезопасным образом.

Инициализация единичного экземпляра класса

```
1  // singletonCallOnce.cpp
2
3  #include <iostream>
4  #include <mutex>
5
6  using namespace std;
7
8  class MySingleton{
9    private:
10     static once_flag initInstanceFlag;
11     static MySingleton* instance;
12     MySingleton() = default;
13     ~MySingleton() = default;
```

```
14
15     static void initSingleton(){
16       instance = new MySingleton();
17     }
18
19   public:
20     MySingleton(const MySingleton&) = delete;
21     MySingleton& operator=(const MySingleton&) = delete;
22
23     static MySingleton* getInstance(){
24       call_once(initInstanceFlag, MySingleton::initSingleton);
25       return instance;
26     }
27 };
28
29 MySingleton* MySingleton::instance = nullptr;
30 once_flag MySingleton::initInstanceFlag;
31
32 int main(){
33   cout << endl;
34
35   cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;
36   cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;
37
38   cout << endl;
39 }
```

Посмотрим сначала на статическую переменную-член `initInstanceFlag`, объявленную в строке 10 и проинициализированную в строке 30. Этот флаг используется в статической функции-члене `getInstance` (строки 23–26) для гарантии того, что статическая функция-член `initSingleton` (строки 15–17) вызывается ровно один раз. В функции `initSingleton` создаётся единственный экземпляр класса.

Ключевые слова default и delete

Программист может заказать у компилятора некоторые особые функции-члены[1], указав ключевое слово `default`. Эти функции особенны тем, что компилятор может самостоятельно создать их реализации.

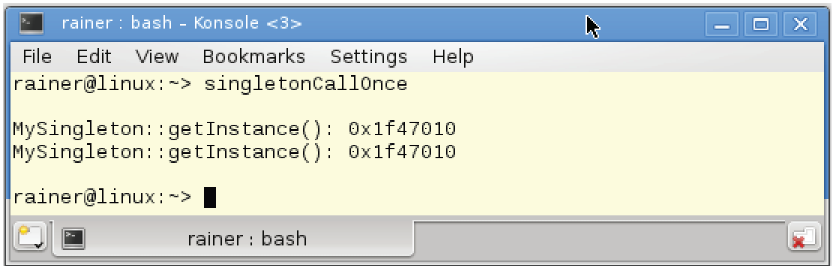
Объявление функции-члена ключевым словом `delete`, напротив, означает исключение этой функции из интерфейса класса, подавляет создание реализации компилятором и, как следствие, делает невозможным её вызов. Попытка вызвать такую функцию-член приводит к ошибке компиляции. Подробности о ключевых словах `default` и `delete` можно найти в справочнике[2].

---

[1] Начиная со стандарта C++11 – конструктор по умолчанию, конструкторы копирования и перемещения, копирующая и перемещающая операции присваивания, деструктор. В стандарте C++ 20 к ним добавились также операции сравнения, причём не только в виде функций-членов, но и в виде свободных (в том числе и дружественных) функций. – *Прим. перев.*

[2] https://isocpp.org/wiki/faq/cpp11-language-classes.

Функция getIstance выводит на печать адрес единственного экземпляра. Ниже показан пример запуска программы.



Потокобезопасное создание единственного экземпляра

3.3.4.3. Локальные статические переменные

Статические переменные, локальные для блока, инициализируются один раз ленивым образом. Это означает, что инициализируются они непосредственно перед первым использованием. Это свойство составляет основу реализации так называемого мейерсовского одиночки, названного в честь Скотта Мейерса[1]. На сегодняшний день это самая элегантная реализация шаблона «Одиночка» на языке C++. В стандарте C++ 11 для локальных статических переменных появилась новая гарантия: их инициализация потокобезопасна. Ниже представлен пример реализации одиночки по Мейерсу.

Мейерсовская реализация шаблона «Одиночка»

```
1  // meyersSingleton.cpp
2
3  class MySingleton{
4  public:
5    static MySingleton& getInstance() {
6      static MySingleton instance;
7      return instance;
8    }
9  private:
10   MySingleton();
11   ~MySingleton();
12   MySingleton(const MySingleton&)= delete;
13   MySingleton& operator=(const MySingleton&)= delete;
14 };
15
16 MySingleton::MySingleton()= default;
17 MySingleton::~MySingleton()= default;
18
19 int main(){
20   MySingleton::getInstance();
21 }
```

[1] https://en.wikipedia.org/wiki/Scott_Meyers.

Поддержка статических переменных

Используя мейерсовскую реализацию шаблона «Одиночка» в многопоточной среде, нужно убедиться, что компилятор реализует требование стандарта C++ 11 о потокобезопасной инициализации статических переменных. Нередко бывает, что программист полагается на описанные в стандарте гарантии, но оказывается, что компилятор не в полной мере им соответствует. В данном случае это может привести к созданию более чем одного объекта.

У данных с потоковой длительностью хранения нет никаких трудностей с многопоточностью. Поговорим теперь о них.

3.4. Данные с потоковой длительностью хранения

Данные потоков, также называемые данными с потоковой длительностью хранения, создаются для каждого потока отдельно. Они напоминают локальные статические данные тем, что их время жизни ограничено временем выполнения потоков[1], и тем, что инициализируются переменные при первом использовании. Это означает, что переменные потоков, объявленные в пространстве имён и объявленные как статические члены класса, инициализируются до первого использования, а переменные потоков, объявленные внутри функции, создаются при её первом вызове. Переменная с потоковой длительностью хранения полностью принадлежит своему потоку.

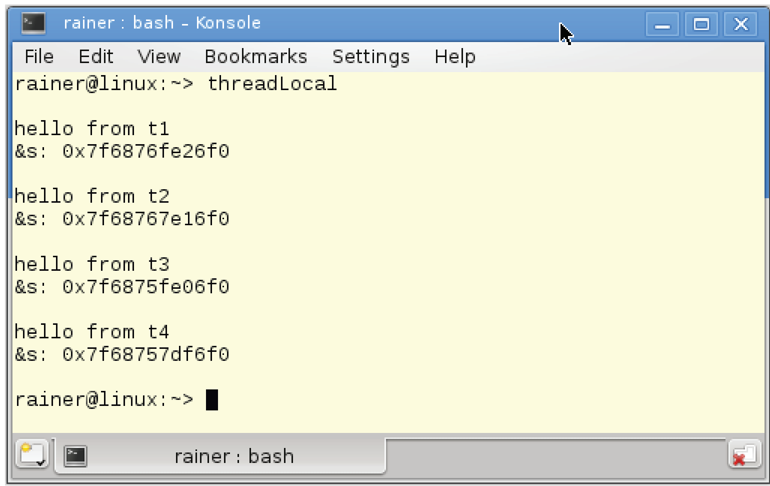
Данные с потоковой длительностью хранения

```
1  // threadLocal.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex coutMutex;
9
10 thread_local std::string s("hello from ");
11
12 void addThreadLocal(std::string const& s2){
13   s += s2;
14   // protect std::cout
15   std::lock_guard<std::mutex> guard(coutMutex);
16   std::cout << s << std::endl;
17   std::cout << "&s: " << &s << std::endl;
18   std::cout << std::endl;
```

[1] В случае локальных статических переменных время жизни ограничено временем работы всей программы. – *Прим. перев.*

```
19 }
20
21 int main(){
22   std::cout << std::endl;
23
24   std::thread t1(addThreadLocal,"t1");
25   std::thread t2(addThreadLocal,"t2");
26   std::thread t3(addThreadLocal,"t3");
27   std::thread t4(addThreadLocal,"t4");
28
29   t1.join();
30   t2.join();
31   t3.join();
32   t4.join();
33 }
```

При объявлении переменной `s` в строке 10 используется ключевое слово `thread_local`, поэтому переменная локальна для потока. В потоках `t1`–`t4`, создаваемых в строках 24–27, выполняется одна и та же функция `addThreadLocal` (строки 12–19). Потоки получают в качестве аргументов строки от «t1» до «t4» и добавляют их в конец строки `s`. Кроме того, в строке 17 функция выводит на печать адрес переменной `s`. Ниже показан результат запуска программы.



Данные с потоковой длительностью хранения

Что для переменной значит принадлежать потоку, неявно демонстрируется выводом в строке 16 и явно – в строке 17. Свой экземпляр переменой `s` создаётся для каждого потока. Каждый поток, во-первых, показывает значение своей переменой `s`, а во-вторых – свой, отличный от других потоков, её адрес.

На своих семинарах автор этой книги часто предлагает вопрос, в чём различие между статическими (`static`), потоковыми (`thread_local`) и локальными переменными. Время жизни статической переменной ограничено временем жизни главного потока; переменная потока живёт до тех пор, пока выполня-

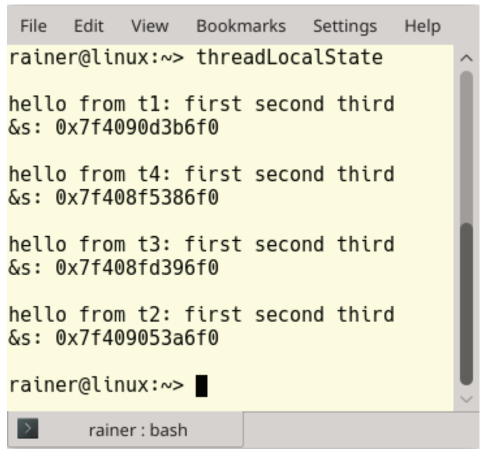
ется её поток; наконец, время жизни локальной переменной определяется временем выполнения блока, в котором она объявлена. Чтобы отчётливее пояснить это различие, ниже показана предыдущая программа `threadLocal.cpp` с небольшим изменением.

Общее состояние для нескольких функций

```
1  // threadLocalState.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex coutMutex;
9
10 thread_local std::string s("hello from ");
11
12 void first(){
13     s += "first ";
14 }
15
16 void second(){
17     s += "second ";
18 }
19
20 void third(){
21     s += "third";
22 }
23
24 void addThreadLocal(std::string const& s2){
25   s += s2;
26
27   first();
28   second();
29   third();
30   // protect std::cout
31   std::lock_guard<std::mutex> guard(coutMutex);
32   std::cout << s << std::endl;
33   std::cout << "&s: " << &s << std::endl;
34   std::cout << std::endl;
35 }
36
37 int main(){
38   std::cout << std::endl;
39
40   std::thread t1(addThreadLocal,"t1: ");
41   std::thread t2(addThreadLocal,"t2: ");
42   std::thread t3(addThreadLocal,"t3: ");
43   std::thread t4(addThreadLocal,"t4: ");
44
45   t1.join();
46   t2.join();
```

```
47   t3.join();
48   t4.join();
49 }
```

В этой версии программы функция `addThreadLocal` (строка 24) вызывает функции `first`, `second` и `third`. Каждая из этих функций дописывает к переменной потока `s` своё имя. Таким образом, строка `s` играет роль состояния, общего для нескольких функций, выполняемых в одном потоке (строки 27–29). Выводимый программой результат свидетельствует о том, что эти переменные в каждом потоке вполне независимы друг от друга.



Данные с потоковым временем хранения

Переработка однопоточной программы в многопоточную

Потоковые переменные помогают при переносе однопоточных программ в многопоточную среду. Если глобальные переменные объявить потоковыми, появляется гарантия, что каждый поток получает свою копию данных. Таким образом устраняется возможность общего изменяемого состояния, которое могло бы привести к гонке данных и, следовательно, к неопределённому поведению.

В отличие от локальных данных потока, переменные условия не столь просты в использовании.

3.5. Переменные условия

Переменные условия позволяют синхронизировать потоки посредством обмена сообщениями. Для их использования нужно подключить заголовочный файл `<condition_variable>`. Один поток выступает отправителем сообщения, а другой поток[1] – получателем. Получатель ждёт, пока не придёт сообщение.

[1] Или несколько потоков, см. функцию `notify_all`. – *Прим. перев.*

Чаще всего переменные условия применяются, когда нужно реализовать способ обработки данных по типу издателя и подписчика или производителя и потребителя. Переменная условия служит связующим звеном между отправителем и получателем сообщения.

Функции-члены переменных условия

Функция	Описание
`notify_one`	Оповестить один ожидающий поток о наступлении события
`notify_all`	Оповестить все ожидающие потоки
`wait`	Ожидать сообщения, держа блокировщик открытым
`wait_for`	Ожидать сообщения, держа блокировщик открытым, но не более заданного промежутка времени
`wait_until`	Ожидать сообщения, держа блокировщик открытым, но не более, чем до заданного момента времени
`native_handle`	Возвращает системный дескриптор переменной условия

Тонкое различие между функциями `notify_all` и `notify_one` состоит в том, что первая оповещает все потоки, ожидающие прихода сообщения, тогда как вторая оповещает только один из них[1]. Следующий пример призван пояснить – перед тем как мы погрузимся в леденящие душу подробности, – что можно передавать в функцию `wait` последним аргументом.

Использование переменных условия

```
1  // conditionVariable.cpp
2
3  #include <iostream>
4  #include <condition_variable>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex mutex_;
9  std::condition_variable condVar;
10
11 bool dataReady{false};
12
13 void doTheWork(){
14   std::cout << "Processing shared data." << std::endl;
15 }
16
17 void waitingForWork(){
18     std::cout << "Worker: Waiting for work." << std::endl;
19     std::unique_lock<std::mutex> lck(mutex_);
20     condVar.wait(lck, []{ return dataReady; });
21     doTheWork();
```

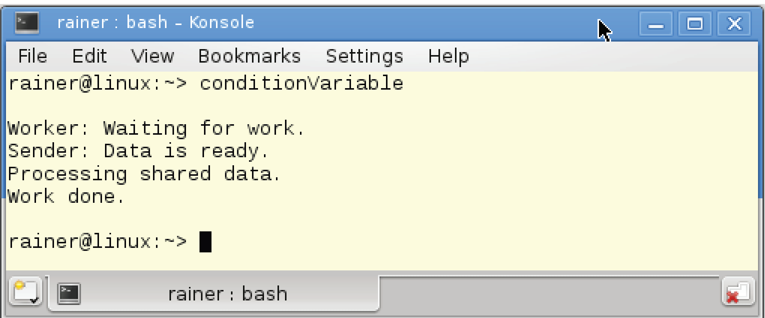
[1] Какой именно из множества ожидающих потоков получит оповещение, не определено – реализация имеет право выбрать адресата любым образом. Программисту следует считать, что поток для пробуждения будет выбран случайным образом. – *Прим. перев.*

```
22     std::cout << "Work done." << std::endl;
23 }
24 
25 void setDataReady(){
26     {
27       std::lock_guard<std::mutex> lck(mutex_);
28       dataReady = true;
29     }
30     std::cout << "Sender: Data is ready."  << std::endl;
31     condVar.notify_one();
32 }
33 
34 int main(){
35   std::cout << std::endl;
36 
37   std::thread t1(waitingForWork);
38   std::thread t2(setDataReady);
39 
40   t1.join();
41   t2.join();
42 
43   std::cout << std::endl;
44 }
```

В этой программе создаются два дочерних потока, `t1` и `t2`. В них выполняются, соответственно, функции `waitingForWork` и `setDataReady` (строки 37 и 38). Функция `setDataReady` оповещает – через переменную условия `condVar`, вызвав у неё функцию `notify_one`, – о завершении некоторой подготовительной работы. Тем временем поток-получатель `t1`, удерживая блокировку[1], начинает ожидание, для чего вызывает у переменной условия `condVar` функцию `wait`. Блокировщик мьютекса нужен как отправителю, так и получателю. При этом отправителю вполне довольно блокировщика `std::lock_guard`, так как захватить и отпустить мьютекс нужно только один раз. Получателю же необходим блокировщик `std::unique_lock`, поскольку функция `wait` может многократно захватывать и освобождать мьютекс. Ниже представлен пример работы этой программы.

---

[1] Следует подчеркнуть важную деталь: на момент начала ожидания, т. е. в момент вызова функции `wait` переменной условия, поток-получатель должен удерживать мьютекс захваченным; сама же функция `wait` проверяет условие и, если оно оказывается ложным, отпускает мьютекс и переводит поток в режим ожидания. Функция `wait` повторно захватывает мьютекс в момент пробуждения, когда через переменную условия приходит оповещение. Предикат (второй аргумент функции `wait`) проверяется при захваченном мьютексе. Если проверка проходит, функция `wait` завершается, мьютекс остаётся захваченным, и выполнение потока-получателя продолжается. В противном случае функция `wait` снова освобождает мьютекс и погружает поток в ожидание. Таким образом, поток-отправитель имеет возможность захватить мьютекс для себя и выполнять свою работу над общими данными. Когда эта работа сделана, он вместе с оповещением через переменную условия отдаёт и мьютекс, тем самым давая потоку-получателю возможность доступа к общим данным как для проверки предиката, так и для последующей их обработки. См. также раздел 3.5.3. – *Прим. перев.*



Синхронизация двух потоков через переменную условия

Тип std::condition_variable_any

Условные переменные типа std::condition_variable могут использовать для ожидания только блокировщики типа std::unique_lock<mutex>; более общий тип std::condition_variable_any, обладающий тем же интерфейсом, допускает любые пользовательские типы, подпадающие под понятие BasicLockable[1].

3.5.1. Использование предиката в функции ожидания

Возможно, читателю интересно узнать, зачем передавать предикат вторым аргументом в функцию wait, ведь этот параметр необязателен. Рассмотрим пример.

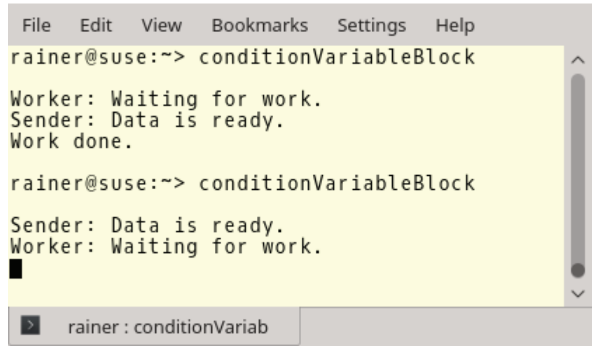
Бесконечное ожидание переменной условия без предиката

```
1  // conditionVariableBlock.cpp
2
3  #include <iostream>
4  #include <condition_variable>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex mutex_;
9  std::condition_variable condVar;
10
11 void waitingForWork() {
12     std::cout << "Worker: Waiting for work." << std::endl;
13
14     std::unique_lock<std::mutex> lck(mutex_);
15     condVar.wait(lck);
16     // do the work
17     std::cout << "Work done." << std::endl;
18 }
```

[1] https://en.cppreference.com/w/cpp/named_req/BasicLockable.

```
19
20 void setDataReady() {
21     std::cout << “Sender: Data is ready.”  << std::endl;
22     condVar.notify_one();
23 }
24
25 int main() {
26   std::cout << std::endl;
27
28   std::thread t1(setDataReady);
29   std::thread t2(waitingForWork);
30
31   t1.join();
32   t2.join();
33
34   std::cout << std::endl;
35 }
```

При первом запуске программа отработала нормально. Однако на втором запуске она зависла, так как отсылка оповещения одним потоком (строка 22) произошла раньше, чем второй поток вошёл в режим ожидания (строка 15).



Бесконечное ожидание переменной условия без предиката

Теперь всё становится ясно. Предикат наделяет переменную условия состоянием. Функция ожидания всегда должна сначала проверить истинность предиката. Предикат, таким образом, помогает бороться с двумя известными слабыми местами переменных условия: утерянным пробуждением и ложным пробуждением.

3.5.2. Утерянные и ложные пробуждения

Утерянным пробуждением называется ситуация, когда поток-отправитель успевает послать оповещение до того, как получатель начинает его ожидать. Как следствие оповещение оказывается утерянным. В стандарте языка C++ переменные условия описаны как механизм одновременной синхронизации:

«Класс `condition_variable` – это примитив синхронизации, который можно использовать для блокировки потока или нескольких потоков одновременно»[1]. В случае потерянного оповещения потоку-получателю остаётся ждать, ждать, ждать…

Ложное пробуждение – это пробуждение ожидающего потока, когда отправители никаких оповещений не посылали. По меньшей мере потоки POSIX[2] и API системы Windows[3] обладают этим недостатком. Одна из причин такого явления – похищенное пробуждение: перед тем как пробуждённый поток-адресат получает шанс запуститься, другой поток успевает вклиниться первым и начинает выполнение.

3.5.3. Процедура ожидания

Ожидающий поток работает по довольно сложному алгоритму. Приведём две самые важные строки из рассмотренного ранее примера `conditionVariable.cpp` – строки 19 и 20.

```
std::unique_lock<std::mutex> lck(mutex_);
condVar.wait(lck, []{ return dataReady; });
Эти две строки эквивалентны следующему коду4:
std::unique_lock<std::mutex> lck(mutex_);
while ( ![]{ return dataReady; }() ) {
    condVar.wait(lck);
}
```

Прежде всего необходимо различать, для чего служат блокировка мьютекса созданием объекта `lck` и последующее ожидание оповещения функцией `wait`.

- В самом начале поток захватывает мьютекс, тем самым получая исключительное право доступа к общим данным, и проверяет предикат – в данном случае просто отрицание переменной `dataReady`:
  – если предикат даёт значение `true`, поток продолжает свою работу;
  – если предикат даёт значение `false`, функция `wait` освобождает мьютекс и переводит поток в состояние ожидания.
- Функция wait переменной условия блокирует поток до тех пор, пока через переменную условия не придёт оповещение или пока не произойдёт ложное пробуждение. Вслед за этим выполняется такая последовательность действий:
  – поток просыпается и заново захватывает мьютекс;
  – проверяется предикат:
    - если предикат даёт значение `true`, поток продолжает свою работу;
    - если предикат даёт значение `false`, функция `wait` освобождает мьютекс и переводит поток в состояние ожидания.

---

[1] https://ru.cppreference.com/w/cpp/thread/condition_variable. – *Прим. перев.*

[2] https://en.wikipedia.org/wiki/POSIX_Threads.

[3] https://en.wikipedia.org/wiki/Windows_API.

[4] Этот код можно сделать проще, выразив условие цикла в виде `while(!dataReady)`. – *Прим. перев.*

Даже если все общие данные потоков состоят из единственной атомарной переменной, её изменение должно происходить под мьютексом, чтобы корректно передавать обновлённые данные ожидающему потоку.

Необходимость мьютекса

Даже если переменную `dataReady` в этом примере сделать атомарной, её модификацию нужно прикрыть мьютексом. В противном случае изменение её значения хоть и станет видимым для ожидающего потока, но это обновление может неправильно синхронизироваться. В итоге состояние гонок может привести к мёртвой блокировке. Что это означает – видимое, но некорректно синхронизированное изменение? Рассмотрим в подробностях процедуру ожидания в предположении, что переменная `dataReady` атомарна, а её изменение не защищено мьютексом.
Тогда оповещение может прийти в тот момент, когда переменная условия `condVar` ещё не начала его ожидать. Иными словами, оповещение может прийти после того, как завершена проверка предиката, но до вызова функции `wait`. В этом случае оповещение теряется, а поток-получатель, скорее всего, будет вечно спать, ожидая оповещения.
Этого бы не случилось, если бы переменная `dataReady` изменяла своё значение под защитой мьютекса. Благодаря синхронизации оповещение могло бы быть отправлено только после того, как поток-получатель перешёл в состояние ожидания.

В большинстве случаев задания (task) предоставляют менее подверженный ошибкам механизм синхронизации. В разделе 3.9.7 будет представлен сравнительный анализ переменных условия и заданий.

3.6. Кооперативное прерывание потоков (стандарт C++ 20)

Появившаяся в стандарте C++ 20 дополнительная возможность прерывать потоки кооперативно (т. е. с их согласия) основывается на классах `std::stop_source`, `std::stop_token` и `std::stop_callback`. Класс потоков `std::jthread` и класс переменных условия `std::condition_variable_any` также поддерживают кооперативное прерывание.

В первую очередь разберёмся, почему бы просто не «убить» поток.

Чем опасно жёсткое прерывание потоков

Жёсткое прерывание потока может быть опасно, так как заранее неизвестно его текущее состояние. Внезапное прерывание потока грозит следующими двумя[1] неприятными последствиями:

---

[1] Можно добавить и третью весьма вероятную неприятность: поток может удерживать некоторый ресурс, будь то динамическая память, открытый файл, графическое окно или системный таймер. Прерывание потока в этом состоянии лишает его возможности когда-либо освободить ресурс. В самом деле, владельцем ресурса, с точки зрения операционной системы, является целый процесс, а не тот или иной его поток. В частности, один поток может ресурс запрашивать (скажем, выделять динамическую память или открывать файл), а затем передавать владение им другому потоку. Поэтому прерывание потока может привести к утечке ресурсов, которая устраняется только завершением всего процесса. – *Прим. перев.*

- поток выполнил свою работу лишь наполовину. Невозможно определить, в каком состоянии он оставил данные, над которыми работал, а значит, и состояние всей программы. Это ведёт к неопределённому поведению, когда любые предположения о корректной работе программы теряют силу;
- поток может находиться в критической секции и удерживать захваченный мьютекс. Прерывание потока в этом состоянии с большой вероятностью приведёт к мёртвой блокировке других потоков.

Классы `std::stop_source` (источник останова), `std::stop_token` (признак останова) и `std::stop_callback` (обработчик останова) позволяют одному потоку асинхронно попросить другой поток о завершении, а этому другому потоку – проверить, имел ли место запрос на завершение. Признак останова (объект типа `std::stop_token`) можно передать в поток в качестве аргумента, чтобы поток затем периодически опрашивал, не пришёл ли запрос для завершения, или использовал этот объект для регистрации обработчика типа `std::stop_callback`. Для посылки запроса на останов используется объект `std::stop_source`. Сигнал от объекта-источника доходит до всех связанных с ним объектов-признаков типа `std::stop_token`. Объекты этих трёх классов совместно управляют состоянием завершения потока.

В последующих разделах кооперативное прерывание потоков рассматривается более подробно.

3.6.1. Класс `std::stop_source`

Объекты этого класса можно создавать двумя способами.

Конструкторы класса `std::stop_source`

```
1  std::stop_source();
2  explicit std::stop_source(std::nostopstate_t) noexcept;
```

Конструктор по умолчанию (строка 1) создаёт источник останова с новым состоянием останова. Конструктор, принимающий один аргумент типа `std::nostopstate_t` (строка 2), создаёт пустой объект-источник, не связанный ни с каким состоянием останова.

Функции-члены класса `std::stop_source`

Функция	Описание
`get_token`	Если останов возможен (см. `stop_possible`), возвращает признак останова (`stop_token`), связанный с тем же состоянием останова, что и объект-источник. В противном случае возвращает пустой (не связанный ни с каким состоянием останова) объект-признак
`stop_possible`	Возвращает значение «истина», если через данный объект-источник можно запросить останов потока[1], иначе – «ложь»

[1] Иными словами, если объект `std::stop_source` не пуст, т. е. содержит некоторое состояние останова. – *Прим. перев.*

Функция	Описание
stop_requested	Возвращает значение «истина», если останов возможен и был запрошен для данного состояния останова через один из связанных с ним объектов-источников
request_stop	Запрашивает останов. Если останов для данного объекта невозможен или уже был запрошен, функция ничего не делает

Вызов функции get_token для объекта-источника возвращает новый объект типа std::stop_token. Этот объект-признак может использоваться внутри потока для проверки того, сделан ли запрос на останов и возможен ли такой запрос. Таким образом, объект-признак наблюдает за состоянием останова, которым управляют объекты-источники.

Вызов функции request_stop у объекта-источника становится виден всем объектам-признакам и другим объектам-источникам, связанным с тем же самым состоянием останова. Помимо этого, вызываются все обработчики[1], зарегистрированные для признаков останова. Все ожидающие переменные условия типа std::condition_variable_any, у которых ожидание связано с соответствующими объектами stop_token, пробуждаются. Когда запрос на останов сделан, отменить его уже невозможно. Функции request_stop (при условии что останов уже был запрошен) и stop_possible выполняются атомарным образом.

Функция stop_requested возвращает значение true, если объект-источник связан с некоторым состоянием останова и останов для него был запрошен ранее. Вызов функции request_stop считается успешным и возвращает значение true, если состояние останова у источника существует и останов ранее не запрашивался.

3.6.2. Класс std::stop_token

Функции-члены класса std::stop_token

Функция	Описание
stop_possible	Возвращает значение «истина», если через данный объект-признак связан с каким-то состоянием останова
stop_requested	Возвращает значение «истина», если останов был запрошен для данного состояния останова, иначе «ложь»

Функция stop_possible возвращает значение «истина» и в том случае, если запрос на останов уже был сделан. Объект типа std::stop_token, созданный конструктором по умолчанию, не связан ни с каким состоянием останова.

Функция stop_requested возвращает значение «истина», если данный объект-признак связан с состоянием останова и для него выполнен запрос на останов.

Если всю функциональность объекта std::stop_token нужно временно отключить, его можно подменить объектом-признаком, созданным по умол-

---

[1] Стоит подчеркнуть, что обработчики, связанные с объектами stop_token, вызываются (в неопределённом порядке) синхронно в том потоке, который сделал запрос на останов через объект stop_source. – *Прим. перев.*

чанию. Следующий небольшой фрагмент кода демонстрирует, как отключать и снова включать способность потока реагировать на запрос останова.

Временное отключение признака останова

```
1  std::jthread jthr([](std::stop_token stoken) {
2      ...
3      std::stop_token interruptDisabled;
4      std::swap(stoken, interruptDisabled);
5      ...
6      std::swap(stoken, interruptDisabled);
7      ...
8  }
```

Объект `interruptDisabled` не связан ни с каким состоянием останова. Это означает, что поток `jthr` способен откликаться на просьбу об останове либо до строки 4, либо после строки 6.

3.6.3. Класс `std::stop_callback`

Класс `std::stop_callback` воплощает идиому RAII: в его конструкторе регистрируется вызываемый объект (в частности, функция), а в деструкторе регистрация отменяется. В следующем примере показано, как пользоваться обработчиками останова.

Использование обработчиков останова

```
1  // invokeCallback.cpp
2
3  #include <atomic>
4  #include <chrono>
5  #include <iostream>
6  #include <thread>
7  #include <vector>
8
9  using namespace std::literals;
10
11 auto func = [](std::stop_token stoken) {
12         std::atomic<int> counter{0};
13         auto thread_id = std::this_thread::get_id();
14         std::stop_callback callBack(stoken, [&counter, thread_id] {
15             std::cout << "Thread id: " << thread_id
16                       << "; counter: " << counter << '\n';
17         });
18         while (counter < 10) {
19             std::this_thread::sleep_for(0.2s);
20             ++counter;
21         }
22     };
23
24 int main() {
```

```
25     std::cout << '\n';
26
27     std::vector<std::jthread> vecThreads(10);
28     for(auto& thr: vecThreads) thr = std::jthread(func);
29
30     std::this_thread::sleep_for(1s);
31
32     for(auto& thr: vecThreads) thr.request_stop();
33
34     std::cout << '\n';
35 }
```

Каждый из десяти потоков выполняет лямбда-функцию `func`, объявленную в строках 11–22. Эта функция в строках 14–17 выводит на печать идентификатор потока и текущее значение атомарного счётчика. Счётчик, объявленный в строке 12, должен быть атомарным, поскольку он модифицируется в дочернем потоке и одновременно читается в обработчике, который вызывается в главном потоке. Благодаря односекундной задержке в главном потоке и задержкам на каждой итерации дочерних потоков в момент запроса на останов обработчик отображает значение счётчиков, равное четырём. Вызов функции `request_stop` для объекта `thr` типа `std::jthread` посылает потоку запрос на останов и, следовательно, приводит к вызову обработчика. Ниже показан результат работы программы[1]. Больше подробностей о кооперативном прерывании потоков типа `std::jthread` можно найти в разделе 3.2.

[1] Приведённый здесь пример нуждается в дополнительных пояснениях и может быть улучшен. Во-первых, напомним: если функция (или, говоря шире, вызываемый объект) принимает первый параметр типа `std::stop_token`, конструктор класса `std::jthread` неявно передаёт объект-признак, связанный с тем же состоянием останова, что и логически содержащийся в объекте `std::jthread` подобъект `stop_source`.
Таким образом, десять дочерних потоков обладают каждый своим источником останова. Поэтому каждому из них нужно посылать свой сигнал останова (строка 32). Можно было бы вместо типа `std::jthread` использовать более простой тип `std::thread` без встроенной поддержки останова и останавливать все потоки единым сигналом извне, как показано ниже:

```
std::stop_source source;
std::vector<std::thread> vecThreads(10);
for (auto& thr : vecThreads)
  thr = std::thread(func, source.get_token());
std::this_thread::sleep_for(1s);
source.request_stop();
for (auto& thr : vecThreads) thr.join();
```

Далее, приведённый пример, иллюстрируя вызов обработчика, вызываемого в ответ на сигнал останова, не прерывает выполнение потока. Этого легко добиться, изменив условие цикла следующим образом:

```
while (counter < 10 && !stoken.stop_requested())
```

Объявление `func` как лямбда-функции вряд ли оправдано. Поведение программы ничуть не изменилось бы, `func` будь обычной функцией:

`void func(std::stop_token stoken)`. – *Прим. перев.*

rainer : bash — Konsole

File Edit View Bookmarks Settings Help

```
rainer@seminar:~> invokeCallback

Thread id: 140276632897280; counter: 4
Thread id: 140276624504576; counter: 4
Thread id: 140276616111872; counter: 4
Thread id: 140276607719168; counter: 4
Thread id: 140276599326464; counter: 4
Thread id: 140276590933760; counter: 4
Thread id: 140276582541056; counter: 4
Thread id: 140276574148352; counter: 4
Thread id: 140276565755648; counter: 4
Thread id: 140276557362944; counter: 4

rainer@seminar:~>
```

rainer : bash

Обработчик останова

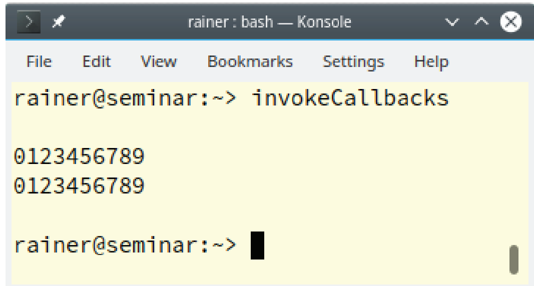
Обработчики останова, зарегистрированные в конструкторе класса `std::stop_callback`, вызываются, как правило, в том потоке, который вызывает функцию `request_stop` соответствующего объекта `std::stop_source`. Однако если останов уже был запрошен к моменту регистрации, обработчик вызывается в том потоке, который конструирует объект `std::stop_callback`.

Для любого из потоков, использующих один объект `std::stop_token`, можно регистрировать сколь угодно много обработчиков останова. Стандарт языка C++ ничего не говорит о том, в каком порядке они вызываются. Пример множественных обработчиков представлен ниже.

Регистрация нескольких обработчиков останова в двух потоках

```
1  // invokeCallbacks.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  using namespace std::literals;
8
9  void func(std::stop_token stopToken) {
10     std::this_thread::sleep_for(100ms);
11     for (int i = 0; i <= 9; ++i) {
12         std::stop_callback cb(stopToken, [i] { std::cout << i; });
13     }
14     std::cout << ‘\n’;
15 }
16
17 int main() {
18     std::cout << ‘\n’;
19
20     std::jthread thr1 = std::jthread(func);
21     std::jthread thr2 = std::jthread(func);
```

```
22     thr1.request_stop();
23     thr2.request_stop();
24
25     std::cout << '\n';
26 }
```



Регистрация нескольких обработчиков останова в двух потоках

3.6.4. Общий механизм посылки сигналов

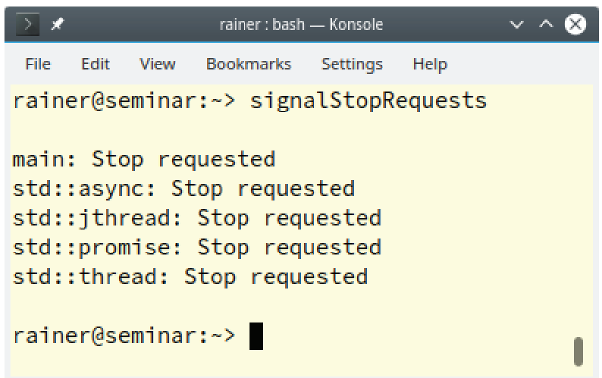
Можно сказать, что работающие в паре классы `std::stop_source` и `std::stop_token` составляют универсальный механизм для посылки сигналов. Копируя объект `std::stop_token`, можно один и тот же сигнал рассылать любым сущностям, выполняющим какой-либо код. В следующем примере показано, как этот механизм рассылки сигналов работает с классами `std::async`, `std::promise`, `std::thread` и `std::jthread` в разных комбинациях.

Рассылка сигнала разным единицам выполнения

```
1  // signalStopRequests.cpp
2
3  #include <iostream>
4  #include <thread>
5  #include <future>
6
7  using namespace std::literals;
8
9  void function1(std::stop_token stopToken, const std::string& str){
10     std::this_thread::sleep_for(1s);
11     if (stopToken.stop_requested())
12         std::cout << str << ": Stop requested\n";
13 }
14
15 void function2(
16     std::promise<void> prom,
17     std::stop_token stopToken, const std::string& str)
18 {
19     std::this_thread::sleep_for(1s);
```

```
20     std::stop_callback callBack(stopToken, [&str] {
21         std::cout << str << ": Stop requested\n";
22     });
23     prom.set_value();
24 }
25
26 int main() {
27     std::cout << '\n';
28
29     std::stop_source stopSource;
30
31     std::stop_token stopToken = std::stop_token(stopSource.get_token());
32
33     std::thread thr1 = std::thread(function1, stopToken, "std::thread");
34
35     std::jthread jthr = std::jthread(
36         function1, stopToken, "std::jthread");
37
38     auto fut1 = std::async([stopToken] {
39         std::this_thread::sleep_for(1s);
40         if (stopToken.stop_requested())
41             std::cout << "std::async: Stop requested\n";
42     });
43
44     std::promise<void> prom;
45     auto fut2 = prom.get_future();
46     std::thread thr2(
47         function2, std::move(prom), stopToken, "std::promise");
48
49     stopSource.request_stop();
50     if (stopToken.stop_requested())
51         std::cout << "main: Stop requested\n";
52
53     thr1.join();
54     thr2.join();
55
56     std::cout << '\n';
57 }
```

Имея объект `stopSource` (объявлен в строке 29), можно создать объект `stopToken` (строка 31) и снимать с него копию при создании каждой единицы выполнения: `std::thread` (строка 33), `std::jthread` (строка 35), `std::async` (строка 37) и `std::promise` (строка 45). Копирование объектов типа `std::stop_token` – дешёвая операция. В строке 47 для объекта `stopSource` вызывается функция `request_stop`. В числе прочих адресатов сигнал получает и главный поток (строка 48). Класс `std::jthread` используется в этом примере из-за набора удобных функций-членов, позволяющих в явном виде управлять кооперативным прерыванием потоков. Более подробные сведения об этом классе читатель найдёт в разделе 3.6.5.



Рассылка сигнала разным единицам выполнения

Читателю может быть любопытно, зачем единицы выполнения ожидают по одной секунде (строки 10, 19 и 39). Задержки нужны для того, чтобы вызову функции `request_stop` в строке 49 было что делать. Единицы выполнения, такие как `std::thread` (строка 33), `std::jthread` (строка 35), `std::async` (строка 37) и `std::promise` (строка 45), в момент появления запроса на останов могут находиться в одном из следующих состояний:

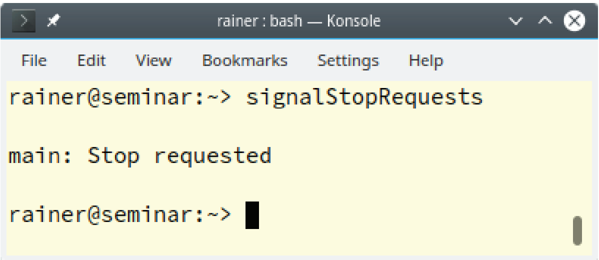
- не запущена. Тогда вызов функции `stop_requested`, когда до него дойдёт выполнение, вернёт значение `true`, и обработчик выполнится;
- выполняется. Единица выполнения получает сигнал в процессе своей работы. Чтобы запрос на останов оказал действие, он должен быть отправлен до того, как единица выполнения дойдёт до вызова функции `stop_requested`, или до инициализации обработчика;
- завершена. Вызов функции `request_stop` не оказывает никакого действия на единицу выполнения, обработчик не вызывается.

Посмотрим, что случится, если дождаться завершения потоков `thr1` и `thr2` до того, как посылать сигнал останова. Для этого нужно строки 53 и 53 поставить перед строкой 49, как показано ниже.

Отправка сигнала после завершения потоков

```
1       thr1.join();
2       thr2.join();
3
4       stopSource.request_stop();
5       if (stopToken.stop_requested())
6           std::cout << “main: Stop requested\n”;
```

Такая перестановка строк приводит к тому, что теперь лишь главный поток реагирует на сигнал.



Отсутствие реакции на запоздавший сигнал

3.6.5. Особенности класса std::jthread

Класс std::jthread представляет собой почти что копию класса std::thread с дополнительными возможностями: кооперативным прерыванием и автоматическим присоединением. Для поддержки обеих этих возможностей поток std::thread обладает встроенным объектом std::stop_token.

Функция	Описание
get_stop_source	Возвращает объект типа std::stop_source, связанный с состоянием останова
get_stop_token	Возвращает объект типа std::stop_token, связанный с состоянием останова
request_stop	Запрашивает останов потока через состояние останова

3.6.6. Новые перегрузки функции wait в классе std::condition_variable_any

Класс std::condition_variable_any представляет собой расширенный вариант класса[1] std::condition_variable. Объекты класса std::condition_variable требуют для работы исключительно блокировщиков типа std::unique_lock<std::mutex>, тогда как тип std::condition_variable_any может работать с любым типом блокировщика, поддерживающим функции-члены lock и unlock.

У трёх знакомых по предыдущим разделам функций-членов: wait, wait_for и wait_until – появляются в классе std::condition_variable_any новые перегрузки, принимающие аргумент типа std::stop_token.

Новые перегрузки функций ожидания

```
1  template <class Predicate>
2  bool wait(
3      Lock& lock,
4      stop_token stoken,
5      Predicate pred);
6
7  template <class Rep, class Period, class Predicate>
```

[1] https://en.cppreference.com/w/cpp/thread/condition_variable.

```
8  bool wait_for(
9      Lock& lock,
10     stop_token stoken,
11     const chrono::duration<Rep, Period>& rel_time,
12     Predicate pred);
13
14 template <class Clock, class Duration, class Predicate>
15 bool wait_until(
16     Lock& lock,
17     stop_token stoken,
18     const chrono::time_point<Clock, Duration>& abs_time,
19     Predicate pred);
```

Новым перегрузкам требуется предикат в качестве обязательного аргумента. Эти функции сначала проверяют, запрошен ли останов потока через объект `std::stop_token`. Три новые функции возвращают значение, которое даёт вычисление предиката. При этом возвращаемое значение не зависит от того, был ли запрошен останов потока, как и от истечения предельного времени ожидания. Новые перегрузки эквивалентны следующим фрагментам кода.

Возможные реализации трёх новых перегрузок

```
1  // функция wait, строки 1-4
2  while (!stoken.stop_requested()) {
3      if (pred()) return true;
4      wait(lock);
5  }
6  return pred();
7
8  // функция wait_for, строки 6-10
9  return wait_until(
10     lock,
11     std::move(stoken),
12     chrono::steady_clock::now() + rel_time,
13     std::move(pred));
14
15 // функция wait_until, строки 12-16
16 while (!stoken.stop_requested()) {
17     if (pred()) return true;
18     if (wait_until(lock, timeout_time) == std::cv_status::timeout)
19         return pred();
20 }
21 return pred();
```

После обращения к функции `wait` можно проверить, был ли запрос на останов потока.

Обработка прерывания потока с помощью функции `wait`

```
1  cv.wait(lock, stoken, predicate);
2  if (stoken.stop_requested()){
3      // interrupt occurred
4  }
```

В следующем примере показано использование переменной условия совместно с механизмом прерывания потока.

Использование переменной условия с запросом на прерывание потока

```
1  // conditionVariableAny.cpp
2
3  #include <condition_variable>
4  #include <thread>
5  #include <iostream>
6  #include <chrono>
7  #include <mutex>
8  #include <thread>
9
10 using namespace std::literals;
11
12 std::mutex mut;
13 std::condition_variable_any condVar;
14
15 bool dataReady;
16
17 void receiver(std::stop_token stopToken) {
18     std::cout << "Waiting" << '\n';
19
20     std::unique_lock<std::mutex> lck(mut);
21     bool ret = condVar.wait(lck, stopToken, []{return dataReady;});
22     if (ret){
23         std::cout << "Notification received: " << '\n';
24     }
25     else{
26          std::cout << "Stop request received" << '\n';
27     }
28 }
29
30 void sender() {
31     std::this_thread::sleep_for(5ms);
32     {
33         std::lock_guard<std::mutex> lck(mut);
34         dataReady = true;
35         std::cout << "Send notification"  << '\n';
36     }
37     condVar.notify_one();
38 }
39
40 int main(){
41   std::cout << '\n';
42
43   std::jthread t1(receiver);
44   std::jthread t2(sender);
45
46   t1.request_stop();
47
48   t1.join();
```

```
49   t2.join();
50
51   std::cout << '\n';
52 }
```

Поток-получатель (строки 17–28) ждёт оповещения от потока-отправителя (строки 30–38). Перед тем как поток-отправитель посылает оповещение в строке 37, главный поток успевает в строке 46 потребовать остановки потока. Вывод программы свидетельствует о том, что запрос на останов происходит ранее, чем оповещение о готовности данных.

```
Waiting
Stop request received
Send notification
```

Обработка запроса на останов переменной условия

3.7. Семафоры (стандарт C++ 20)

Семафоры – это механизм синхронизации, используемый для управления одновременным доступом потоков к общему ресурсу. Считающий семафор – это разновидность семафора, ведущий счётчик потоков, которым ещё разрешается вход в критическую секцию. Конструктор семафора устанавливает начальное значение счётчика. Захват семафора уменьшает счётчик на единицу, а освобождение – увеличивает. Если поток пытается захватить семафор, когда его значение равно нулю, поток блокируется до тех пор, пока значение счётчика не увеличится в результате освобождения семафора каким-то другим потоком.

Эдсгер Дейкстра – изобретатель семафоров

Понятие семафора предложил в 1965 г. нидерландский учёный в области информатики и программирования Эдсгер Вибе Дейкстра. Семафор – это структура данных, содержащая очередь и счётчик. Счётчик инициализируется значением, большим или равным нулю. Семафор поддерживает две операции: `wait` и `signal`. Операция `wait` захватывает семафор, уменьшая значение счётчика, если оно положительно, или блокирует поток в противном случае. Операция `signal` освобождает семафор путём увеличения счётчика[1]. Постановка заблокированных потоков в очередь необходима для предотвращения ресурсного голода[2].

---

[1] И если очередь заблокированных потоков не пуста, пробуждает первый поток из неё. – *Прим. перев.*

[2] Ресурсный голод – ситуация, когда поток или процесс не может продолжить работу и проводит неопределённо долгое время в ожидании из-за того, что система каждый раз отказывает ему в предоставления некоторого ресурса, см. https://en.wikipedia.org/wiki/Starvation_(computer_science). – *Прим. перев.*

Термин «семафор» первоначально означал механическое средство сигнализации для подвижного состава на железных дорогах.

Семафор на железной дороге[1]

Стандарт C++ 20 содержит тип двоичного семафора `std::binary_semaphore`, который представляет собой псевдоним для типа `std::counting_semaphore<1>`, у которого нижняя граница наибольшего значения счётчика равна 1. С помощью типа `std::binary_semaphore` можно реализовать блокировки[2].

```
using binary_semaphore = std::counting_semaphore<1>;
```

В отличие от объектов типа `std::mutex`, семафоры не привязаны к определённым потокам. Это означает, что операции захвата и освобождения семафора могут выполняться в различных потоках. В следующей таблице показан интерфейс класса `std::counting_semaphore`.

Функции-члены класса std::counting_semaphore

Пример применения функции	Описание
`std::counting_semaphore sem{num}`	Конструктор. Создаёт семафор с начальным значением num
`sem.max()`	Статическая функция. Возвращает наибольшее возможное значение счётчика
`sem.release(upd = 1)`	Увеличивает счётчик на величину upd и разблокирует потоки, ожидающие этот семафор
`sem.acquire()`	Уменьшает значение счётчика или блокирует поток, пока счётчик не станет больше нуля
`sem.try_acquire()`	Пытается уменьшить значение счётчика, если он больше нуля; в противном случае возвращает значение `false`

---

[1] Изображение первоначально загружено пользователем Amos E Wolfe для английской версии Википедии, впоследствии перенесено в Викисклад. https://commons.wikimedia.org/w/index.php?curid=1972304.

[2] https://en.cppreference.com/w/cpp/named_req/BasicLockable.

Пример применения функции	Описание
`sem.try_acquire_for(relTime)`	Пытается уменьшить значение счётчика, если он больше нуля; в противном случае ожидает не более заданного промежутка времени и, если семафор по-прежнему невозможно захватить, возвращает значение `false`
`sem.try_acquire_until(absTime)`	Пытается уменьшить значение счётчика, если он больше нуля; в противном случае ожидает не более, чем до заданного момента времени и, если семафор по-прежнему невозможно захватить, возвращает значение `false`

Вызов конструктора

```
std::counting_semaphore<10> sem(5)
```

создаёт семафор с нижней границей наибольшего значения счётчика, равной 10, и начальным значением счётчика 5. Вызов `sem.max()` возвращает наибольшее возможное значение внутреннего счётчика. При вызове `sem.release(upd)` должны выполнятся следующие ограничения: `upd >= 0` и `upd + counter <= sem.max()`. Функция `try_acquire_for` принимает в качестве аргумента промежуток времени, тогда как функция `try_acquire_until` принимает момент времени. Функции `try_acquire`, `try_acquire_for` и `try_acquire_until` возвращают значение логического типа, выражающее успешность вызова.

Семафоры обычно используются для организации взаимодействия между отправителями и получателями. Например, путём инициализации семафора значением 0 можно заблокировать поток-получатель на вызове функции `acquire` до тех пор, пока поток-отправитель не сформирует сообщение и не вызовет для семафора функцию `release`. Таким образом, получатель ждёт оповещения от отправителя. С помощью семафоров легко можно воплотить однократную синхронизацию потоков.

Синхронизация потоков на основе семафора

```
1  // threadSynchronisationSemaphore.cpp
2
3  #include <iostream>
4  #include <semaphore>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::counting_semaphore<1> prepareSignal(0);
11
12 void prepareWork() {
13     myVec.insert(myVec.end(), {0, 1, 0, 3});
14     std::cout << "Sender: Data prepared."  << '\n';
15     prepareSignal.release();
16 }
17
18 void completeWork() {
19     std::cout << "Waiter: Waiting for data." << '\n';
20     prepareSignal.acquire();
```

```
21     myVec[2] = 2;
22     std::cout << “Waiter: Complete the work.” << ‘\n’;
23     for (auto i: myVec) std::cout << i << “ “;
24     std::cout << ‘\n’;
25 }
26
27 int main() {
28     std::cout << ‘\n’;
29
30     std::thread t1(prepareWork);
31     std::thread t2(completeWork);
32
33     t1.join();
34     t2.join();
35
36     std::cout << ‘\n’;
37 }
```

В строке 10 объявляется переменная `prepareSignal` типа `std::counting_semaphore`, которая может принимать лишь значения 0 и 1. В этой программе переменная инициализируется значением 0. Это означает, что вызов функции `release` в строке 15 установит значение счётчика в 1 и разблокирует второй поток, ожидающий на вызове функции `acquire` в строке 20. Ниже представлен пример выполнения этой программы.

```
x64 Native Tools Command Prompt for VS 2019

C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3


C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3


C:\Users\seminar>threadSynchronisationSemaphore.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3


C:\Users\seminar>threadSynchronisationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3


C:\Users\seminar>
```

Синхронизация потоков на основе семафора

3.8. Защёлки и барьеры (стандарт C++ 20)

Защёлки и барьеры – это средства синхронизации, позволяющие заблокировать поток до тех пор, пока некоторый счётчик не достигнет нуля. Нужно сразу же подчеркнуть, что барьеры, о которых идёт речь здесь, не имеют ничего общего с барьерами памяти, знакомыми из предыдущей главы. В стандарте C++ 20 барьеры и защёлки представлены двумя классами: `std::latch` и `std::barrier`. Одновременный вызов функций-членов для одного и того же объекта какого-либо из этих классов не приводит к гонке данных.

Начнём с ответов на два вопроса.

1. В чём различие между этими двумя механизмами координации потоков? Объект типа `std::latch` можно использовать лишь один раз, тогда как класс `std::barrier` допускает многократное использование. Объект типа `std::latch` может быть полезен в случаях, когда несколько потоков совместно решают единственную задачу. Тип `std::barrier` помогает управлять выполнением повторяющихся задач в нескольких потоках. Кроме того, класс `std::barrier` позволяет выполнить определённую функцию-обработчик на так называемом заключительном шаге (т. е. когда значение счётчика достигает нуля).
2. Какие новые сценарии использования допускают защёлки и барьеры, которых нельзя было бы реализовать средствами стандартов C++ 11 и C++ 14, комбинируя различным образом фьючерсы, потоки, переменные условия и блокировщики? Никаких принципиально новых возможностей эти два класса не добавляют, однако они гораздо удобнее в использовании. Также они оказываются более эффективными, так как их внутренняя реализация часто основывается на неблокирующих механизмах.

3.8.1. Класс `std::latch`

Рассмотрим подробнее интерфейс класса `std::latch`.

Функции-члены класса std::latch

Функция	Описание
`std::latch lat{cnt}`	Создать защёлку, задав начальное значение счётчика
`count_down(upd)`	Атомарным образом уменьшить значение счётчика на величину `upd` (по умолчанию 1), не блокируя вызывающий поток
`try_wait`	Возвращает значение `true`, если счётчик равен нулю
`wait`	Возвращает управление, если счётчик равен нулю. Иначе – блокирует выполнение до тех пор, пока счётчик на достигнет нуля
`arrive_and_wait(upd)`	Эквивалентно `count_down(upd); wait();`
`max`	Статическая. Возвращает наибольшее значение счётчика, поддерживаемое реализацией

Аргумент upd в обеих функциях можно опускать, по умолчанию его значение равно 1. Если значение этого аргумента больше, чем текущее значение счётчика, или отрицательно, программа обладает неопределённым поведением. Выполнение функции try_wait никогда не приводит к ожиданию, вопреки её названию.

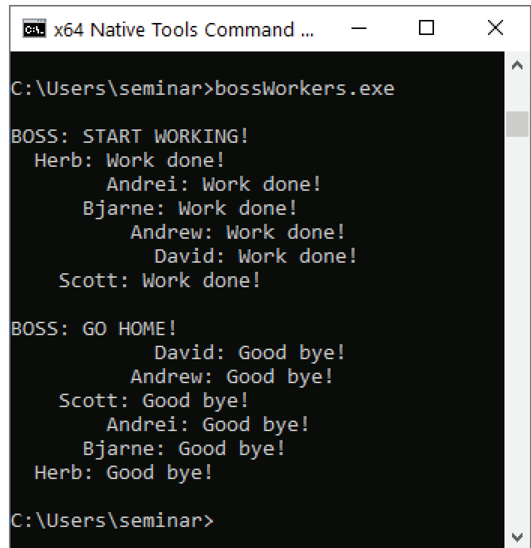
В следующей программе показано, как с помощью двух защёлок организовать взаимодействие потоков по типу «начальник–подчинённые». Для синхронизированного вывода на консоль служит функция synchronizedOut (строка 13) – так удобнее будет проследить ход выполнения программы.

Взаимодействие начальника и подчинённых на основе защёлок

```
1  // bossWorkers.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <latch>
6  #include <thread>
7
8  std::latch workDone(6);
9  std::latch goHome(1);
10
11 std::mutex coutMutex;
12
13 void synchronizedOut(const std::string s) {
14     std::lock_guard<std::mutex> lo(coutMutex);
15     std::cout << s;
16 }
17
18 class Worker {
19  public:
20      Worker(std::string n): name(n) { };
21
22       void operator() (){
23           // notify the boss when work is done
24           synchronizedOut(name + ": " + "Work done!\n");
25           workDone.count_down();
26
27           // waiting before going home
28           goHome.wait();
29           synchronizedOut(name + ": " + "Good bye!\n");
30       }
31  private:
32     std::string name;
33 };
34
35 int main() {
36     std::cout << '\n';
37
38     std::cout << "BOSS: START WORKING! " << '\n';
39
```

```
40     Worker herb("  Herb");
41     std::thread herbWork(herb);
42
43     Worker scott("    Scott");
44     std::thread scottWork(scott);
45
46     Worker bjarne("      Bjarne");
47     std::thread bjarneWork(bjarne);
48
49     Worker andrei("        Andrei");
50     std::thread andreiWork(andrei);
51
52     Worker andrew("          Andrew");
53     std::thread andrewWork(andrew);
54
55     Worker david("            David");
56     std::thread davidWork(david);
57
58     workDone.wait();
59
60     std::cout << '\n';
61
62     goHome.count_down();
63
64     std::cout << "BOSS: GO HOME!" << '\n';
65
66     herbWork.join();
67     scottWork.join();
68     bjarneWork.join();
69     andreiWork.join();
70     andrewWork.join();
71     davidWork.join();
72 }
```

Идея взаимодействия потоков вполне очевидна. Каждый из шести потоков, `herb`, `scott`, `bjarne`, `andrei`, `andrew` и `david`, созданных в строках 40–56, должен выполнить свою работу. Закончив, поток уменьшает счётчик `workDone` типа `std::latch` (строка 25). Главный поток программы, выполняющий роль начальника, блокируется в строке 58 до тех пор, пока счётчик не достигнет нуля. Когда это произойдёт, т. е. когда последний поток-подчинённый рапортует о завершении своей работы, начальник командует работникам расходиться по домам, используя для этого вторую защёлку: `goHome`. У этой защёлки начальное значение счётчика равно 1 (строка 9). Все потоки-подчинённые ждут, заблокированные на функции `wait`, до тех пор, пока этот счётчик не обнулится. Пример выполнения программы показан на рисунке.



Взаимодействие начальника и подчинённых на основе защёлок

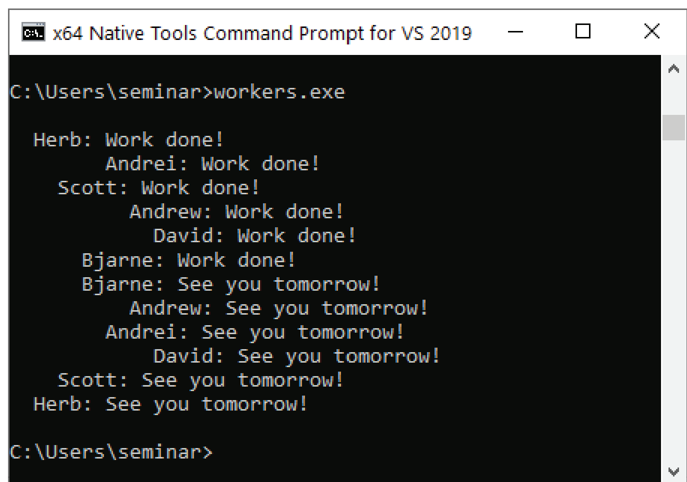
Присмотревшись внимательнее к протоколу взаимодействия этих потоков, можно заметить, что работники могут обойтись и без начальника. Код программы представлен ниже.

Взаимодействие работников на основе защёлок

```
1  // workers.cpp
2
3  #include <iostream>
4  #include <barrier>
5  #include <mutex>
6  #include <thread>
7
8  std::latch workDone(6);
9  std::mutex coutMutex;
10
11 void synchronizedOut(const std::string& s) {
12     std::lock_guard<std::mutex> lo(coutMutex);
13     std::cout << s;
14 }
15
16 class Worker {
17 public:
18     Worker(std::string n): name(n) { };
19
```

```
20     void operator() () {
21         synchronizedOut(name + ": " + "Work done!\n");
22         workDone.arrive_and_wait();  // wait until all work is done
23         synchronizedOut(name + ": " + "See you tomorrow!\n");
24     }
25  private:
26     std::string name;
27 };
28
29 int main() {
30     std::cout << '\n';
31
32     Worker herb("  Herb");
33     std::thread herbWork(herb);
34
35     Worker scott("    Scott");
36     std::thread scottWork(scott);
37
38     Worker bjarne("      Bjarne");
39     std::thread bjarneWork(bjarne);
40
41     Worker andrei("        Andrei");
42     std::thread andreiWork(andrei);
43
44     Worker andrew("          Andrew");
45     std::thread andrewWork(andrew);
46
47     Worker david("            David");
48     std::thread davidWork(david);
49
50     herbWork.join();
51     scottWork.join();
52     bjarneWork.join();
53     andreiWork.join();
54     andrewWork.join();
55     davidWork.join();
56 }
```

Не так много остаётся добавить к этой упрощённой программе. Вызов функции `arrive_and_wait` в строке 22 эквивалентен вызову двух функций: сначала `count_down`, затем `wait`. При таком подходе потоки сами координируют свою работу, необходимости в отдельном потоке-начальнике, как в предыдущей программе, более нет.



Взаимодействие работников на основе защёлок

Класс `std::barrier` во многом подобен классу `std::latch`.

3.8.2. Класс `std::barrier`

Между классами `std::latch` и `std::barrier` есть два главных различия. Во-первых, объект класса `std::barrier` можно использовать много раз; во-вторых, значение счётчика можно заново устанавливать перед новым использованием. Начальное значение счётчика устанавливается в конструкторе. Вызовы функций `arrive`, `arrive_and_wait` и `arrive_and_drop` уменьшают значение счётчика. Кроме того, функция `arrive_and_drop` уменьшает на единицу значение, которое станет начальным для счётчика на последующих фазах. Сразу после завершения текущей фазы, когда счётчик достигает нуля, запускается так называемый заключительный шаг. На этом шаге запускается заданный в конструкторе вызываемый объект – обработчик.

Заключительный шаг выполняется следующим образом:

1) потоки заблокированы в ожидании на функциях `arrive`;
2) из этих потоков произвольным образом выбирается один, и в нём выполняется обработчик. Обработчик должен иметь спецификацию `noexcept` и не может выбрасывать исключения;
3) когда обработчик завершает работу, все ожидающие потоки разблокируются.

Функции-члены класса std::barrier

Функция	Описание
`std::barrier bar{cnt}`	Конструктор. Создаёт объект-барьер с начальным значением счётчика
`std::barrier bar{cnt, call}`	Конструктор. Создаёт объект-барьер с начальным значением счётчика и обработчиком
`bar.arrive(upd)`	Атомарным образом уменьшает значение счётчика на заданную величину
`bar.wait()`	Блокирует поток до обнуления счётчика и выполнения обработчика
`bar.arrive_and_wait()`	Эквивалентна конструкции `wait(arrive())`
`bar.arrive_and_drop()`	Уменьшает счётчик и уменьшает начальное значение счётчика для последующих фаз
`std::barrier::max`	Статическая. Возвращает наибольшее значение счётчика, поддерживаемое реализацией

Вызов функции `arrive_and_drop` приводит к тому, что следующая фаза начнётся с на единицу меньшего начального значения счётчика. В следующей программе количество работников уменьшается наполовину во второй фазе.

Работники на полную ставку и на полставки

```
1  // fullTimePartTimeWorkers.cpp
2
3  #include <iostream>
4  #include <barrier>
5  #include <mutex>
6  #include <string>
7  #include <thread>
8
9  std::barrier workDone(6);
10 std::mutex coutMutex;
11
12 void synchronizedOut(const std::string& s) noexcept {
13     std::lock_guard<std::mutex> lo(coutMutex);
14     std::cout << s;
15 }
16
17 class FullTimeWorker {
18  public:
19     FullTimeWorker(std::string n): name(n) { };
20
21     void operator() () {
22         synchronizedOut(name + ": " + "Morning work done!\n");
23         workDone.arrive_and_wait();  // Wait until morning work is done
24         synchronizedOut(name + ": " + "Afternoon work done!\n");
25         workDone.arrive_and_wait();  // Wait until afternoon work is done
26
27     }
28  private:
29     std::string name;
30 };
31
```

```
32 class PartTimeWorker {
33  public:
34    PartTimeWorker(std::string n): name(n) { };
35
36    void operator() () {
37        synchronizedOut(name + ": " + "Morning work done!\n");
38        workDone.arrive_and_drop();  // Wait until morning work is done
39    }
40  private:
41    std::string name;
42 };
43
44 int main() {
45    std::cout << '\n';
46
47    FullTimeWorker herb("  Herb");
48    std::thread herbWork(herb);
49
50    FullTimeWorker scott("    Scott");
51    std::thread scottWork(scott);
52
53    FullTimeWorker bjarne("      Bjarne");
54    std::thread bjarneWork(bjarne);
55
56    PartTimeWorker andrei("        Andrei");
57    std::thread andreiWork(andrei);
58
59    PartTimeWorker andrew("          Andrew");
60    std::thread andrewWork(andrew);
61
62    PartTimeWorker david("            David");
63    std::thread davidWork(david);
64
65    herbWork.join();
66    scottWork.join();
67    bjarneWork.join();
68    andreiWork.join();
69    andrewWork.join();
70    davidWork.join();
71 }
```

Эта программа моделирует организацию, в которой есть два вида работников: одни трудятся на полную ставку (строка 17), другие (строка 32) – на полставки. Первые работают весь день, вторые – только до обеда. Соответственно, первые дважды вызывают функцию `arrive_and_wait` (строки 23 и 25), тогда как вторым довольно один раз (в строке 38) вызвать функцию `arrive_and_drop`. Именно этот вызов позволяет работнику пропустить вторую половину рабочего дня. На первой фазе (соответствует первой половине дня) число сотрудников, от которых ожидается работа, равно шести, а на второй (моделирует вторую половину дня) – трём. Результат работы программы представлен на рисунке.

```
x64 Native Tools Command Prompt for...

C:\Users\seminar>fullTimePartTimeWorkers.exe

  Herb: Morning work done!
      Bjarne: Morning work done!
        Andrei: Morning work done!
          Andrew: Morning work done!
            David: Morning work done!
    Scott: Morning work done!
    Scott: Afternoon work done!
      Bjarne: Afternoon work done!
  Herb: Afternoon work done!

C:\Users\seminar>
```

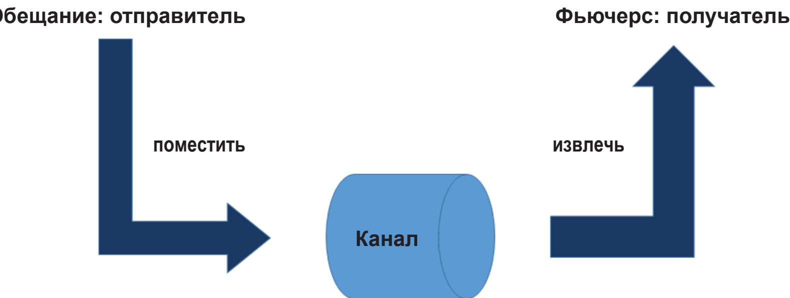
Работники на полную ставку и на полставки

3.9. Асинхронные задания

Помимо потоков, стандарт языка C++ включает ещё задания, позволяющие выполнять работу асинхронно. Чтобы использовать задания, нужно подключить заголовочный файл `<future>`. Задание характеризуется пакетом работы, которую предстоит выполнить, и обладает двумя частями: обещанием (promise) и фьючерсом (future). Они соединены собой в подобие канала, по которому проходят данные. Обещание запускает пакет работы и помещает результат его выполнения в канал; фьючерс извлекает этот результат из канала. Оба этих разъёма могут функционировать различных потоках. Существенно, что фьючерс позволяет забрать готовый результат работы в любое время после того, как он готов. Поэтому запуск задания посредством обещания и запрос результата через соответствующий фьючерс становятся независимыми друг от друга.

Задания как каналы передачи данных

Задания ведут себя подобно каналам, по которым данные проходят от входного разъёма до выходного. Один конец канала называется обещанием, другой – фьючерсом. Этими разъёмами может управлять один и тот же канал, а могут – разные. Обещание помещает данные в канал. Фьючерс в неопределённый момент в будущем извлекает из канала результат их преобразования.



Задания как каналы передачи данных

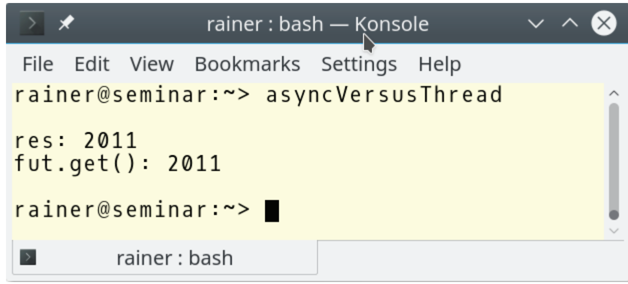
3.9.1. Отличие заданий от потоков

Задания – совсем не то же самое, что потоки. Начнём с примера.

Отличие асинхронных заданий от потоков

```
1  // asyncVersusThread.cpp
2
3  #include <future>
4  #include <thread>
5  #include <iostream>
6
7  int main(){
8    std::cout << std::endl;
9
10   int res;
11   std::thread t([&]{ res = 2000 + 11; });
12   t.join();
13   std::cout << "res: " << res << std::endl;
14
15   auto fut= std::async([]{ return 2000 + 11; });
16   std::cout << "fut.get(): " << fut.get() << std::endl;
17
18   std::cout << std::endl;
19 }
```

Дочерний поток `t` и асинхронная функция `fut` выполняют одно и то же вычисление: складывают числа 2000 и 11. Родительский поток получает результат из потока `t` через находящуюся в общем доступе переменную `res` и отображает её значение в строке 13. Вызов функции `std::async` в строке 15 создаёт канал передачи данных от источника (обещания) к получателю (фьючерсу). Далее, в строке 16, фьючерс используется для того, чтобы запросить данные из канала функцией `get`, и тем вынуждает выполнить асинхронное вычисление до конца. Поэтому вызов функции `get` блокирует выполнение главного потока. Ниже представлен результат работы программы.



Задание и поток

Разобрав этот пример, можно более точно описать различия между потоками и заданиями. Различия показаны в следующей таблице.

Отличие заданий от потоков

Критерий	Потоки	Задания
Основные сущности	Родительский и дочерний потоки	Обещание и фьючерс
Способ передачи данных	Общая переменная	Канал
Отдельный поток	Всегда	Иногда
Синхронизация	Функция `join` ожидает завершения потока	Функция `get` блокирует выполнение
Исключение в дочернем потоке (задании)	Оба потока завершаются вместе со всем процессом	Передаётся через обещание и фьючерс
Передаваемые данные	Значения	Значения, оповещения и исключения

Для работы с потоками нужно подключить заголовочный файл `<thread>`, а для работы с заданиями – `<future>`. Для обмена данными между родительским и дочерним потоками нужна переменная, к которой имеют доступ оба потока. Взаимодействие с заданием происходит через канал. Как следствие заданиям не нужны примитивы синхронизации наподобие мьютексов.

Если общей переменной, через которую обмениваются данными родительский и дочерний потоки, можно злоупотребить[1], взаимодействие с заданием носит более явный характер. Результат выполнения задания можно запросить через фьючерс только один раз, вызвав его функцию `get`. Повторный вызов этой функции на том же фьючерсе приводит к неопределённому поведению. Это не относится, однако, к классу `std::shared_future`, из которого значение можно запрашивать многократно.

Родительский поток ждёт завершения дочернего, вызывая функцию `join`. С фьючерсом нужно использовать функцию `get`, которая блокирует выполнение до тех пор, пока результат задания не станет доступен.

Если исключение возникает и не перехватывается в потоке, завершается и этот поток, и создавший его, и весь процесс. Для сравнения: обещание умеет отправить своё исключение фьючерсу, откуда его можно достать и обработать.

Обещание может обслуживать один или несколько фьючерсов. Оно может посылать значение, исключение или просто оповещение. Обещания можно использовать в качестве безопасной замены для переменной условия.

Самый простой способ создать фьючерс предоставляет функция `std::async`.

3.9.2. Функция `std::async`

Функция `std::async` ведёт себя как асинхронный вызов функции, заданной пользователем. В качестве аргументов функция `std::async` принимает вызываемый объект и аргументы для передачи ему. Функция `std::async` представляет собой вариадический шаблон и потому может принимать произвольное число аргументов. Результатом вызова функции `std::async` становится объект-фьючерс. Он играет роль разъёма, через который впоследствии можно

[1] В самом деле: поскольку два потока имеют к ней доступ, любой из них может, меняя значение переменной, влиять на поведение другого. – *Прим. перев.*

получить (вызвав функцию-член get) результат асинхронного выполнения пользовательской функции.

Всегда следует предпочитать асинхронные вызовы

Реализация стандартной библиотеки C++ сама решает, выполнять асинхронный вызов в отдельном потоке или нет. Это решение может зависеть от числа доступных ядер процессора, загруженности системы, размера пакета работы. Вызывая функцию std::async, программист лишь передаёт ей задание, которое должно быть выполнено. Вся работа по возможному созданию потока и управлению временем его жизни перекладывается на внутренние механизмы реализации.

Помимо того, при вызове функции std::async можно необязательным параметром передать политику запуска.

3.9.2.1. Политика запуска

Посредством политики можно в явном виде указать, каким образом реализации следует выполнить асинхронный вызов: в том же потоке, который создал вызов (std::launch::deferred), или в другом потоке (std::launch::async).

Строгие и ленивые вычисления

Строгое и ленивое вычисления – это две противоположных способа вычислять значение выражения. При строгой стратегии[1] выражение вычисляется немедленно, тогда как при ленивой[2] стратегии вычисление откладывается до тех пор, пока значение не станет необходимо. Строгое вычисление также называют жадным, а ленивое – отложенным, или вычислением по требованию. Ленивая стратегия вычислений часто помогает сберечь время и ресурсы процессора, предотвращает вычисление данных, которые могут не понадобиться в будущем.

Особенность выражения вида

```
auto fut = std::async(std::launch::deferred, ... )
```

состоит в том, что обещание не запускается немедленно. Вместо этого оно будет выполнено ленивым образом только в момент вызова fut.get(). Иными словами, обещание запускается только тогда, когда фьючерс в явном виде запрашивает его результат.

Строгое и ленивое вычисления фьючерса

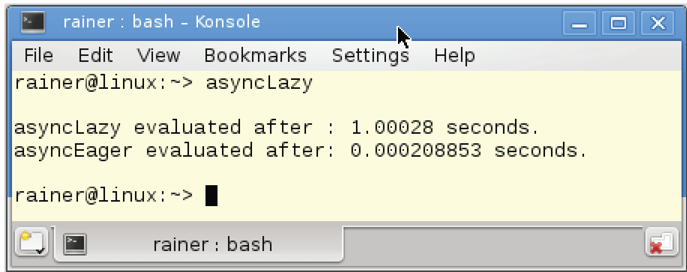
```
1  // asyncLazy.cpp
2
3  #include <chrono>
4  #include <future>
5  #include <iostream>
6
7  int main(){
```

[1] https://ru.wikipedia.org/wiki/Стратегия_вычисления.

[2] https://ru.wikipedia.org/wiki/Ленивые_вычисления.

```
8     std::cout << std::endl;
9
10    auto begin= std::chrono::system_clock::now();
11
12    auto asyncLazy=std::async(std::launch::deferred,
13                              []{ return std::chrono::system_clock::now(); });
14
15    auto asyncEager=std::async(std::launch::async,
16                               []{ return std::chrono::system_clock::now(); });
17
18    std::this_thread::sleep_for(std::chrono::seconds(1));
19
20    auto lazyStart= asyncLazy.get() - begin;
21    auto eagerStart= asyncEager.get() - begin;
22
23    auto lazyDuration= std::chrono::duration<double>(lazyStart).count();
24    auto eagerDuration=  std::chrono::duration<double>(eagerStart).count();
25
26    std::cout << "asyncLazy evaluated after : " << lazyDuration
27              << " seconds." << std::endl;
28    std::cout << "asyncEager evaluated after: " << eagerDuration
29              << " seconds." << std::endl;
30
31    std::cout << std::endl;
32  }
```

Оба вызова функции `std::async` (в строках 12 и 15) возвращают текущий момент времени. Однако первый вызов ленивый, тогда как второй – строгий. Односекундная задержка в строке 18 позволяет в этом убедиться. Вызов функции `get` в строке 20 запускает выполнение обещания, данного в строке 12, и полученный момент времени отделён одной секундой от начала выполнения программы. Для второго обещания дело обстоит иначе, поскольку его выполнение начинается немедленно в отдельном потоке. Пример выполнения программы показан на рисунке.



Строгое и ленивое вычисления

В общем случае нет необходимости хранить фьючерс в переменной.

3.9.2.2. Запустить и забыть

Особый случай составляют фьючерсы, о которых забывают сразу после создания. Они не сохраняются в каких-либо переменных и должны запускаться немедленно в момент создания. Существенно, что обещания таких фьючерсов должны выполняться в отдельном потоке, чтобы фьючерс мог начать работу немедленно. Для этого нужно использовать политику запуска `std::launch::async`.

Сравним поведение обычного фьючерса, присвоенного в переменную, и короткоживущего, который создаётся и в переменную не присваивается.

```
auto fut= std::async([]{ return 2011; });
std::cout << fut.get() << '\n';

std::async(std::launch::async,
    []{ std::cout << "fire and forget" << '\n'; });
```

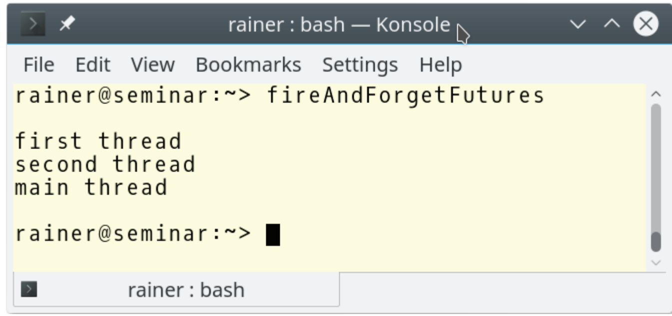
Короткоживущие фьючерсы выглядят удобными, но имеют существенный недостаток. Деструктор фьючерса, созданного вызовом функции `std::async`, ожидает, пока соответствующее обещание не будет выполнено. В данном случае ожидание мало отличается от блокирования. Таким образом, выполнение программы блокируется на деструкторе фьючерса. Это особенно отчётливо проявляется, когда фьючерс, созданный функцией `std::async`, не сохраняется в переменной. То, что, казалось бы, должно работать параллельно, выполняется на самом деле последовательно.

Короткоживущие фьючерсы

```
1  // blocking.cpp
2
3  #include <chrono>
4  #include <future>
5  #include <iostream>
6  #include <thread>
7
8  int main(){
9    std::cout << std::endl;
10
11   std::async(std::launch::async, []{
12     std::this_thread::sleep_for(std::chrono::seconds(2));
13     std::cout << "first thread" << std::endl;
14     });
15
16   std::async(std::launch::async, []{
17     std::this_thread::sleep_for(std::chrono::seconds(1));
18     std::cout << "second thread" << std::endl;}
19     );
20
21   std::cout << "main thread" << std::endl;
```

```
22
23   std::cout << std::endl;
24 }
```

В этой программе два асинхронных задания запускаются в отдельных потоках. Однако получившиеся в результате запуска объекты-фьючерсы не присвоены переменным. Как следствие деструкторы обоих фьючерсов блокируют выполнение главного потока, пока связанные с ними обещания не будут выполнены. В результате этого обещания выполняются в том же порядке, в котором они записаны в исходном коде. Последовательность выполнения остаётся неизменной, сколько бы ни запускать программу. Вот что выводит программа в результате своего выполнения.



Короткоживущие фьючерсы

В следующем разделе будет показано, что функция `std::async` предоставляет удобный механизм, чтобы переложить большую вычислительную задачу на плечи многочисленных исполнителей.

3.9.2.3. Параллельное вычисление скалярного произведения

Вычисление скалярного произведения векторов можно распределить между четырьмя асинхронными вызовами.

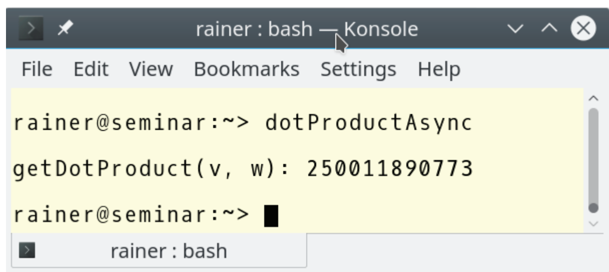
Скалярное умножение векторов с помощью асинхронных вызовов

```
1  // dotProductAsync.cpp
2
3  #include <iostream>
4  #include <future>
5  #include <random>
6  #include <vector>
7  #include <numeric>
8
9  using namespace std;
10
11 static const int NUM= 100000000;
12
```

```
13 long long getDotProduct(vector<int>& v, vector<int>& w){
14   auto vSize = v.size();
15
16   auto future1 = async([&]{ return inner_product(
17       &v[0], &v[vSize/4], &w[0], 0LL);
18   });
19
20   auto future1 = async([&]{ return inner_product(
21     &v[vSize/4], &v[vSize/2], &w[vSize/4], 0LL);
22   });
23
24   auto future1 = async([&]{ return inner_product(
25     &v[vSize/2], &v[vSize* 3/4], &w[vSize/2], 0LL);
26   });
27
28   auto future1 = async([&]{ return inner_product(
29     &v[vSize * 3/4], &v[vSize], &w[vSize * 3/4], 0LL);
30   });
31
32   return future1.get() + future2.get() + future3.get() + future4.get();
33 }
34
35
36 int main(){
37   cout << endl;
38
39   random_device seed;
40
41   // generator
42   mt19937 engine(seed());
43
44   // distribution
45   uniform_int_distribution<int> dist(0, 100);
46
47   // fill the vectors
48   vector<int> v, w;
49   v.reserve(NUM);
50   w.reserve(NUM);
51   for (int i=0; i< NUM; ++i){
52     v.push_back(dist(engine));
53     w.push_back(dist(engine));
54   }
55
56   cout << “getDotProduct(v, w): “ << getDotProduct(v, w) << endl;
57
58   cout << endl;
59 }
```

В этой программе используются средства из библиотеки random для работы со случайными числами, которая присутствует в стандарте начиная с версии C++ 11. Сначала создаются и заполняются случайными значениями два вектора, v и w (строки 49–55). Каждый вектор содержит сто миллионов

элементов. Функция `dist`, параметризованная генератором `engine`, создаёт в строках 53 и 54 случайные числа, равномерно распределённые в диапазоне от 0 до 100. Вычисление скалярного произведения происходит в функции `dotProduct` (строки 13–33). Под управлением стандартный алгоритм `std::inner_product` запускается асинхронно четыре раза, отдельно для каждой четверти длины вектора. В операторе `return` результаты этих четырёх асинхронных запусков суммируются. Ниже показан результат работы программы.



Скалярное умножение векторов с помощью асинхронных вызовов

В следующем разделе будет рассказано о типе `std::packaged_task`, который тоже часто используют для параллельных вычислений.

3.9.3. Тип std::packaged_task

Тип `std::packaged_task` представляет собой обёртку над вызываемым объектом, которая позволяет вызывать его асинхронно. Функция `get_future` позволяет получить связанный с обёрткой фьючерс. Перегруженная операция вызова для объекта `std::packaged_task` запускает выполнение завёрнутого в него вызываемого объекта.

Работа с объектами типа `std::packaged_task` обычно состоит из следующих четырёх шагов.

1. Завернуть действия, которые предполагается выполнить, в объект:

```
std::packaged_task<int(int, int)> sumTask(
    [](int a, int b){ return a + b; });
```

2. Создать фьючерс:

```
std::future<int> sumResult= sumTask.get_future();
```

3. Запустить вычисление:

```
sumTask(2000, 11);
```

4. Запросить результат вычисления:

```
sumResult.get();
```

Покажем эти четыре шага на следующем примере.

Параллельные вычисления с помощью типа std::packaged_task

```
1  // packagedTask.cpp
2
3  #include <utility>
4  #include <future>
5  #include <iostream>
6  #include <thread>
7  #include <deque>
8
9  class SumUp{
10   public:
11     int operator()(int beg, int end){
12       long long int sum{0};
13       for (int i = beg; i < end; ++i ) sum += i;
14       return sum;
15     }
16 };
17
18 int main(){
19   std::cout << std::endl;
20
21   SumUp sumUp1;
22   SumUp sumUp2;
23   SumUp sumUp3;
24   SumUp sumUp4;
25
26   // wrap the tasks
27   std::packaged_task<int(int, int)> sumTask1(sumUp1);
28   std::packaged_task<int(int, int)> sumTask2(sumUp2);
29   std::packaged_task<int(int, int)> sumTask3(sumUp3);
30   std::packaged_task<int(int, int)> sumTask4(sumUp4);
31
32   // create the futures
33   std::future<int> sumResult1 = sumTask1.get_future();
34   std::future<int> sumResult2 = sumTask2.get_future();
35   std::future<int> sumResult3 = sumTask3.get_future();
36   std::future<int> sumResult4 = sumTask4.get_future();
37
38   // push the tasks on the container
39   std::deque<std::packaged_task<int(int,int)>> allTasks;
40   allTasks.push_back(std::move(sumTask1));
41   allTasks.push_back(std::move(sumTask2));
42   allTasks.push_back(std::move(sumTask3));
43   allTasks.push_back(std::move(sumTask4));
44
45   int begin{1};
46   int increment{2500};
47   int end = begin + increment;
48
```

```
49   // perform each calculation in a separate thread
50   while (not allTasks.empty()){
51     std::packaged_task<int(int, int)> myTask =
52         std::move(allTasks.front());
53     allTasks.pop_front();
54     std::thread sumThread(std::move(myTask), begin, end);
55     begin = end;
56     end += increment;
57     sumThread.detach();
58   }
59
60   // pick up the results
61   auto sum = sumResult1.get() + sumResult2.get() +
62              sumResult3.get() + sumResult4.get();
63
64   std::cout << "sum of 0 .. 10000 = " << sum << std::endl;
65
66   std::cout << std::endl;
67 }
```

Назначение этой программы – вычислить сумму всех чисел от 0 до 10000 с помощью четырёх асинхронных заданий, выполняемых посредством объектов типа `std::packaged_task` в отдельных потоках. Чтобы получить окончательный результат, остаётся просуммировать промежуточные результаты четырёх фьючерсов. Конечно, если бы речь не шла об учебном примере, для решения этой задачи лучше подошла бы «формула Гаусса-малыша»[1].

Шаг 1: создать обёртки. Работа, которую предстоит выполнить, запаковывается в четыре объекта типа `std::packaged_task` (строки 27–30). Задания на работу представлены экземплярами класса `SumUp`, объявленного в строках 9–16. Собственно работа выполняется в перегруженной операции вызова (строки 11–15). Эта функция вычисляет и возвращает сумму чисел от `beg` до `end-1`. Объекты-обёртки, объявленные в строках 27–30, могут управлять такими вызываемыми объектами, у которых два аргумента и возвращаемое значение типа `int`.

Шаг 2: создать фьючерсы. Имея объекты-обёртки типа `std::packaged_task`, нужно теперь получить из них фьючерсы. Это делается в строках 33–36. В канале, по которому проходят данные, объект `std::packaged_task` играет роль обещания. В этом примере тип фьючерса указан явно (`std::future<int>`), хотя вывод типа можно поручить и компилятору, воспользовавшись ключевым словом `auto`.

---

[1] Частный случай формулы для суммы арифметической прогрессии с нулевым первым членом и единичной разностью, т. е. суммы натуральных чисел от 0 до некоторого *n*, выведенный Карлом Фридрихом Гауссом в возрасте девяти лет. Когда учитель задал ученикам вычислить сумму чисел от 1 до 100, ученик Гаусс заметил, что утомительных и кропотливых вычислений можно избежать, если суммируемые числа разбить на пары:

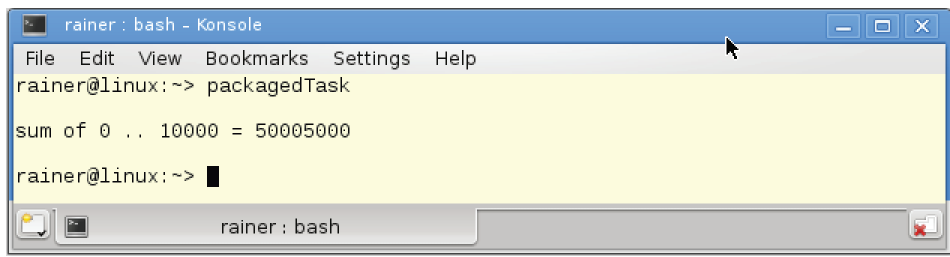
$(1 + 100) + (2 + 99) + (3 + 98) + \ldots + (50 + 51)$.

Каждая пара слагаемых даёт в сумме 101, всего же таких пар имеется 50. Следовательно, искомая сумма составляет 51 × 101. – *Прим. перев.*

Шаг 3: запустить вычисление. Теперь пора выполнить работу. Объекты типа `packaged_task` помещаются в контейнер `std::deque` (в строках 39–43). Затем каждое задание запускается на выполнение в строках 50–58. Для этого объект-обёртка из начала очереди перемещается во временную переменную (строка 51) и запускается в новом потоке (строка 54), после чего поток продолжает выполняться в фоновом режиме (строка 57). Тип `std::packaged_task` не поддерживает копирования, поэтому в строка 51 и 54 использована семантика перемещения. Такое же ограничение имеет место для обещаний, а также для фьючерсов и потоков. Единственным исключением из этого правила является тип `std::shared_future`.

Шаг 4: запросить результат вычисления. На заключительном шаге нужно получить результат вычислений из каждого фьючерса и просуммировать их – это происходит в строке 61.

Пример работы программы показан на рисунке.



Вычисление суммы с помощью типа `std::packaged_task`

В следующей таблице показан интерфейс класса `std::packaged_task`.

Функция-член	Описание
`swap`	Меняет местами содержимое двух объектов. Вызов `pack.swap(pack2)` эквивалентен вызову `std::swap(pack, pack2)`
`valid`	Проверяет, содержит ли обёртка в себе функцию (или вызываемый объект)
`get_future`	Возвращает фьючерс, связанный с асинхронным заданием
`make_ready_at_thread_exit`	Выполняет содержащуюся в обёртке функцию. Результат становится доступен по завершении потока
`reset`	Сбрасывает состояние объекта-обёртки, очищая сохранённый результат предыдущего запуска

В отличие от объектов типа `std::future` или `std::promise`, асинхронные задания типа `std::packaged_task` можно очищать и использовать повторно. В следующем примере показан такой способ обращения с заданиями.

Повторное использование асинхронных заданий

```
1  // packagedTaskReuse.cpp
2
3  #include <functional>
```

```
4  #include <future>
5  #include <iostream>
6  #include <utility>
7  #include <vector>
8
9  void calcProducts(
10     std::packaged_task<int(int, int)>& task,
11     const std::vector<std::pair<int, int>>& pairs)
12 {
13     for (auto& pair: pairs){
14         auto fut = task.get_future();
15         task(pair.first, pair.second);
16         std::cout
17             << pair.first
18             << " * "
19             << pair.second
20             << " = "
21             <<  fut.get()
22             << std::endl;
23         task.reset();
24     }
25 }
26
27 int main(){
28     std::cout << std::endl;
29
30     std::vector<std::pair<int, int>> allPairs;
31     allPairs.push_back(std::make_pair(1, 2));
32     allPairs.push_back(std::make_pair(2, 3));
33     allPairs.push_back(std::make_pair(3, 4));
34     allPairs.push_back(std::make_pair(4, 5));
35
36     std::packaged_task<int(int, int)> task{
37         [](int fir, int sec){ return fir * sec; }};
38
39     calcProducts(task, allPairs);
40
41     std::cout << std::endl;
42
43     std::thread t(calcProducts, std::ref(task), allPairs);
44     t.join();
45
46     std::cout << std::endl;
47 }
```

Функция `calcProducts`, объявленная в строке 9, получает два аргумента: асинхронное задание и вектор пар целых чисел. Задание используется для того, чтобы вычислять произведение каждой пары чисел (строка 15). В конце каждой итерации (строка 23) состояние объекта-задания очищается, чтобы на следующей итерации его можно было использовать заново. Функция `calcProducts` выполняется один раз в главном потоке (строка 39) и второй раз – в отдельном потоке (строка 43). Результат работы программы показан на рисунке.

```
File   Edit   View   Bookmarks   Settings   Help
rainer@linux:~> packagedTaskReuse

1 * 2 = 2
2 * 3 = 6
3 * 4 = 12
4 * 5 = 20

1 * 2 = 2
2 * 3 = 6
3 * 4 = 12
4 * 5 = 20

rainer@linux:~>
rainer : bash
```

Повторное использование асинхронных заданий

3.9.4. Типы `std::promise` и `std::future`

Шаблоны классов `std::promise` и `std::future` предоставляют программисту полную власть над асинхронными заданиями. Вместе они составляют могучую пару. Обещание (`std::promise`) позволяет поместить значение, исключение или просто оповещение в канал обработки данных. Одно обещание может поставлять данные одновременно для множества фьючерсов. В будущих версиях стандарта могут появиться расширения фьючерсов, поддерживающие композицию.

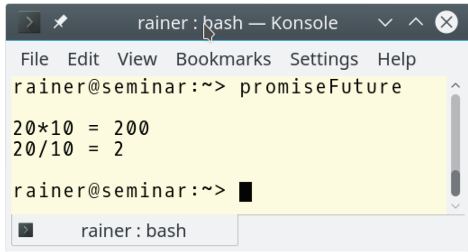
Ниже показан простейший пример использования обещаний и фьючерсов. Работа с двумя концами канала обработки данных может происходить в различных потоках –в этом случае канал обеспечивает взаимодействие между потоками.

Использование обещаний и фьючерсов

```
1  // promiseFuture.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <thread>
6  #include <utility>
7
8  void product(std::promise<int>&& intPromise, int a, int b){
9    intPromise.set_value(a*b);
10 }
11
12 struct Div{
13   void operator() (std::promise<int>&& intPromise, int a, int b) const {
14     intPromise.set_value(a/b);
15   }
16 };
17
18 int main(){
19   int a = 20;
```

```
20   int b = 10;
21 
22   std::cout << std::endl;
23 
24   // define the promises
25   std::promise<int> prodPromise;
26   std::promise<int> divPromise;
27 
28   // get the futures
29   std::future<int> prodResult = prodPromise.get_future();
30   std::future<int> divResult = divPromise.get_future();
31 
32   // calculate the result in a separate thread
33   std::thread prodThread(product, std::move(prodPromise), a, b);
34   Div div;
35   std::thread divThread(div, std::move(divPromise), a, b);
36 
37   // get the result
38   std::cout << "20*10 = " << prodResult.get() << std::endl;
39   std::cout << "20/10 = " << divResult.get() << std::endl;
40 
41   prodThread.join();
42 
43   divThread.join();
44 
45   std::cout << std::endl;
46 }
```

Поток `prodThread` создаётся в строке 33, в качестве параметров ему передаются функция `product`, определённая в строках 8–10, обещание `prodPromise` (строка 25) и два числа: `a` и `b`. Чтобы понять назначение этих параметров, нужно присмотреться к сигнатуре функции. Первым аргументом конструктор потока всегда принимает вызываемый объект. Здесь это функция `product`. Функции `product`, в свою очередь, нужны обещание, причём непременно по ссылке rvalue, и два числа. Они и составляют последние три аргумента, передаваемые при создании потока `prodThread`. Остальное довольно просто. Поток `divThread`, создаваемый в строке 35, делит одно число на другое. Для этой цели он использует функциональный объект – экземпляр `div` типа `Div` (строки 12–16). Наконец, вызов функции-члена `get` извлекает из фьючерсов результаты асинхронных вычислений. Работа программы показана на рисунке.



Деление и умножение чисел в асинхронных заданиях

3.9.4.1. Тип std::promise

Этот класс позволяет на вход асинхронного задания подать значение, исключение или оповещение. Кроме того, обещание может передать эти данные с задержкой, по завершении потока. Интерфейс данного класса показан в таблице.

Функция-член	Описание
swap	Обменять местами содержимое двух объектов
get_future	Возвращает фьючерс, связанный с обещанием
set_value	Присваивает обещанию значение
set_exception	Устанавливает в обещании исключение
set_value_at_thread_exit	Присваивает обещанию значение, которое станет доступно клиентам только после завершения текущего потока
set_exception_at_thread_exit	Устанавливает в обещании исключение, которое станет доступно клиентам только после завершения текущего потока

Если попытаться присвоить объекту-обещанию значение или исключение более одного раза, выбрасывается исключение типа std::future_error.

3.9.4.2. Тип std::future

Этот тип позволяет:

- получать значение из обещания;
- узнавать у объекта-обещания, доступно ли в нём значение;
- ждать оповещения от обещания, в том числе с заданной предельной продолжительностью ожидания или до заданного предельного момента времени;
- создавать фьючерс для общего доступа (типа std::shared_future).

Интерфейс класса показан в следующей таблице.

Функция-член	Описание
share	Возвращает объект типа std::shared_future. После этого вызов функции valid для текущего объекта возвращает значение false
get	Возвращает содержащийся во фьючерсе результат или выбрасывает исключение
valid	Проверяет, связан ли фьючерс с состоянием (которое может быть ещё не установлено). После вызова функции get возвращает значение false
wait	Ожидает появления результата (в том числе исключения)
wait_for	Ожидает появления результата (в том числе исключения), но не дольше, чем заданный промежуток времени
wait_until	Ожидает появления результата (в том числе исключения), но не дольше, чем до заданного момента времени

В отличие от функции wait, которая ничего не возвращает, функции wait_for и wait_until возвращают состояние фьючерса.

3.9.4.2.1. Состояние фьючерса

Функции-члены `wait_for` и `wait_until` классов `std::future` и `std::shared_future` возвращают состояние фьючерса. Всего имеется три возможных состояния, объявленных в стандартной библиотеке следующим образом:

Состояния фьючерсов

```
enum class future_status {
    ready,
    timeout,
    deferred
};
```

Смысл этих значений разъясняется в следующей таблице.

Состояние	Описание
deferred	Выполнение асинхронного задания не начато
ready	Результат задания готов
timeout	Истекло предельное время ожидания

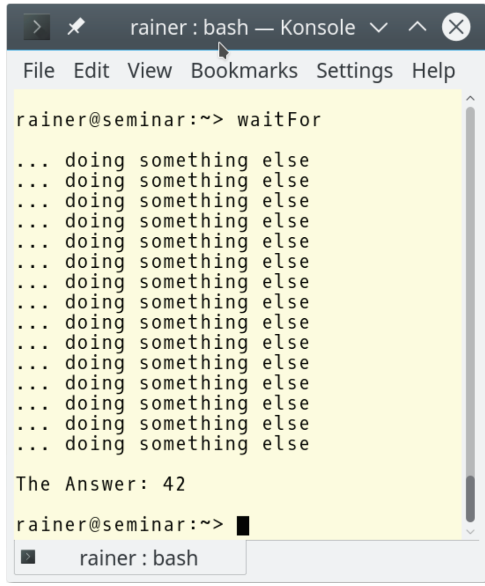
Благодаря наличию функций `wait_for` и `wait_until` фьючерс позволяет ожидать готовности соответствующего обещания.

Ожидание объекта-обещания

```
1  // waitFor.cpp
2
3  #include <iostream>
4  #include <future>
5  #include <thread>
6  #include <chrono>
7
8  using namespace std::literals::chrono_literals;
9
10 void getAnswer(std::promise<int> intPromise){
11     std::this_thread::sleep_for(3s);
12     intPromise.set_value(42);
13 }
14
15 int main(){
16    std::cout << std::endl;
17
18     std::promise<int> answerPromise;
19     auto fut = answerPromise.get_future();
20
21     std::thread prodThread(getAnswer, std::move(answerPromise));
22
23     std::future_status status{};
24     do {
25         status = fut.wait_for(0.2s);
26         std::cout << "... doing something else" << std::endl;
```

```
27      } while (status != std::future_status::ready);
28
29      std::cout << std::endl;
30
31      std::cout << "The Answer: " << fut.get() << '\n';
32
33      prodThread.join();
34
35      std::cout << std::endl;
36  }
```

Пока фьючерс fut ждёт обещанного значения на вход, он может делать что-то ещё (англ. *something else*), как показано на рисунке.



В ожидании обещания

Если бы у фьючерса fut запросили результат более одного раза, произошло бы исключение типа std::future_error. Между обещаниями и фьючерсами имеет место отношение «один к одному». В отличие от обычных фьючерсов, тип std::shared_future допускает совместную работу одного обещания с множеством фьючерсов.

3.9.5. Тип std::shared_future

Есть два способа создать объект этого типа.

1. Можно получить фьючерс из обещания prom и присвоить его в переменную типа std::shared_future:

```
std::shared_future fut = prom.get_future();
```

2. Кроме того, можно вызвать для фьючерса `fut` функцию `share`. После этого выражение `fut.valid()` будет возвращать значение `false`.

Фьючерс совместного доступа `std::shared_future` связан со своим объектом-обещанием. Класс `std::shared_future` обладает таким же интерфейсом[1], как и класс `std::future`. Если таких фьючерсов несколько, у каждого можно запрашивать результат независимо от других.

Покажем на примере особенности работы с типом `std::shared_future`. В следующей программе объекты этого типа создаются непосредственно.

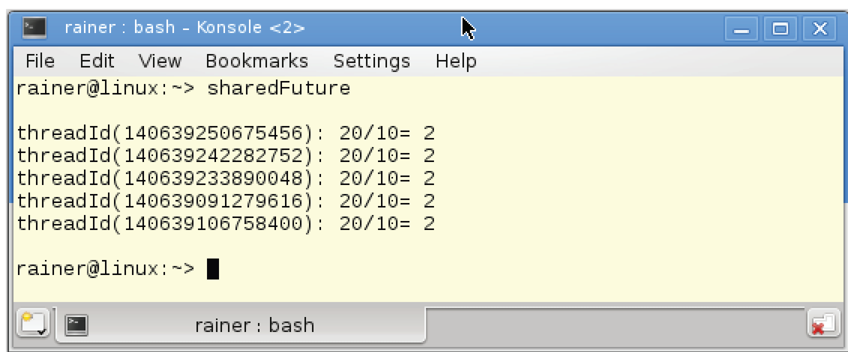
Создание нескольких фьючерсов из одного обещания

```
1  // sharedFuture.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <thread>
6  #include <utility>
7
8  std::mutex coutMutex;
9
10 struct Div{
11   void operator()(std::promise<int>&& intPromise, int a, int b){
12     intPromise.set_value(a/b);
13   }
14 };
15
16 struct Requestor{
17   void operator ()(std::shared_future<int> shaFut){
18     // lock std::cout
19     std::lock_guard<std::mutex> coutGuard(coutMutex);
20
21     // get the thread id
22     std::cout << "threadId(" << std::this_thread::get_id() << "): " ;
23
24     std::cout << "20/10= " << shaFut.get() << std::endl;
25   }
26 };
27
28 int main(){
29   std::cout << std::endl;
30
31   // define the promises
32   std::promise<int> divPromise;
33
34   // get the futures
35   std::shared_future<int> divResult = divPromise.get_future();
36
```

[1] В дополнение к интерфейсу класса `std::future`, класс `std::shared_future` обладает также конструктором копирования и копирующей операцией присваивания. – *Прим. перев.*

```
37    // calculate the result in a separat thread
38    Div div;
39    std::thread divThread(div, std::move(divPromise), 20, 10);
40
41    Requestor req;
42    std::thread sharedThread1(req, divResult);
43    std::thread sharedThread2(req, divResult);
44    std::thread sharedThread3(req, divResult);
45    std::thread sharedThread4(req, divResult);
46    std::thread sharedThread5(req, divResult);
47
48    divThread.join();
49
50    sharedThread1.join();
51    sharedThread2.join();
52    sharedThread3.join();
53    sharedThread4.join();
54    sharedThread5.join();
55
56    std::cout << std::endl;
57  }
```

В этом примере обещание и фьючерс используются в функциональных объектах, работающих в отдельных потоках. В строке 39 объект-обещание `divPromise` передаётся (путём перемещения) в поток `divThread`. Затем каждый из пяти потоков-потребителей получает свою копию фьючерса совместного доступа (строки 42–46). Подчеркнём ещё раз: в отличие от объектов типа `std::future`, которые можно только перемещать, объекты типа `std::shared_future` можно также копировать. Главный поток в строках 50–54 ждёт, пока дочерние потоки напечатают свои результаты и завершатся. Результат работы программы показан на рисунке.



Копирование фьючерсов совместного доступа

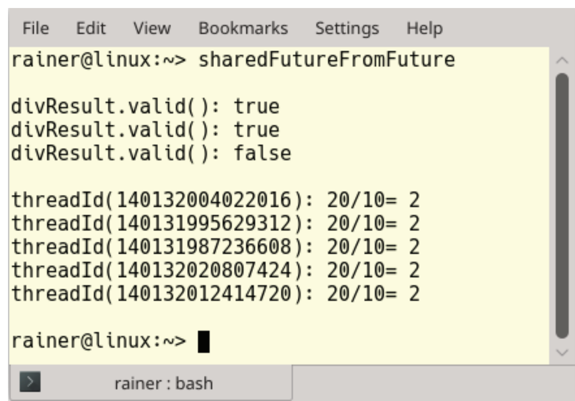
Как уже говорилось выше, фьючерс совместного доступа можно создавать также и вызовом функции-члена `share` для обычного фьючерса. Следующая программа демонстрирует этот способ.

Создание фьючерса общего доступа из обычного фьючерса

```
1  // sharedFutureFromFuture.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <thread>
6  #include <utility>
7
8  std::mutex coutMutex;
9
10 struct Div{
11   void operator()(std::promise<int>&& intPromise, int a, int b){
12     intPromise.set_value(a/b);
13   }
14 };
15
16 struct Requestor{
17   void operator ()(std::shared_future<int> shaFut){
18     // lock std::cout
19     std::lock_guard<std::mutex> coutGuard(coutMutex);
20
21     // get the thread id
22     std::cout << “threadId(“ << std::this_thread::get_id() << “): “ ;
23
24     std::cout << “20/10= “ << shaFut.get() << std::endl;
25   }
26 };
27
28 int main(){
29   std::cout << std::boolalpha << std::endl;
30
31   // define the promises
32   std::promise<int> divPromise;
33
34   // get the future
35   std::future<int> divResult = divPromise.get_future();
36
37   std::cout << “divResult.valid(): “ << divResult.valid() << std::endl;
38
39   // calculate the result in a separat thread
40   Div div;
41   std::thread divThread(div, std::move(divPromise), 20, 10);
42
43   std::cout << “divResult.valid(): “ << divResult.valid() << std::endl;
44
45   std::shared_future<int> sharedResult = divResult.share();
46
47   std::cout << “divResult.valid(): “ << divResult.valid() << “\n\n”;
48
49   Requestor req;
50   std::thread sharedThread1(req, sharedResult);
51   std::thread sharedThread2(req, sharedResult);
52   std::thread sharedThread3(req, sharedResult);
```

```
53    std::thread sharedThread4(req, sharedResult);
54    std::thread sharedThread5(req, sharedResult);
55
56    divThread.join();
57
58    sharedThread1.join();
59    sharedThread2.join();
60    sharedThread3.join();
61    sharedThread4.join();
62    sharedThread5.join();
63
64    std::cout << std::endl;
65  }
```

Первые два вызова функции `valid` для объекта `divResult` в строках 37 и 43 возвращают значение `true`. Ситуация меняется после вызова функции-члена `share` в строке 45: старый фьючерс теряет своё состояние, которое теперь передаётся под управление фьючерсов совместного доступа. Результат работы программы показан на рисунке.



Создание фьючерса совместного доступа

3.9.6. Обработка исключений в асинхронных заданиях

Если вызываемый объект, превращённый в асинхронное задание функцией `std::async` или классом `std::packaged_task`, выбрасывает исключение, это исключение запоминается в состоянии фьючерса и повторно выбрасывается из него при попытке взять из фьючерса результат функцией `get`. Код, использующий фьючерс, должен позаботиться об обработке исключения.

Класс `std::promise` позволяет запомнить исключение в состоянии асинхронного задания, для чего служит функция-член `set_exception`, в качестве аргумента которой обычно передают значение, полученное из функции `std::current_exception()`.

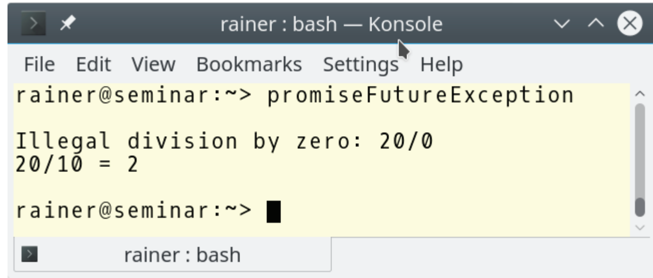
Деление на ноль ведёт к неопределённому поведению. В следующем примере функция `executeDivision` отображает результат деления или исключение.

Обработка исключений в асинхронных заданиях

```
1  // promiseFutureException.cpp
2
3  #include <exception>
4  #include <future>
5  #include <iostream>
6  #include <thread>
7  #include <utility>
8
9  struct Div{
10   void operator()(std::promise<int>&& intPromise, int a, int b) {
11     try {
12       if ( b==0 ) {
13           std::string errMess = std::string("Illegal division by zero: ") +s
14                                 std::to_string(a) + "/" + std::to_string(b);
15           throw std::runtime_error(errMess);
16       }
17       intPromise.set_value(a/b);
18     }
19     catch ( ... ) {
20       intPromise.set_exception(std::current_exception());
21     }
22   }
23 };
24
25 void executeDivision(int nom, int denom) {
26   std::promise<int> divPromise;
27   std::future<int> divResult= divPromise.get_future();
28
29   Div div;
30   std::thread divThread(div,std::move(divPromise), nom, denom);
31
32   // get the result or the exception
33   try {
34     std::cout
35         << nom
36         << "/"
37         << Denom
38         << " = "
39         << divResult.get()
40         << std::endl;
41   }
42   catch (std::runtime_error& e){
43     std::cout << e.what() << std::endl;
44   }
45
46   divThread.join();
```

```
47 }
48
49 int main() {
50   std::cout << std::endl;
51
52   executeDivision(20, 0);
53   executeDivision(20, 10);
54
55   std::cout << std::endl;
56 }
```

Обещание используется для обработки как нормального случая, так и случая, когда делитель равен нулю. В этом последнем случае вместо возвращаемого значения в обещание сохраняется исключение (строка 20). Фьючерс блоке `try-catch` (строки 33–44) обрабатывает это исключение. На рисунке представлен результат работы программы.



Обработка исключений в асинхронных заданиях

Функции std::current_exception и std::make_exception_ptr

Функция `std::current_exception` превращает текущее исключение в значение типа `std::exception_ptr`. Этот тип позволяет хранить либо объект-исключение, либо ссылку на него. Если данную функцию вызывать в такой момент, когда никакое исключение не находится в процессе обработки, она возвращает пустое значение типа `std::exception_ptr`.

Вместо того чтобы выбрасывать исключение в блоке `try`, сразу ловить его в блоке `catch` и превращать в значение типа `std::exception_ptr`, можно воспользоваться функцией `std::make_exception_ptr`, например:

```
intPromise.set_exception(
  std::make_exception_ptr(
    std::runtime_error(errMess)));
```

Если объект типа `std::promise` разрушается по окончании времени жизни, притом что для него не вызывалась функция-член из семейства `set_`, или если объект типа `std::packaged_task` разрушается, не будучи вызванным, в связанном с ними состоянии сохраняется исключение типа `std::future_error` с кодом `std::future_errc::broken_promise`.

3.9.7. Оповещения

Асинхронные задания представляют собой безопасную замену переменным условия. У этих механизмов довольно много общего, обещания и фьючерсы также можно использовать для синхронизации потоков. В большинстве случаев обещаниям и фьючерсам следует отдавать предпочтение.

Прежде чем рассмотреть пример программы, приведём сравнение двух механизмов оповещения.

Критерий	Переменные условия	Асинхронные задания
Многократная синхронизация	Да	Нет
Критическая секция	Да	Нет
Обработка ошибок на приёмной стороне	Нет	Да
Опасность ложных пробуждений	Да	Нет
Опасность утерянного пробуждения	Да	Нет

Преимущество переменных условия перед механизмом обещаний и фьючерсов состоит в том, что одну переменную условия можно использовать многократно. Обещание, напротив, может послать оповещение лишь один раз; поэтому для имитации функциональных возможностей переменных условия пришлось бы использовать множество пар обещание–фьючерс. Однако если требуется однократная синхронизация потоков, использовать переменную условия оказывается гораздо более хлопотно. Паре из обещания и фьючерса не нужна переменная в общем доступе, а значит, не нужен и вспомогательный объект-блокировщик, она не подвержена опасности утерянных и ложных пробуждений. Кроме того, асинхронные задания умеют обрабатывать исключения. Таким образом, имеется множество причин предпочесть асинхронные задания переменным условия.

Помнит ли читатель из предыдущих разделов, сколь сложны в использовании переменные условия? Если нет, ниже показаны основные части, из которых состоит синхронизация двух потоков.

```
void waitingForWork() {
    std::cout << “Worker: Waiting for work.” << ‘\n’;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << “Work done.” << ‘\n’;
}

void setDataReady() {
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << “Sender: Data is ready.” << ‘\n’;
    condVar.notify_one();
}
```

Функция `setDataReady` отправляет оповещение, а функция `waitingForWork` ждёт его. Ниже показано, как добиться того же результата с помощью асинхронных заданий.

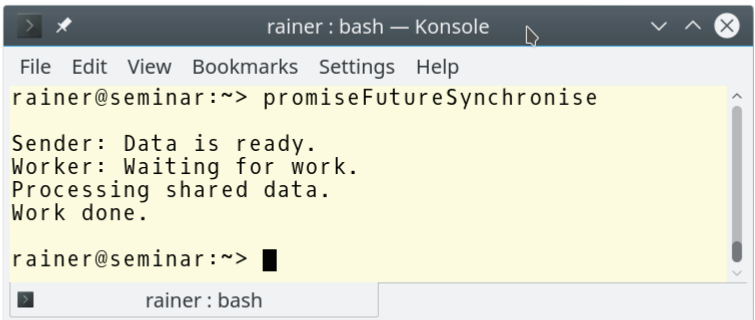
Синхронизация потоков посредством асинхронных заданий

```
1  // promiseFutureSynchronize.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <utility>
6
7
8  void doTheWork(){
9    std::cout << "Processing shared data." << std::endl;
10 }
11
12 void waitingForWork(std::future<void>&& fut){
13   std::cout << "Worker: Waiting for work." << std::endl;
14   fut.wait();
15   doTheWork();
16   std::cout << "Work done." << std::endl;
17 }
18
19 void setDataReady(std::promise<void>&& prom){
20     std::cout << "Sender: Data is ready."  << std::endl;
21     prom.set_value();
22 }
23
24 int main(){
25   std::cout << std::endl;
26
27   std::promise<void> sendReady;
28   auto fut = sendReady.get_future();
29
30   std::thread t1(waitingForWork, std::move(fut));
31   std::thread t2(setDataReady, std::move(sendReady));
32
33   t1.join();
34   t2.join();
35
36   std::cout << std::endl;
37 }
```

Эта программа выглядит довольно просто. Из обещания `sendReady`, объявленного в строке 27, программа получает фьючерс `fut` (строка 28). Поскольку обещание имеет тип `void`, оно может передавать не значения, а лишь оповещения. Объекты, составляющие передающий и приёмный концы канала, перемещаются в потоки `t1` и `t2` (строки 30 и 31). Первый поток с помощью

функции wait фьючерса fut в строке 14 ждёт оповещения, которое второй поток отправляет через объект-обещание prom функцией set_value (строка 21).

Устройство и поведение этой программы – такие же, как у соответствующей программы из раздела 3.5 о переменных условия. Пример работы программы показан на рисунке.



Использование асинхронного задания вместо переменной условия

3.10. Синхронизированные потоки вывода (стандарт C++ 20)

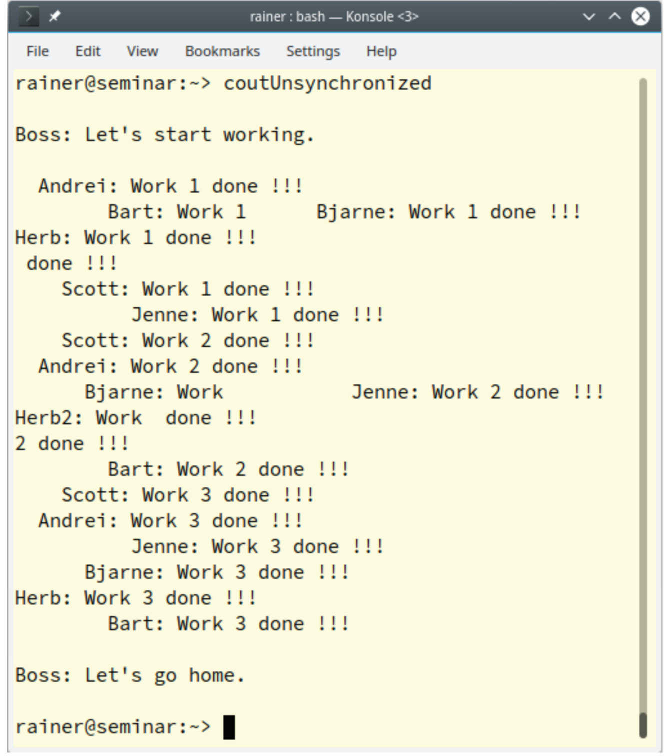
Что происходит, если отправлять данные в поток вывода std::cout без всякой синхронизации?

Несинхронизированный доступ к потоку std::cout

```
1  // coutUnsynchronised.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  class Worker{
8  public:
9    Worker(std::string n):name(n){};
10     void operator() (){
11       for (int i = 1; i <= 3; ++i){
12         // begin work
13         std::this_thread::sleep_for(std::chrono::milliseconds(200));
14         // end work
```

```
15          std::cout
16              << name
17              << ": "
18              << "Work "
19              << i
20              << " done !!!"
21              << std::endl;
22        }
23      }
24  private:
25    std::string name;
26  };
27
28  int main(){
29    std::cout << std::endl;
30
31    std::cout << "Boss: Let's start working.\n\n";
32
33    std::thread herb= std::thread(Worker("Herb"));
34    std::thread andrei= std::thread(Worker("  Andrei"));
35    std::thread scott= std::thread(Worker("    Scott"));
36    std::thread bjarne= std::thread(Worker("      Bjarne"));
37    std::thread bart= std::thread(Worker("        Bart"));
38    std::thread jenne= std::thread(Worker("          Jenne"));
39
40    herb.join();
41    andrei.join();
42    scott.join();
43    bjarne.join();
44    bart.join();
45    jenne.join();
46
47    std::cout << "\n" << "Boss: Let's go home." << std::endl;
48
49    std::cout << std::endl;
50  }
```

У начальника есть шесть работников (строки 33–38). Каждому работнику нужно выполнить три задания, каждое из которых занимает 1/5 секунды (строка 13). Выполнив каждое задание, работник громко кричит об этом в стандартный поток вывода (строка 15). Когда все работники сделают всю работу, начальник отпускает их домой. Результат работы программы можно видеть на рисунке. Сколько путаницы для такой простой организации! Каждый работник выкрикивает свой доклад, не обращая внимания на своих коллег!



Несинхронизированный вывод в поток `std::cout`

Стоит напомнить, что причудливое чередование операций вывода в поток – лишь дефект наблюдаемого поведения программы, а не гонка данных.

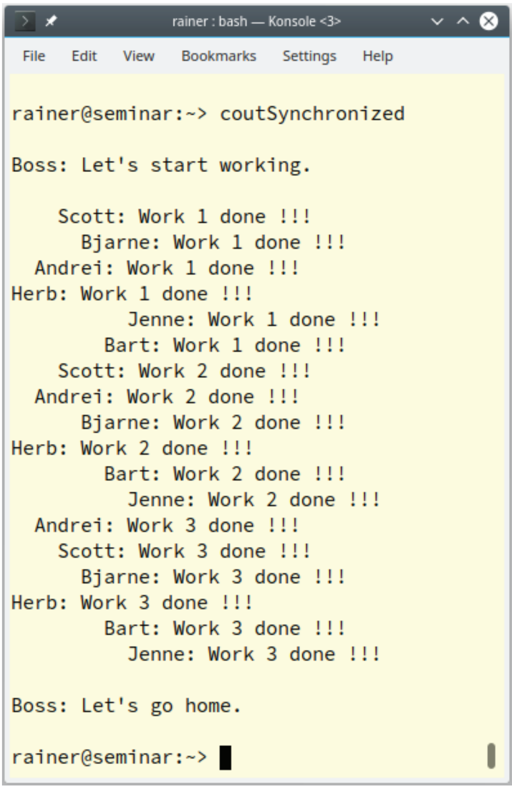
Как справиться с этой трудностью? Стандарт C++ 11 предлагал единственный ответ: воспользоваться объектом-синхронизатором наподобие `lock_guard` для синхронизации доступа потоков к объекту `std::cout`, как показано в следующей программе.

Синхронизированный доступ к потоку `std::cout`

```
1  // coutSynchronised.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex coutMutex;
9
10 class Worker{
11 public:
```

```
12   Worker(std::string n):name(n){};
13
14     void operator() (){
15       for (int i = 1; i <= 3; ++i) {
16         // begin work
17         std::this_thread::sleep_for(std::chrono::milliseconds(200));
18         // end work
19         std::lock_guard<std::mutex> coutLock(coutMutex);
20         std::cout
21             << name
22             << ": "
23             << "Work "
24             << i
25             << " done !!!"
26             << std::endl;
27       }
28     }
29 private:
30   std::string name;
31 };
32
33 int main(){
34   std::cout << std::endl;
35
36   std::cout << "Boss: Let's start working." << "\n\n";
37
38   std::thread herb= std::thread(Worker("Herb"));
39   std::thread andrei= std::thread(Worker("  Andrei"));
40   std::thread scott= std::thread(Worker("    Scott"));
41   std::thread bjarne= std::thread(Worker("      Bjarne"));
42   std::thread bart= std::thread(Worker("        Bart"));
43   std::thread jenne= std::thread(Worker("          Jenne"));
44
45   herb.join();
46   andrei.join();
47   scott.join();
48   bjarne.join();
49   bart.join();
50   jenne.join();
51
52   std::cout << "\n" << "Boss: Let's go home." << std::endl;
53
54   std::cout << std::endl;
55 }
```

Мьютекс `coutMutex`, объявленный в строке 8, защищает глобальный объект `std::cout` от одновременного доступа. Оборачивание этого мьютекса во вспомогательный объект типа `std::lock_guard` гарантирует, что мьютекс будет захвачен в его конструкторе (строка 19) и освобождён в деструкторе при выходе из блока (строка 27).



Синхронизированный вывод в поток `std::cout`

С появлением стандарта C++ 20 синхронизированный вывод в поток `std::cout` становится проще простого. Тип `std::basic_syncbuf` представляет собой обёртку над типом[1] `std::basic_streambuf`. Все посылаемые в него данные он не вывоит, а накапливает. Объект-обёртка посылает накопленное содержимое в находящийся под её управлением объект только в момент своего разрушения. Следовательно, данные выводятся в виде непрерывной последовательности символов, и операции вывода из различных потоков между собой не перемешиваются.

Благодаря шаблону класса `std::basic_osyncstream` становится возможным напрямую писать в поток вывода с синхронизацией. В стандарте C++ 20 определены две специализации этого шаблона: для символов типа `char` и типа `wchar_t`.

```
std::osyncstream      std::basic_osyncstream<char>
std::wosyncstream     std::basic_osyncstream<wchar_t>
```

Эти классы-обёртки позволяют создавать именованные синхронизированные потоки вывода. Ниже показан результат переделки разобранной ранее несинхронизированной программы с использованием синхронных потоков.

[1] https://en.cppreference.com/w/cpp/io/basic_streambuf.

Синхронизированный вывод с помощью обёртки `std::osyncstream`

```
1  // synchronizedOutput.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <syncstream>
6  #include <thread>
7
8  class Worker{
9  public:
10   Worker(std::string n):name(n) {};
11     void operator() (){
12       for (int i = 1; i <= 3; ++i) {
13         // begin work
14         std::this_thread::sleep_for(std::chrono::milliseconds(200));
15         // end work
16         std::osyncstream syncStream(std::cout);
17         syncStream
18             << name
19             << “: “
20             << “Work “
21             << i
22             << “ done !!!”
23             << ‘\n’;
24       }
25     }
26 private:
27   std::string name;
28 };
29
30 int main() {
31   std::cout << ‘\n’;
32
33   std::cout << “Boss: Let’s start working.\n\n”;
34
35   std::thread herb= std::thread(Worker(“Herb”));
36   std::thread andrei= std::thread(Worker(“  Andrei”));
37   std::thread scott= std::thread(Worker(“    Scott”));
38   std::thread bjarne= std::thread(Worker(“      Bjarne”));
39   std::thread bart= std::thread(Worker(“       Bart”));
40   std::thread jenne= std::thread(Worker(“          Jenne”));
41
42   herb.join();
43   andrei.join();
44   scott.join();
45   bjarne.join();
46   bart.join();
47   jenne.join();
48
49   std::cout << “\n” << “Boss: Let’s go home.” << ‘\n’;
50
51   std::cout << ‘\n’;
52 }
```

Единственное отличие этой программы от программы coutUnsynchronized.cpp состоит в том, что теперь поток std::cout обёрнут в объект типа std::osyncstream (строка 16). Чтобы пользоваться в программе типом std::osyncstream, нужно подключить заголовочный файл <syncstream>. Когда объект-обёртка заканчивает своё существование с выходом из блока в строке 24, символы из него за одну операцию отправляются в поток std::cout. Стоит отметить, что обращения к потоку std::cout в функции main происходят тогда, когда все работники закончили свою работу, и эти обращения синхронизировать не нужно.

Поскольку объект syncStream, объявленный в строке 16, используется только один раз, здесь мог бы лучше подойти временный объект. В следующем фрагменте кода показано незначительное изменение.

```
void operator() (){
  for (int i = 1; i <= 3; ++i) {
    // begin work
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // end work
    std::osyncstream (std::cout)
        << name
        << ": "
        << "Work "
        << i
        << " done !!!"
        << '\n';
  }
}
```

В классе std::basic_osyncstream содержатся две интересные функции:

- emit – осуществляет вывод всех накопленных символов и очищает буфер;
- get_wrapped – возвращает указатель на буфер, завёрнутый в буфер-синхронизатор.

На сайте cppreference.com приводится пример[1] того, как можно управлять последовательностью вывода в поток с помощью функции-члена get_wrapped.

Управление последовательностью вывода

```
// sequenceOutput.cpp

#include <syncstream>
#include <iostream>

int main() {
  std::osyncstream bout1(std::cout);
  bout1 << "Hello, ";
  {
```

[1] https://en.cppreference.com/w/cpp/io/basic_osyncstream/get_wrapped.

```
    std::osyncstream(bout1.get_wrapped())
        << "Goodbye, "
        << "Planet!"
        << '\n';
  } // emits the contents of the temporary buffer

  bout1 << "World!" << '\n';
} // emits the contents of bout1
```

Результат работы этой программы показан на рисунке.

```
Goodbye, Planet!
Hello, World!
```

Управление последовательностью вывода

3.11. Краткие итоги

В стандартной библиотеке языка C++ поддерживается два вида потоков: обычные потоки `std::thread` (начиная со стандарта C++ 11) и усовершенствованные `std::jthread` (со стандарта C++ 20).

- Объект типа `std::thread` получает в конструкторе вызываемый объект и немедленно запускает его на выполнение в отдельном потоке. Создатель потока отвечает за его дальнейшую судьбу. Это означает, что он должен либо дождаться завершения потока с помощью функции-члена `join`, либо отсоединить поток от объекта функцией `detach`. Объект находится в присоединяемом состоянии, если к нему не применялась функция `join` или `detach`. Если объект типа `std::thread` находится в присоединяемом состоянии на момент вызова деструктора, деструктор вызывает функцию `std::terminate`, и выполнение всей программы завершается аварийно.
- Интерфейс класса `std::jthread` расширен по сравнению с классом `std::thread`. Он автоматически ждёт завершения потока в деструкторе и поддерживает кооперативное прерывание потока.

Операции, выполняемые разными потоками над находящейся в совместном доступе переменной, нужно координировать, если хотя бы один поток её модифицирует. Мьютекс позволяет добиться того, что в любой момент времени не более одного потока имеет доступ к общей переменной. Чтобы гарантировать автоматическое освобождение мьютекса и тем самым избежать множества возможных проблем, мьютекс нужно обернуть в объект-блокировщик. Стандартная библиотека языка C++ содержит несколько видов мьютексов и блокировщиков.

Если данные доступны нескольким потокам, но только для чтения, достаточно лишь проинициализировать их потокобезопасным способом. Стандарт языка C++ предоставляет для этого несколько механизмов, включая

константные выражения, статические переменные внутри блока, а также функцию `std::call_once` вместе с флагом `std::once_flag`.

Объявление переменной со спецификатором `thread_local` означает, что каждый поток получает свою копию этой переменной. Время жизни такой копии ограничено временем выполнения соответствующего потока.

Переменные условия позволяют синхронизировать потоки путём обмена оповещениями. Один поток выступает отправителем сообщения, другой – получателем. Данный механизм позволяет заблокировать поток-получатель в ожидании сигнала от потока-отправителя.

В классы `std::jthread` и `std::condition_variable_any` заложена поддержка кооперативного прерывания. Для кооперативного прерывания потоков используются классы `std::stop_source`, `std::stop_token` и `std::stop_callback`.

Семафоры – это механизм синхронизации, предназначенный для управления одновременным доступом потоков к общему ресурсу. Счётчик семафора инициализируется в конструкторе. Захват семафора уменьшает значение счётчика, а освобождение – увеличивает. Если поток пытается захватить семафор, когда его счётчик на нуле, поток блокируется до тех пор, пока другой поток, освободив семафор, не увеличит его значение.

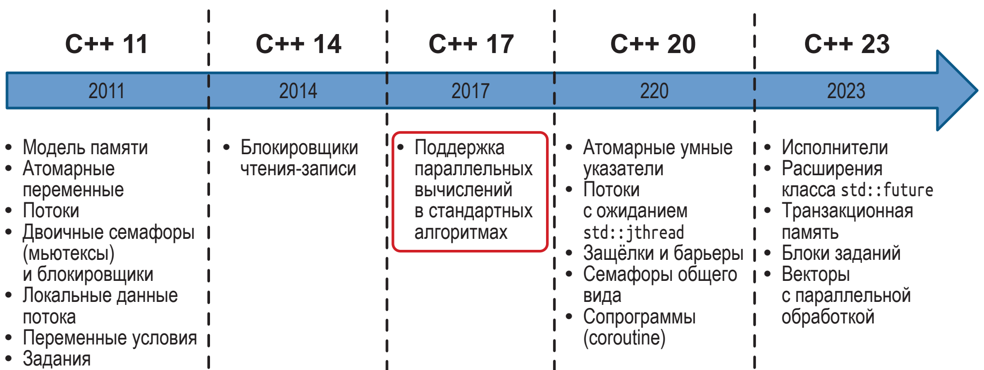
Защёлки и барьеры также представляют собой средства координации потоков, которые позволяют блокировать некоторые потоки до тех пор, пока счётчик не достигнет нуля. Начальное значение счётчика задаётся в конструкторе.

Задания имеют много общего с потоками. Если поток явным образом запускается в момент создания, то задание – это лишь то, что должно быть выполнено когда-либо в будущем. Реализация стандартной библиотеки сама управляет временем жизни заданий, как в простейшем случае – когда задания создаются функцией `std::async`. Задания похожи на каналы, по которым данные передаются от источника к приёмнику, преобразовываясь по пути. С их помощью можно организовать безопасный обмен данными между потоками. Обещание на одном конце канала помещает в него данные, а фьючерс на другом конце их извлекает. Передаваемыми по каналу данными могут быть значения, исключения или просто оповещения. Помимо шаблона функции `std::async`, в стандарте языка C++ есть также шаблоны классов `std::promise` и `std::future`, которые позволяют полнее управлять заданиями.

Стандарт языка C++ поддерживает синхронизированные потоки вывода. Синхронизированный поток накапливает выводимые данные в буфере и отправляет всё содержимое буфера в настоящий поток в момент уничтожения. Следовательно, все данные, помещённые в такой поток, превращаются в непрерывную последовательность символов, которые не перемешиваются с другими данными.

4. Параллельные алгоритмы в стандартной библиотеке

В стандартной библиотеке языка C++ содержится более ста алгоритмов для поиска, подсчёта и преобразования элементов в контейнерах и их диапазонах. С выходом стандарта C++ 17 появились 69 новых перегрузок для имеющихся алгоритмов и 8 полностью новых алгоритмов, способных работать параллельно. Новые перегрузки и новые алгоритмы принимают в качестве аргумента так называемую политику выполнения.



Параллельные алгоритмы в стандарте C++ 17

С помощью политики выполнения программист может сообщить реализации, должен ли алгоритм выполняться последовательно, параллельно или параллельно с векторизацией. Для того чтобы воспользоваться политикой выполнения, нужно подключить заголовочный файл `<execution>`.

4.1. Политики выполнения

Стандартом определены следующие три[1] политики:

- `std::execution::sequenced_policy`;
- `std::execution::parallel_policy`;
- `std::execution::parallel_unsequenced_policy`.

Соответствующие им значения-константы[2] определяют, должен ли алгоритм выполняться последовательно, параллельно или векторизированно:

- `std::execution::seq` предписывает последовательное, в один поток, выполнение алгоритма;
- `std::execution::par` разрешает выполнение в несколько потоков;
- `std::execution::par_unseq` разрешает выполнять алгоритм в несколько потоков, чередовать итерации циклов в этих потоках и, кроме того, разрешает использование расширенных векторных инструкций ОКМД[3].

Таким образом, передача в функцию объектов `std::execution::par` и `std::execution::par_unseq` в качестве аргументов позволяет алгоритму выполняться в параллельном или параллельно-векторном режиме. Подчеркнём, что политика означает именно разрешение параллельной работы, а не требование.

В следующем фрагменте кода показано применение трёх политик выполнения.

Политики выполнения

```
1  std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3  // standard sequential sort
4  std::sort(v.begin(), v.end());
5
6  // sequential execution
7  std::sort(std::execution::seq, v.begin(), v.end());
8
```

[1] Стоит упомянуть, что в стандарте C++ 20 появилась также четвертая политика выполнения, кодируемая типом `unsequenced_policy` и константой `unseq` этого типа, которая предписывает выполнять алгоритм в один поток, но разрешает при этом пользоваться расширенными векторными инструкциями. – *Прим. перев.*

[2] Следует подчеркнуть, что первые три (`sequenced_policy`, `parallel_policy`, `parallel_unsequenced_policy`) суть типы данных, тогда как последние три (`seq`, `par`, `par_unseq`) – константы этих типов. Типы данных, используемые для политик выполнения, фиктивны в том смысле, что в объектах этих типов не содержится никаких данных и, следовательно, передавая объекты-политики в функции стандартной библиотеки, программист не передаёт им никаких данных на этапе выполнения. Типы-политики нужны исключительно на этапе компиляции для выбора соответствующей перегрузки. – *Прим. перев.*

[3] ОКМД (одиночные команды, множественные данные, англ. SIMD – single instructions, multiple data) – способ аппаратной поддержки параллельных вычислений, при котором одна арифметическая инструкция применяется не к одиночному значению, а к набору (вектору) данных одновременно. – *Прим. перев.*

```
9  // permitting parallel execution
10 std::sort(std::execution::par, v.begin(), v.end());
11
12 // permitting parallel and vectorized execution
13 std::sort(std::execution::par_unseq, v.begin(), v.end());
```

В строке 4 показано использование классического последовательного варианта функции `std::sort`. Кроме того, стандарт C++ 17 позволяет явно указывать нужный вариант алгоритма: последовательный (строка 7), параллельный (строка 10) или параллельный с векторизацией (строка 13).

С помощью шаблона `std::is_execution_policy` можно узнать, является ли тип `T` стандартной или зависящей от реализации политикой выполнения: для этого нужно взять значение выражения

```
std::is_execution_policy<T>::value
```

Это выражение имеет значение `true`, если тип `T` есть один из типов `sequenced_policy`, `parallel_policy`, `parallel_unsequenced_policy` или тип политики выполнения, специфичный для конкретной реализации. В противном случае данное выражение даёт значение `false`.

4.1.1. Параллельное и векторизованное выполнение

Выполняется ли алгоритм параллельным и векторизованным способом, зависит от множества факторов. Например, это зависит от того, поддерживает ли процессор и операционная система инструкции типа ОКОД (англ. *SIMD*). Также это зависит от компилятора и от уровня оптимизации, использованного при трансляции кода.

В следующем примере показан простой цикл, заполняющий вектор значениями.

Заполнение вектора значениями

```
1  const int SIZE= 8;
2
3  int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
4  int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
5
6  int main(){
7      for (int i = 0; i < SIZE; ++i) {
8          res[i] = vec[i]+5;
9      }
10 }
```

Строка 8 играет в этом примере ключевую роль. Благодаря средству Compiler Explorer[1] можно в подробностях изучить машинные инструкции, которые генерирует компилятор clang 3.6.

[1] https://godbolt.org/.

4.1.1.1. Код без оптимизации

Ниже показаны ассемблерные инструкции, которые выдаёт компилятор с выключенной оптимизацией. Все сложения выполняются последовательно.

```
movslq  -8(%rbp), %rax
movl    vec(,%rax,4), %ecx
addl    $5, %ecx
movslq  -8(%rbp), %rax
movl    %ecx, res(,%rax,4)
```

Последовательное выполнение

4.1.1.2. Максимальная оптимизация

При наивысшем уровне оптимизации `-O3` генерируется машинный код, использующий специальные регистры наподобие `xmm0`, в которых может храниться 128 бит, или 4 целых числа. В этом случае сложение выполняется одновременно для четырёх элементов вектора.

```
movdqa  .LCPI0_0(%rip), %xmm0    # xmm0 = [5,5,5,5]
movdqa  vec(%rip), %xmm1
paddd   %xmm0, %xmm1
movdqa  %xmm1, res(%rip)
paddd   vec+16(%rip), %xmm0
movdqa  %xmm0, res+16(%rip)
xorl    %eax, %eax
```

Выполнение с векторизацией

Перегруженная версия алгоритма без политики выполнения отличается от перегрузки с явно заданной последовательной политикой `std::execution::seq` в одном отношении: обработкой исключений.

4.1.2. Обработка исключений

Если во время выполнения алгоритма с явно заданной политикой выполнения возникает исключение, вызывается функция `std::terminate`[1]. Эта функция вызывает обработчик `std::terminate_handler`[2]. В свою очередь, обработчик по умолчанию вызывает функцию `std::abort`[3], которая аварийно завершает

[1] https://en.cppreference.com/w/cpp/error/terminate.
[2] https://en.cppreference.com/w/cpp/error/terminate_handler.
[3] https://en.cppreference.com/w/cpp/utility/program/abort.

программу. Обработка исключений составляет различие между вызовом алгоритма без какой-либо политики выполнения и вызовом с явно заданной последовательной политикой `std::execution::seq`. Алгоритм без политики выполнения пропускает исключение наружу, тем самым давая возможность его обработать. Следующая программа иллюстрирует это различие.

Политики выполнения и обработка исключений

```
1  // exceptionExecutionPolicy.cpp
2
3  #include <algorithm>
4  #include <execution>
5  #include <iostream>
6  #include <stdexcept>
7  #include <string>
8  #include <vector>
9
10 int main(){
11     std::cout << std::endl;
12
13     std::vector<int> myVec{1, 2, 3, 4, 5};
14
15     try{
16         std::for_each(myVec.begin(), myVec.end(),
17             [](int){ throw std::runtime_error("Without execution policy"); }
18         );
19     }
20     catch(const std::runtime_error& e){
21         std::cout << e.what() << std::endl;
22     }
23
24     try{
25         std::for_each(std::execution::seq, myVec.begin(), myVec.end(),
26             [](int){ throw std::runtime_error("With execution policy"); }
27         );
28     }
29     catch(const std::runtime_error& e){
30         std::cout << e.what() << std::endl;
31     }
32     catch(...){
33         std::cout << "Catch-all exceptions" << std::endl;
34     }
35 }
```

Обработчик в строке 20 успешно ловит исключение типа `std::runtime_error`, но обработчик в строке 29 и даже универсальный обработчик в строке 32 не отрабатывает.

Если воспользоваться новой версией компилятора MSVC с флагом `std:c++latest`, программа выдаёт ожидаемый результат:

```
x64 Native Tools-Eingabeaufforderung für VS 2017
C:\Users\rainer>cl.exe /EHsc /W4 /WX /std:c++latest /MD /O2 exceptionExecutionPolicy.cpp
Microsoft (R) C/C++-Optimierungscompiler Version 19.16.27025.1 für x64
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

exceptionExecutionPolicy.cpp
Microsoft (R) Incremental Linker Version 14.16.27025.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:exceptionExecutionPolicy.exe
exceptionExecutionPolicy.obj

C:\Users\rainer>exceptionExecutionPolicy.exe

Without execution policy

C:\Users\rainer>
```

Исключение в алгоритме с явно заданной политикой выполнения

Легко убедиться, что перехватывается только первое исключение.

4.1.3. Опасность гонок данных и мёртвых блокировок

Параллельные алгоритмы не обеспечивают автоматическую защиту от гонок данных и мёртвых блокировок. Рассмотрим пример.

Параллельное выполнение с гонкой данных

```
std::vector<int> v = {1, 2, 3 };
int sum = 0;

std::for_each(std::execution::par, v.begin(), v.end(), [&sum](int x){
    sum += x * x;
});
```

В этом крохотном фрагменте кода имеет место гонка данных. Переменная sum, в которой должна накапливаться сумма квадратов элементов вектора, может модифицироваться параллельно из нескольких потоков. Эту переменную нужно защитить от одновременных модификаций, например так, как показано ниже.

Корректное параллельное выполнение

```
std::vector<int> v = {1, 2, 3 };
int sum = 0;
std::mutex m;

std::for_each(std::execution::par, v.begin(), v.end(), [&sum](int x){
    std::lock_guard<std::mutex> lock(m);
    sum += x * x;
});
```

Если теперь изменить политику выполнения на параллельно-векторизованную (par_unseq), как показано ниже, получится состояние гонки, обычно приводящее к мёртвой блокировке.

Параллельно-векторизованное выполнение с мёртвой блокировкой

```
std::vector<int> v = {1, 2, 3 };
int sum = 0;
std::mutex m;

std::for_each(std::execution::par_unseq, v.begin(), v.end(), [&sum](int x){
    std::lock_guard<std::mutex> lock(m);
    sum += x * x;
});
```

При данной политике выполнения могут произойти два подряд вызова лямбда-функции в одном и том же потоке. Повторная попытка захватить нерекурсивный мьютекс представляет собой неопределённое поведение и в большинстве случаев приводит к мёртвой блокировке. Этого можно избежать, если сделать переменную sum атомарной, как показано ниже.

Корректное параллельно-векторизованное выполнение

```
std::vector<int> v = {1, 2, 3 };
std::atomic<int> sum = 0;

std::for_each(std::execution::par_unseq, v.begin(), v.end(), [&sum](int x){
    sum += x * x;
});
```

Поскольку переменная стала атомарной, можно воспользоваться даже ослабленной семантикой, заменив оператор присваивания следующим:

```
sum.fetch_add(x * x, std::memory_order_relaxed);
```

Политику выполнения можно передавать в качестве параметра в 69 алгоритмов из стандартной библиотеки. Кроме того, стандарт C++ 17 пополнился восемью новыми алгоритмами.

4.2. Алгоритмы стандартной библиотеки

Ниже приведён список из 69 стандартных алгоритмов в пространстве имён std, получивших возможность параллельного выполнения.

```
adjacent_difference    adjacent_find    all_of          any_of
copy                   copy_if          copy_n          count
count_if               equal            fill            fill_n
find                   find_end         find_first_of   find_if
find_if_not            generate         generate_n      includes
inner_product          inplace_merge    is_heap         is_heap_until
```

is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
remove	remove_copy	remove_copy_if	remove_if
replace	replace_copy	replace_copy_if	replace_if
reverse	reverse_copy	rotate	rotate_copy
search	search_n	set_difference	set_intersection
set_symmetric_difference	set_union	sort	stable_partition
stable_sort	swap_ranges	transform	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

Кроме того, в стандарте появились новые алгоритмы.

4.3. Новые параллельные алгоритмы

Новые алгоритмы находятся в пространстве имён `std`. Алгоритмы `std::for_each` и `std::for_each_n` требуют заголовочного файла `<algorithm>`. Оставшимся шести нужен заголовочный файл `<numeric>`. Краткие сведения об этих алгоритмах представлены в следующей таблице.

Алгоритм	Описание
for_each	Применяет унарный вызываемый объект к каждому элементу диапазона
for_each_n	Применяет унарный вызываемый объект к первым n элементам диапазона
exclusive_scan	Применяет вызываемый объект (принимающий два аргумента) к элементам диапазона слева направо и записывает частичный результат каждой итерации (не включающий очередной элемент входного диапазона) в другой диапазон. Если вызываемый объект неассоциативен, результат алгоритма недетерминирован. Этот алгоритм представляет собой параллельный аналог алгоритма std::partial_sum[1]
inclusive_scan	Подобно предыдущему алгоритму, однако текущий элемент входного диапазона включается в соответствующий частичный результат, записываемый в выходной диапазон
transform_exclusive_scan	Подобно алгоритму exclusive_scan, но с применением унарного преобразователя к каждому элементу входного диапазона
transform_inclusive_scan	Подобно алгоритму inclusive_scan, но с применением унарного преобразователя к каждому элементу входного диапазона
reduce	Применяет вызываемый объект (требующий двух аргументов) ко всем элементам диапазона и начальному значению, возвращает итоговый результат. Если вызываемый объект неассоциативен или некоммутативен, результат недетерминирован. Представляет собой параллельный аналог алгоритма std::accumulate[2]
transform_reduce	Применяет унарный вызываемый объект к входному диапазону (или бинарный вызываемый объект к двум входным диапазонам) и выполняет алгоритм reduce для полученных результатов

[1] http://en.cppreference.com/w/cpp/algorithm/partial_sum.

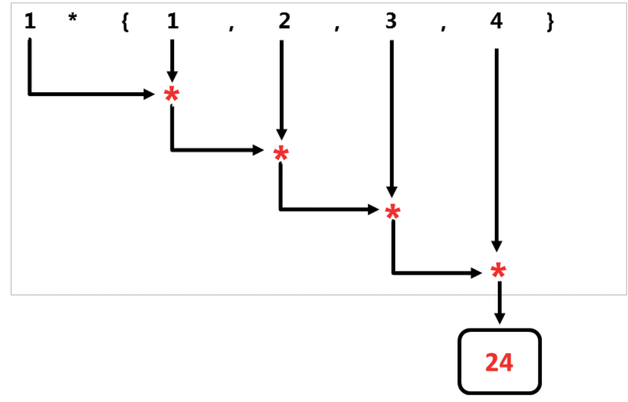
[2] http://en.cppreference.com/w/cpp/algorithm/accumulate.

Это краткое описание может оказаться непростым для понимания, но если читатель уже знаком с алгоритмами `std::accumulate` и `std::partial_sum`, их параллельные вариации `reduce` и `scan` также покажутся знакомыми. Алгоритм `reduce` представляет собой параллельное дополнение к алгоритму `accumulate`, а два алгоритма `scan` – к алгоритму `partial_sum`. Параллельный характер вычислений есть та причина, по которой функции `reduce`, в отличие от функции `accumulate`, требуется ассоциативность и коммутативность вызываемого объекта. Это соображение справедливо и в отношении функций `scan`, которые отличаются от функции `partial_sum` тем, что от вызываемого объекта требуют ассоциативности.

Разберём исчерпывающие примеры применения этих алгоритмов, а затем поговорим о том, как они связаны с парадигмой функционального программирования. Не станем останавливаться лишь на новой функции `std::for_each`. Её единственное отличие от более старой версии, известной начиная со стандарта C++ 98, состоит в том, что она не возвращает никакого значения, тогда как ранее возвращала унарную функцию. Если последовательный алгоритм `std::accumulate` обрабатывает элементы по порядку, слева направо, то алгоритм `std::reduce` делает это в произвольном порядке. Покажем для начала небольшой фрагмент кода, иллюстрирующий работу функций `std::accumulate` и `std::reduce`. Пусть обе применяют лямбда-функцию, перемножающую свои аргументы.

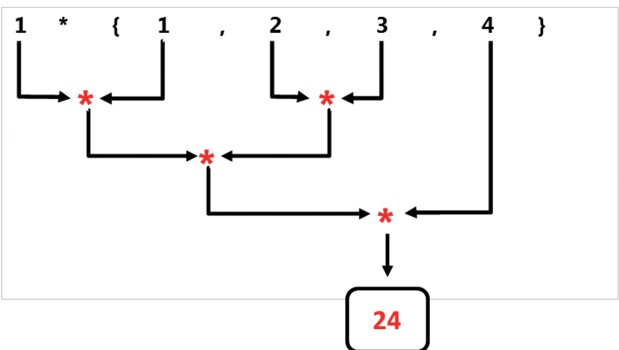
```
std::vector<int> v{1, 2, 3, 4};

std::accumulate(v.begin(), v.end(), 1,
    [](int a, int b){ return a * b; });
std::reduce(std::execution::par, v.begin(), v.end(), 1,
    [](int a, int b){ return a * b; });
```

Следующие две диаграммы дают представление о стратегии вычислений, используемой функциями `std::accumulate` и `std::reduce`. Первая продвигается по контейнеру последовательно, на каждом шагу применяя бинарную операцию к ранее накопленному значению и очередному элементу.



Стратегия алгоритма `std::accumulate`

В противоположность ей вторая функция применяет бинарную операцию в заранее неопределённом порядке.



Стратегия алгоритма std::reduce

Ассоциативность позволяет применять бинарную операцию к любым подряд стоящим парам элементов, а благодаря коммутативности вычислять промежуточные значения можно в любом порядке.

Новые алгоритмы

```
1   // newAlgorithm.cpp
2
3   #include <algorithm>
4   #include <execution>
5   #include <numeric>
6   #include <iostream>
7   #include <string>
8   #include <vector>
9
10  int main(){
11    // for_each_n
12    std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13    std::for_each_n(
14              std::execution::par,
15              intVec.begin(),
16              5,
17              [](int& arg){ arg *= arg; });
18    std::cout << "for_each_n: ";
19    for (auto v: intVec) std::cout << v << " ";
20    std::cout << "\n\n";
21
22    // exclusive_scan and inclusive_scan
23    std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
24    std::exclusive_scan(
25              std::execution::par,
26              resVec.begin(), resVec.end(), resVec.begin(),
27              1,
```

```
28              [](int fir, int sec){ return fir * sec; });
29        std::cout << "exclusive_scan: ";
30      for (auto v: resVec) std::cout << v << " ";
31      std::cout << std::endl;
32
33      std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8, 9};
34      std::inclusive_scan(
35              std::execution::par,
36              resVec2.begin(),
37              resVec2.end(),
38              resVec2.begin(),
39              [](int fir, int sec){ return fir * sec; },
40              1);
41        std::cout << "inclusive_scan: ";
42      for (auto v: resVec2) std::cout << v << " ";
43      std::cout << "\n\n";
44
45      // transform_exclusive_scan and transform_inclusive_scan
46      std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8, 9};
47      std::vector<int> resVec4(resVec3.size());
48      std::transform_exclusive_scan(
49              std::execution::par,
50              resVec3.begin(),
51              resVec3.end(),
52              resVec4.begin(),
53              0,
54              [](int fir, int sec){ return fir + sec; },
55              [](int arg){ return arg * arg; });
56      std::cout << "transform_exclusive_scan: ";
57      for (auto v: resVec4) std::cout << v << " ";
58      std::cout << std::endl;
59
60      std::vector<std::string> strVec{"Only","for","testing","purpose"};
61      std::vector<int> resVec5(strVec.size());
62
63      std::transform_inclusive_scan(
64              std::execution::par,
65              strVec.begin(),
66              strVec.end(),
67              resVec5.begin(),
68              0,
69              [](auto fir, auto sec){ return fir + sec; },
70              [](auto s){ return s.length(); });
71      std::cout << "transform_inclusive_scan: ";
72      for (auto v: resVec5) std::cout << v << " ";
73      std::cout << "\n\n";
74
75      // reduce and transform_reduce
76      std::vector<std::string> strVec2{"Only","for","testing","purpose"};
77
78      std::string res = std::reduce(
79              std::execution::par,
80              strVec2.begin() + 1,
```

```
81              strVec2.end(),
82              strVec2[0],
83              [](auto fir, auto sec){ return fir + ":" + sec; });
84     std::cout << "reduce: " << res << std::endl;
85
86     std::size_t res7 = std::transform_reduce(
87              std::execution::par,
88              strVec2.begin(),
89              strVec2.end(),
90              [](std::string s){ return s.length(); },
91              0,
92              [](std::size_t a, std::size_t b){ return a + b; });
93     std::cout << "transform_reduce: " << res7 << std::endl;
94 }
```

Новые параллельные алгоритмы применены здесь к вектору целых чисел, объявленному в строке 12, и к вектору строк, объявленному в строке 60.

Алгоритм `std::for_each_n` в строке 13 заменяет первые пять элементов вектора их квадратами.

Вызовы функций `std::exclusive_scan` в строке 24 и `std::inclusive_scan` в строке 34 работают похожим образом. Обе применяют двухместную операцию к элементам диапазона. Различие состоит в том, что первая выдаёт промежуточное значение до того, как применить операцию к текущему элементу входного контейнера.

Вызов функции `std::transform_exclusive_scan` в строке 48 довольно сложен для понимания и требует некоторых пояснений. К каждому элементу вектора `resVec3` сначала применяется лямбда-функция, возводящая его в квадрат. Затем к полученным значениям применяется операция сложения с нулём в качестве начального значения. Получаемые промежуточные результаты записываются в другой вектор `resVec4`.

Функция `std::transform_inclusive_scan` в строке 63 работает сходным образом. Для каждой строки из входного контейнера вычисляется длина, затем полученные числа складываются, бегущие суммы помещаются в другой контейнер.

Работу функции `std::reduce` понять просто: она вставляет двоеточие между каждой парой соседних элементов входного вектора строк. Чтобы строка-результат не начиналась с двоеточия, диапазон, к которому применяется алгоритм `std::reduce`, начинается со второго элемента (`strVec2.begin() + 1`), а в качестве начального значения берётся первый элемент `strVec2[0]`.

Превращение алгоритма `transform_reduce` в `map_reduce`

Функция `transform_reduce`, вызываемая в строке 86, заслуживает особого внимания. Заметим, во-первых, что алгоритм, названный `transform` в стандартной библиотеке языка C++, во многих других языках называется `map`. Поэтому функцию `transform_reduce` можно было бы также назвать `map_reduce`. Теперь читатель наверняка заметит, что функция `transform_reduce` есть не что иное, как реализация знаменитого параллельного алгоритма MapReduce[1] на языке C++. В самом деле, эта функция применя-

[1] https://en.wikipedia.org/wiki/MapReduce.

ет унарную функцию (в данном примере – функцию, возвращающую длину строки) к диапазону входных данных и затем сворачивает совокупность полученных значений в единственное значение-результат (в нашем примере – сумму длин).

Представленный на рисунке результат работы программы поможет понять работу этих алгоритмов.

```
File Edit View Bookmarks Settings Help
rainer@suse:~> newAlgorithm

for_each_n: 1 4 9 16 25 6 7 8 9 10

exclusive_scan: 1 1 2 6 24 120 720 5040 40320
inclusive_scan: 1 2 6 24 120 720 5040 40320 362880

transform_exclusive_scan: 0 1 5 14 30 55 91 140 204
transform_inclusive_scan: 4 7 14 21

reduce: Only:for:testing:purpose
transform_reduce: 21

rainer@suse:~>
rainer : bash
```

Новые алгоритмы

4.3.1. Новые перегрузки

Все функции из семейств `reduce` и `scan` в стандартной библиотеке языка C++ обладают несколькими перегруженными вариантами. В простейшей форме их можно вызывать без бинарной операции и начального значения. Если не передавать бинарную операцию, по умолчанию будет использовано сложение. Если же не задавать начальное значение, будет, в зависимости от алгоритма, использоваться такое значение:

- в алгоритмах `std::inclusive_scan` и `std::transform_inclusive_scan` в качестве начального значения берётся первый элемент диапазона;
- в алгоритмах `std::reduce` and `std::transform_reduce` берётся значение по умолчанию типа элемента входного диапазона:

```
typename std::iterator_traits<InputIt>::value_type{}
```

Посмотрим теперь на эти новые алгоритмы с точки зрения функционального программирования.

4.3.2. Наследие функционального программирования

Если говорить кратко, то новые функции из стандарта языка C++ обладают прямыми аналогами в чистом функциональном языке Haskell:

- функция `for_each_n` языка C++ соответствует в языке Haskell функции `map`;
- функции `exclusive_scan` и `inclusive_scan` соответствуют функциям `scanl` и `scanl1`;
- функции `transform_exclusive_scan` и `transform_inclusive_scan` подобны композиции функции `map` с функциями `scanl` и `scanl1`;
- функция `reduce` делает то же, что в языке Haskell функции `foldl` и `foldl1`;
- функция `transform_reduce` соответствует композиции функции `map` с функциями `foldl` и `foldl1`.

Прежде чем продемонстрировать язык Haskell в действии, стоит сказать несколько слов об упомянутых здесь функциях:

- функция `map` применяет функцию к каждому элементу списка и строит список результатов;
- функции `foldl` и `foldl1` сворачивают список, применяя двухместную операцию ко всем его элементам и возвращая итоговое значение, причём функция `foldl` принимает начальное значение в качестве аргумента, а функция `foldl` требует, чтобы список был непустым – роль начального значения играет его первый элемент;
- функции `scanl` и `scanl1` применяют двухместную операцию к элементам списка, как функции `foldl` и `foldl1`, но возвращают список промежуточных результатов;
- функции `foldl`, `foldl1`, `scanl`, `scanl1` обрабатывают списки слева направо, т. е. от начала к концу.

Посмотрим на функции языка Haskell в действии. На следующем рисунке показано окно интерпретатора.

```
File  Edit  View  Bookmarks  Settings  Help
Prelude>
Prelude> let ints = [1..9]                                        (1)
Prelude>
Prelude> let strings= ["Only","for","testing","purpose"]          (2)
Prelude>
Prelude> map (\a -> a * a) ints                                   (3)
[1,4,9,16,25,36,49,64,81]
Prelude>
Prelude> scanl (*) 1 ints                                         (4)
[1,1,2,6,24,120,720,5040,40320,362880]
Prelude>
Prelude> scanl (+) 0 ints                                         (5)
[0,1,3,6,10,15,21,28,36,45]
Prelude>
Prelude> scanl (+) 0 . map(\a -> a * a) $ ints                   (6)
[0,1,5,14,30,55,91,140,204,285]
Prelude>
Prelude> scanl1 (+) . map(\a -> length a) $ strings              (7)
[4,7,14,21]
Prelude>
Prelude> foldl1 (\l r -> l ++ ":" ++ r) strings                   (8)
"Only:for:testing:purpose"
Prelude>
Prelude> foldl (+) 0 . map (\a -> length a) $ strings            (9)
21
Prelude>
rainer : ghc
```

Алгоритмы обработки списков в языке Haskell

В строках, помеченных цифрами (1) и (2), определены список целых чисел и список строк. В строке (3) к списку чисел применяется функция, возводящая свой аргумент в квадрат. Строки (4) и (5) несколько сложнее. Выражение (4) строит список произведений всех чисел с начала списка (1) до текущего элемента, начиная с числа 1 как нейтрального элемента умножения. Выражение 5 делает то же самое для операции сложения. Выражения (6), (7) и (9) бросают вызов тем, кто привык к императивному стилю. Читать их лучше справа налево. Выражение (7) представляет собой композицию функций. Точкой обозначается операция, которая результат одной функции (правого операнда) подаёт на вход другой (левого операнда). Сначала к списку-аргументу применяется функция `map`, которая каждой строке из входного списка ставит в соответствие её длину. Затем функция `scanl1` формирует список бегущих сумм. Выражение (9) похоже на (7) с тем отличием, что функция `foldl` вырабатывает лишь одно итоговое значение и требует начального значения – в данном случае это число 0. Выражение (8) должно быть понятным: оно соединяет между собой строки из входного списка, вставляя между ними двоеточие.

4.4. Поддержка в различных компиляторах

Насколько известно автору, в настоящее время нет полной и отвечающей стандарту реализации параллельных алгоритмов стандартной библиотеки[1]. В компиляторах Microsoft Visual Compiler и GCC доступны по крайней мере последовательная и параллельная политики выполнения, но не параллельно-векторизованная. Если попытаться вызвать алгоритм, передав ему параметр `std::execution::par_unseq`, реализация библиотек подставит вместо неё параллельную политику `std::execution::par`.

4.4.1. Компилятор Microsoft Visual Compiler

В версии 2017 update 8 (номер версии 15.8) в библиотеке поддерживается около тридцати параллельных алгоритмов:

```
adjacent_difference    adjacent_find       all_of
any_of                 count               count_if
equal                  exclusive_scan      find
find_end               find_first_of       find_if
```

[1] Это утверждение было справедливо на момент написания книги. Ситуация становится значительно лучше: полная реализация параллельных алгоритмов имеется в библиотеке GCC libstdc++ с версии 9, Intel Parallel STL с версии 18.0 и MSVC STL с версии 15.7, а библиотека Clang libc++ версии 11.0 поддерживает политики выполнения, включая параллельно-векторизованную. – *Прим. перев.*

```
for_each              for_each_n                  inclusive_scan
mismatch              none_of                     reduce
remove                remove_if                   search
search_n              sort                        stable_sort
transform             transform_exclusive_scan    transform_inclusive_scan
transform_reduce
```

4.4.2. Компилятор GCC

Благодаря библиотеке Threading Building Blocks[1] (TBB) фирмы Intel стало возможно использовать параллельные стандартные алгоритмы также и с компилятором GCC версии 9. TBB – это библиотека шаблонов, разработанная для параллельного программирования на многоядерных процессорах. Параллельные стандартные алгоритмы появились в библиотеке TBB начиная с версии 2018.

Использовать библиотеку TBB просто. Нужно лишь подключить её для связывания, указав ключ `-tbb`.

Использование библиотеки Threading Building Blocks

4.4.3. Будущие реализации параллельных стандартных алгоритмов

Своё первое знакомство с параллельными стандартными алгоритмами автор получил, воспользовавшись каркасом HPX[2]. Каркас HPX (High-Performance ParallelX) представляет собой набор средств общего назначения для создания параллельных и распределённых приложений любого масштаба. В этом каркасе имеется, в собственном пространстве имён, также и реализация параллельных алгоритмов из стандарта языка C++.

Ради полноты изложения стоит упомянуть и другие частичные реализации параллельных стандартных алгоритмов:

- Intel[3];
- Thibaut Lutz[4];

---

[1] https://en.wikipedia.org/wiki/Threading_Building_Blocks.
[2] http://stellar.cct.lsu.edu/projects/hpx/.
[3] https://software.intel.com/en-us/get-started-with-pstl.
[4] https://github.com/t-lutz/ParallelSTL.

- Nvidia (thrust)[1];
- Codeplay[2].

4.5. Вопросы производительности

Следующая программа вычисляет тангенсы очень большого контейнера случайных чисел с применением последовательной, параллельной и параллельно-векторизованной политики.

Производительность при различных политиках выполнения

```
1  // parallelSTLPerformance.cpp
2
3  #include <algorithm>
4  #include <cmath>
5  #include <chrono>
6  #include <execution>
7  #include <iostream>
8  #include <random>
9  #include <string>
10 #include <vector>
11
12 constexpr long long size = 500’000’000;
13
14 const double pi = std::acos(-1);
15
16 template <typename Func>
17 void getExecutionTime(const std::string& title, Func func){
18     const auto start = std::chrono::steady_clock::now();
19     func();
20     const std::chrono::duration<double> dur =
21         std::chrono::steady_clock::now() - start;
22     std::cout << title << “: “ << dur.count() << “ sec. “ << std::endl;
23 }
24
25 int main(){
26     std::cout << std::endl;
27
28     std::vector<double> randValues;
29     randValues.reserve(size);
30
31     std::mt19937 engine;
32     std::uniform_real_distribution<> uniformDist(0, pi / 2);
33     for (long long i = 0 ; i < size ; ++i)
34         randValues.push_back(uniformDist(engine));
35
```

1 https://thrust.github.io/doc/group__execution__policies.html.
2 https://github.com/KhronosGroup/SyclParallelSTL.

```
36      std::vector<double> workVec(randValues);
37
38      getExecutionTime(
39          "std::execution::seq",
40          [workVec]() mutable {
41              std::transform(
42                  std::execution::seq,
43                  workVec.begin(),
44                  workVec.end(),
45                  workVec.begin(),
46                  [](double arg){ return std::tan(arg); });
47      });
48
49      getExecutionTime(
50          "std::execution::par",
51          [workVec]() mutable {
52              std::transform(
53                  std::execution::par,
54                  workVec.begin(),
55                  workVec.end(),
56                  workVec.begin(),
57                  [](double arg){ return std::tan(arg); });
58      });
59
60      getExecutionTime(
61          "std::execution::par_unseq",
62          [workVec]() mutable {
63              std::transform(
64                  std::execution::par_unseq,
65                  workVec.begin(),
66                  workVec.end(),
67                  workVec.begin(),
68                  [](double arg){ return std::tan(arg); });
69      });
70  }
```

Одно и то же вычисление выполняется с последовательной политикой в строке 38, параллельной в строке 49 и параллельно-векторизованной в строке 60. Сначала в вектор `randValues` записываются 500 миллионов случайных чисел из полуоткрытого интервала [0, pi/2). Функция-шаблон `getExecutionTime`, объявленная в строках 16–23, получает название политики выполнения и лямбда-функцию, выполняет эту функцию и выводит на печать продолжительность её выполнения. Одна важная особенность трёх лямбда-функций, объявленных в строках 40, 53 и 62, состоит в том, что они объявлены с ключевым словом `mutable`. Это нужно для того, чтобы функции могли модифицировать свою копию контейнера `workVec`. По умолчанию лямбда-функции превращаются компилятором в неизменяемые объекты. Если лямбда-функции нужно менять значение переменных, полученных через замыкание по значению, её нужно объявлять с ключевым словом `mutable`.

О сравнении компиляторов

Следует особо подчеркнуть, что в задачи этого раздела не входит сравнение компиляторов Microsoft Visual Compiler и GCC. Быстродействие программ измерялось на различных машинах с отличающимися характеристиками. Показанные ниже данные дают лишь самое приблизительное представление. Если читателя интересует выигрыш производительности на его конкретной системе, следует запустить программу на ней. Цель описанного здесь эксперимента – узнать, приносит ли выгоду параллельное выполнение алгоритмов стандартной библиотеки, и насколько большую. Нас здесь интересует относительная производительность последовательных и параллельных алгоритмов.

При компиляции программы под ОС Windows и Linux использовался максимальный уровень оптимизации. Это означает, что компилятор запускался с параметром /O2 на ОС Windows и -O3 под ОС Linux.

4.5.1. Компилятор Microsoft Visual Compiler

Ноутбук автора обладает восемью логическими ядрами, но параллельное выполнение дало прирост быстродействия более, чем в десять раз.

```
x64 Native Tools-Eingabeaufforderung für VS 2017
C:\Users\rainer>cl.exe /EHsc /W4 /WX /std:c++latest /MD /O2 parallelSTLPerformance.cpp
Microsoft (R) C/C++-Optimierungscompiler Version 19.16.27025.1 für x64
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

parallelSTLPerformance.cpp
Microsoft (R) Incremental Linker Version 14.16.27025.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:parallelSTLPerformance.exe
parallelSTLPerformance.obj

C:\Users\rainer>parallelSTLPerformance.exe

std::execution::seq: 5.44017 sec.
std::execution::par: 0.455092 sec.
std::execution::par_unseq: 0.458994 sec.

C:\Users\rainer>
```

Производительность политик выполнения
при использовании компилятора Microsoft Visual Compiler

Числа для параллельного и параллельно-векторизованного выполнения приблизительно совпадают. В блоге команды разработчиков среды Visual C++ говорится: «Следует иметь в виду, что в системе Visual C++ параллельная и параллельно-векторизованная политики реализованы одинаково, поэтому не стоит ожидать повышения производительности от использования политики par_unseq с этой реализацией, однако когда-нибудь могут появиться и иные реализации, дающие компилятору большую свободу действий».

4.5.2. Компилятор GCC

Имеющийся у автора компьютер под управлением ОС Linux обладает лишь четырьмя ядрами. Результаты измерений показаны ниже.

```
rainer@seminar:~> parallelSTLPerformance

std::execution::seq: 12.3034 sec.
std::execution::par: 3.21129 sec.
std::execution::par_unseq: 3.20413 sec.

rainer@seminar:~>
```

Производительность политик выполнения
при использовании компилятора GCC

Эти показатели вполне ожидаемы. На четырёх ядрах параллельное выполнение происходит примерно вчетверо быстрее, чем последовательное. Время работы параллельного и параллельно-векторизованного алгоритма приблизительно одинаково. Отсюда можно сделать предположение, что в компиляторе GCC использована та же стратегия, что и в компиляторе фирмы Microsoft для ОС Windows. А именно, когда программист просит параллельную векторизованную политику выполнения, указав параметр `std::execution::par_unseq`, он получает параллельную политику `std::execution::unseq`. Такое поведение вполне отвечает стандарту C++ 17, поскольку политики выполнения носят рекомендательный, а не обязательный характер.

4.6. Краткие итоги

Начиная со стандарта C++ 17 для большинства алгоритмов обработки контейнеров в стандартной библиотеке доступны параллельные версии. Алгоритмы можно вызывать с так называемой политикой выполнения. Политикой выполнения определяется, может ли алгоритм выполняться последовательно, параллельно или параллельно с использованием векторных инструкций процессора.

Помимо 69 старых алгоритмов, получивших возможность параллельного или параллельно-векторизованного выполнения, библиотека пополнилась новыми алгоритмами, предназначенными для поэлементного преобразования, свёртки и формирования бегущих сумм.

5. Сопрограммы в стандарте C++ 20

Сопрограммы (англ. *coroutines*) – это функции, которые могут приостанавливать своё выполнение, сохраняя внутреннее состояние. Эволюция функций в языке C++ продвинулась ещё на шаг.

C++ 11	C++ 14	C++ 17	C++ 20	C++ 23
2011	2014	2017	220	2023
• Модель памяти • Атомарные переменные • Потоки • Двоичные семафоры (мьютексы) и блокировщики • Локальные данные потока • Переменные условия • Задания	• Блокировщики чтения-записи	• Поддержка параллельных вычислений в стандартных алгоритмах	• Атомарные умные указатели • Потоки с ожиданием `std::jthread` • Защёлки и барьеры • Семафоры общего вида • Сопрограммы (coroutine)	• Исполнители • Расширения класса `std::future` • Транзакционная память • Блоки заданий • Векторы с параллельной обработкой

Сопрограммы

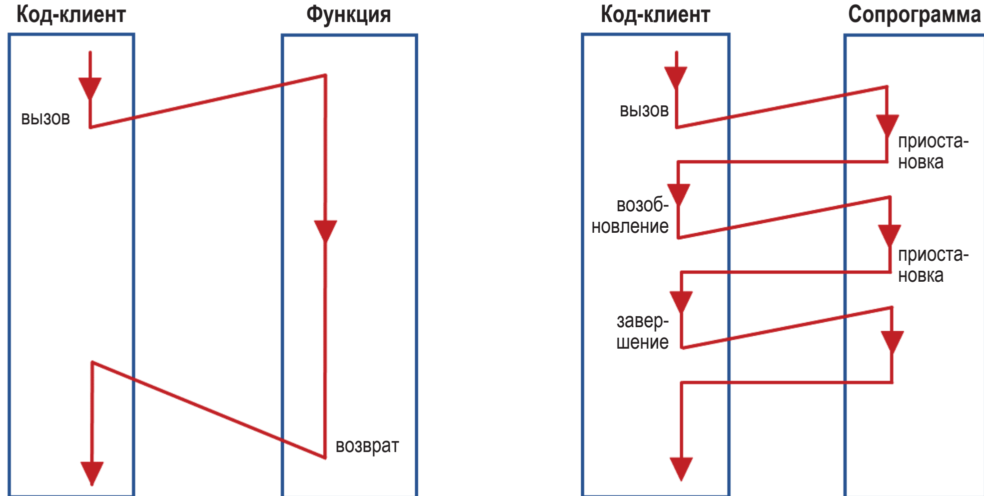
Трудность понимания сопрограмм

Понять сопрограммы оказалось поначалу непросто даже для автора этой книги. Читателю рекомендуется при первом знакомстве с данной главой пропустить разделы о концептуальной модели и процессе функционирования и сосредоточиться на примерах из разделов «Вариации на тему фьючерса», «Модификации и обобщения генератора» и «Варианты жизненного цикла задания». Внимательное изучение примеров и самостоятельные эксперименты с ними должны выработать у читателя начальное понимание, с которым можно погружаться в подробности и тонкости работы с сопрограммами.

То, о чём в этом разделе говорится как о новшестве стандарта C++ 20, на самом деле представляет собой довольно старую идею. Впервые термин «сопрограмма» ввёл Мелвин Конуэй[1]. Он использовал данное понятие в своей публикации 1963 года, посвящённой построению компиляторов. Дональд

[1] https://en.wikipedia.org/wiki/Melvin_Conway.

Кнут[1] назвал процедуры частным случаем сопрограмм. Иногда проходит немало времени, прежде чем идея получает признание.



Сопрограммы по сравнению с функциями

Если функцию можно лишь вызвать и, по завершении её работы, получить результат, то сопрограмму можно вызвать, получить промежуточный результат, обработать его, пока работа сопрограммы приостановлена, затем продолжить или прервать её выполнение.

В стандарте C++ 20 появились два новых ключевых слова: `co_await` и `co_yield`, а понятие выполнения функции соответственно расширено.

Ключевое слово `co_await` позволяет приостанавливать и возобновлять вычисление выражения. Если конструкцию вида `co_await expression` использовать в функции `func`, операция auto `getResult = func()` не блокирует выполнение вызывающего кода, если результат функции ещё не готов. Вместо блокирования, требующего значительных ресурсов, здесь имеет место легковесное ожидание.

Ключевое слово `co_yield` позволяет создавать функции-генераторы. Такая функция возвращает очередной элемент последовательности каждый раз, когда её вызывают. Функцию-генератор можно рассматривать как своего рода поток, из которого можно получать значения одно за другим. Этот поток может быть бесконечным. Таким образом, на основе сопрограмм можно реализовать на языке C++ парадигму ленивых вычислений.

[1] https://en.wikipedia.org/wiki/Donald_Knuth.

5.1. Функция-генератор

Следующая программа предельно проста. Функция `getNumbers` возвращает по порядку целые числа, начиная со значения `begin`, пока число не превысит значение `end`, с шагом `inc`. Начальное значение `begin` должно быть меньше, чем конечное значение `end`, а шаг `inc` должен быть положительным.

Жадная функция-генератор

```
1  // greedyGenerator.cpp
2  
3  #include <iostream>
4  #include <vector>
5  
6  std::vector<int> getNumbers(int begin, int end, int inc = 1) {
7      std::vector<int> numbers;
8      for (int i = begin; i < end; i += inc) {
9          numbers.push_back(i);
10     }
11 
12     return numbers;
13 }
14 
15 int main() {
16 
17     std::cout << '\n';
18 
19     const auto numbers= getNumbers(-10, 11);
20 
21     for (auto n: numbers) std::cout << n << " ";
22 
23     std::cout << "\n\n";
24 
25     for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";
26 
27     std::cout << "\n\n";
28 }
```

Конечно, писать такую функцию своими руками – значит изобретать велосипед, так как эту задачу решает библиотечная функция `std::iota`[1]. Для полноты изложения на следующем рисунке показан результат работы программы.

[1] http://en.cppreference.com/w/cpp/algorithm/iota.

```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
rainer@suse:~> greedyGenerator
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
rainer@suse:~>
rainer : bash
```

Функция-генератор в действии

Для нашей цели существенны две особенности этой программы. С одной стороны, в вектор `numbers`, объявленный в строке 7, всегда помещаются все числа из заданного диапазона – даже если вызывающий контекст обработает, скажем, лишь первые пять из тысячи элементов. С другой стороны, функцию `getNumbers` довольно легко преобразовать в ленивый генератор. Следующая программа намеренно оставлена незавершённой: в ней опущено определение шаблона `generator`.

Ленивая функция-генератор

```
1  // lazyGenerator.cpp
2
3  #include <iostream>
4
5  generator<int> generatorForNumbers(int begin, int inc = 1) {
6      while (true) {
7          co_yield i;
8      }
9  }
10
11 int main() {
12     std::cout << '\n';
13
14     const auto numbers = generatorForNumbers(-10);
15
16     for (int i= 1; i <= 20; ++i) std::cout << numbers() << " ";
17
18     std::cout << "\n\n";
19
20     for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";
21
22     std::cout << "\n\n";
23 }
```

Сопрограмма `generatorForNumbers` в этой новой программе возвращает генератор. Объект `numbers`, объявленный в строке 14, или вызов функции-генератора в строке 20 при каждом обращении возвращает очередное число. Так, цикл по диапазону (англ. *range-based*) запрашивает у сопрограммы результат. Сопрограмма при каждом вызове немедленно возвращает значение числа `i` (счётчика цикла) посредством оператора `co_yield` и приостанавливает своё

выполнение. При следующем обращении к сопрограмме она продолжит своё выполнение с этого же места.

Выражение в строке 20 демонстрирует, что вызов сопрограммы можно объединить с использованием её результата.

Нужно особо подчеркнуть одно обстоятельство. Сопрограмма `generatorForNumbers` генерирует бесконечный поток чисел, поскольку цикл в строке 6 никогда не завершается. Это нормально, если пользовательский код запрашивает конечное число значений, как в строке 16. Однако ситуация меняется в строке 20, поскольку цикл на пользовательской стороне должен получить из потока *все* данные. В этом случае цикл будет выполняться вечно.

5.2. Особенности сопрограмм

Сопрограммы обладают рядом уникальных особенностей.

5.2.1. Типичные сценарии использования

Сопрограммы часто применяются для создания приложений, управляемых событиями[1], к которым относятся системы имитационного моделирования, игры, серверы, интерфейсы пользователя или даже некоторые алгоритмы обработки больших потоков данных. Сопрограммы также часто используют для реализации кооперативной многозадачности[2]. Суть её состоит в том, что множество задач могут выполняться на одном процессоре или ядре, но их выполнение чередуется таким образом, чтобы создавать иллюзию параллельной работы; при этом каждая из множества совместно выполняемых задач берёт себе столько процессорного времени, сколько сама сочтёт нужным, однако обязуется за конечное время отдать управление, тем самым позволив другим задачам продолжить свою работу. Этим кооперативная многозадачность отличается от вытесняющей, при которой планировщик решает, на какой промежуток времени предоставить процессор той или иной задаче, и принудительно забирает у неё доступ к процессору по истечении этого времени.

Существует несколько разновидностей сопрограмм, которые следует разобрать отдельно.

5.2.2. Разновидности сопрограмм

Среди сопрограмм, как они определены в стандарте C++ 20, можно выделить частные случаи: по способу передачи управления – симметричные и асимметричные, по способу представления в языке – «первого класса» и некото-

[1] https://ru.wikipedia.org/wiki/Событийно-ориентированное_программирование.

[2] https://ru.wikipedia.org/wiki/Многозадачность#Совместная_или_кооперативная_многозадачность.

рые ограниченные языковые конструкции, по способу локализации внутреннего состояния – обладающие и не обладающие стеком.

Обращение к асимметричной сопрограмме возвращает управление вызывающему коду, тогда как симметричная сопрограмма может делегировать выполнение другой сопрограмме.

Сопрограммы первого класса выглядят подобно функциям и ведут себя подобно данным. Последнее означает, что их можно использовать в качестве аргументов или возвращаемых значений функций, а также сохранять в переменных.

Если сопрограмма не пользуется стеком, её выполнение можно приостанавливать и возобновлять. При этом управление и выработанное значение передаются вызвавшему коду. Такие сопрограммы должны хранить своё состояние для последующего возобновления за пределами стека. Сопрограммы без стека часто называют возобновляемыми.

5.2.3. Требования к сопрограммам

Гор Нишанов в документе N4402[1] так описывает требования, которыми руководствовались при разработке механизма сопрограмм:

- высокая масштабируемость (возможность одновременной работы миллиардов сопрограмм);
- высокая эффективность приостановок и возобновлений по сравнению с накладными расходами на вызов функции;
- удобное взаимодействие с прочими элементами языка без дополнительных усилий;
- открытость программного интерфейса, позволяющая разрабатывать библиотеки сопрограмм, предоставляющие разнообразные высокоуровневые средства, такие как генераторы, «горутины»[2] (англ. *goroutines*), задания и многое другое;
- возможность использования в средах, где механизм исключений запрещён или недоступен.

Требованиям масштабируемости и лёгкой интеграции с существующими средствами языка лучше всего удовлетворяют сопрограммы без стека. Для сопрограммы со стеком выделяется по умолчанию 1 МБ памяти на ОС Windows и 2 МБ на ОС Linux.

Обычную функцию можно преобразовать в сопрограмму четырьмя способами.

5.2.4. Преобразование функции в сопрограмму

Функция становится сопрограммой, если в ней используется хотя бы одна из следующих конструкций:

---

1 https://isocpp.org/files/papers/N4402.pdf.

2 https://tour.golang.org/concurrency/1.

- ключевое слово `co_return`;
- ключевое слово `co_await`;
- ключевое слово `co_yield`;
- выражение `co_await` в цикле по диапазону.

Различие между фабрикой сопрограмм и объектом-сопрограммой

Термин «сопрограмма» часто используется в двух различных смыслах: с одной стороны, его применяют к функциям, в которых используются ключевые слова `co_return`, `co_await` и `co_yield`, а с другой – к объекту-сопрограмме. Использование одного и того же слова для обозначения двух различных явлений может сбить читателя с толку (как в своё время сбило автора этой книги). Следует прояснить оба термина. Рассмотрим код.

Простая сопрограмма, возвращающая значение 2001

```
MyFuture<int> createFuture() {
   co_return 2021;
}

int main() {
   auto fut = createFuture();
   std::cout << "fut.get(): " << fut.get() << '\n';
}
```

В этом примере фигурируют функция `createFuture` и возвращаемый ею объект типа `MyFuture<int>`. Оба называют сопрограммами. Говоря более точно, функция `createFuture` – это фабрика сопрограмм, которая возвращает объект-сопрограмму. Объект-сопрограмма инкапсулирует приостанавливаемое и возобновляемое вычисление, реализует определённую концептуальную модель и обладает определённым в этой модели поведением. Ниже, в разделе, посвящённом оператору `co_return`, будет подробно рассказано о внутреннем устройстве и использовании этой элементарной сопрограммы.

5.2.4.1. Ограничения

Сопрограммы не могут содержать оператор `return`. Тип возвращаемого значения сопрограммы не может быть оставлен на волю компилятора. Это касается как неограниченного вывода типа (ключевое слово `auto`), так и ограниченного посредством концептов (англ. *concept*).

Кроме того, в сопрограммы не могут быть преобразованы функции с произвольным числом аргументов[1], объявленные со спецификаторами `constexpr` и `consteval`, конструкторы, деструкторы, а также главная функция программы.

5.3. Концептуальная модель

Инфраструктуру для создания сопрограмм составляют два десятка функций, некоторые из которых программисту нужно определить самостоятельно,

---

1 https://en.cppreference.com/w/cpp/language/variadic_arguments.

а некоторые можно переопределить при желании, чтобы настроить свои сопрограммы под конкретные требования.

Для работы сопрограммы нужны три составные части: объект-обещание, дескриптор сопрограммы и кадр. Клиент получает дескриптор сопрограммы, чтобы через него взаимодействовать с объектом-обещанием, который хранит своё текущее состояние в кадре.

5.3.1. Объект-обещание

Работа с объектом-обещанием происходит из кода сопрограммы. Через него клиенту становятся видны результаты работы сопрограммы или исключения. Объект-обещание должен поддерживать следующий интерфейс.

Интерфейс объекта-обещания

Функция-член	Описание
Конструктор по умолчанию	Должно поддерживаться создание объектов по умолчанию
`initial_suspend()`	Определяет, находится ли сопрограмма в приостановленном состоянии сразу после запуска
`final_suspend noexcept()`	Определяет, находится ли сопрограмма в приостановленном состоянии сразу перед завершением
`unhandled_exception()`	Вызывается при возникновении исключения
`get_return_object()`	Возвращает возобновляемый объект, связанный с данным обещанием
`return_value(val)`	Вызывается оператором `co_return val`
`return_void()`	Вызывается оператором `co_return`
`yield_value(val)`	Вызывается оператором `co_yield val`

Компилятор автоматически вставляет вызовы этих функций, когда создаёт исполняемый код сопрограммы. В разделе о процессе функционирования сопрограмм подробно рассказано о том, при каких условиях и в какой последовательности вызываются эти функции.

Функция-член `get_return_object` возвращает объект, которым клиент может пользоваться для взаимодействия с данным обещанием и, следовательно, с сопрограммой, в том числе возобновлять вычисление. Объект-обещание должен обладать по меньшей мере одной из функций-членов `return_value`, `return_void` или `yield_value`. При этом нет нужды определять функции `return_value` и `return_void`, если сопрограмма никогда не завершается.

Функции `yield_value`, `initial_suspend` и `final_suspend` возвращают контроллер ожидания (англ. *awaitable*). С помощью этих объектов можно управлять приостановкой сопрограммы.

5.3.2. Дескриптор сопрограммы

Дескриптор сопрограммы – это промежуточный объект, через который клиент может возобновлять выполнение сопрограммы или полностью прекращать её выполнение. Дескриптор сопрограммы является частью возобнов-

ляемой функции. Проиллюстрируем это примером. В следующем фрагменте кода показан тип генератора, обладающий своим дескриптором.

Работа с дескриптором сопрограммы

```
1  template<typename T>
2  struct Generator {
3      struct promise_type;
4      using handle_type = std::coroutine_handle<promise_type>;
5      handle_type coro;
6
7      Generator(handle_type h): coro(h) {}
8
9      ~Generator() {
10         if (coro) coro.destroy();
11     }
12
13     T getValue() {
14         return coro.promise().current_value;
15     }
16
17     bool next() {
18         coro.resume();
19         return !coro.done();
20     }
21 ...
22 }
```

Конструктор, объявленный в строке 7, получает от вызывающей стороны дескриптор, связанный с объектом-обещанием[1]. Функции-члены `next` (строка 17) и `getValue` (строка 13) позволяют клиенту возобновить работу обещания или получить его текущее значение, пользуясь для этого дескриптором сопрограммы. Рассмотрим теперь применение этих объектов.

Вызов сопрограммы

```
Generator<int> coroutineFactory(); // returns a coroutine object

auto gen = coroutineFactory();
gen.next();
auto result = gen.getValue();
```

Реализация всех функций шаблона `Generator` делегирует работу дескриптору. Так, через дескриптор можно:

- продолжить выполнение сопрограммы;
- получить из сопрограммы очередной результат вычислений;
- узнать, не завершалась ли сопрограмма;
- завершить сопрограмму и удалить связанные с ней данные из памяти.

Отметим, что состояние сопрограммы автоматически разрушается, когда заканчивается выполнение тела её функции. Обращение к переменной `coro`

[1] https://en.cppreference.com/w/cpp/coroutine/coroutine_handle.

как к логическому значению (строка 10) даёт результат `true` только тогда, когда выполнение тела сопрограммы окончено.

Возобновляемые объекты должны обладать вложенным типом обещания

Определяя тип данных, который будет представлять возобновляемые вычисления, программист должен объявить в нём тип-член `promise_type`. Возможная альтернатива состоит в том, чтобы создать специализацию шаблона `std::coroutine_traits`[1] для типа `Generator` и определить в ней общедоступный тип-член `promise_type`.

5.3.3. Кадр сопрограммы

Кадр – это внутреннее состояние сопрограммы, хранящееся обычно в куче. Он включает в себя объект-обещание, о котором уже шла речь, копию параметров, с которыми сопрограмма вызвана, данные о точках приостановки, значения локальных переменных – как тех, чьё время жизни заканчивается до текущей точки приостановки, так и тех, которые продолжают существовать после неё.

Для того чтобы оптимизировать размещение сопрограммы в памяти, нужно соблюдать два обязательных условия:

1) время жизни сопрограммы должно полностью находиться внутри времени жизни вызывающего контекста;
2) вызывающий контекст знает размер кадра сопрограммы.

Ключевыми для механизма сопрограмм являются понятия прообраза ожидания (англ. *awaitable*) и контроллера ожидания (awaiter).

5.4. Ожидание отложенного вычисления

Функции-члены `yield_value`, `initial_suspend` и `final_suspend` возвращают прообразы ожидания.

5.4.1. Прообраз ожидания

Прообраз ожидания (англ. *awaitable*) – это объект, на основе которого создаётся контроллер ожидания (awaiter), а от последнего уже зависит, будет ли сопрограмма делать паузу в своём выполнении. Компилятор автоматически генерирует вызовы функций-членов, которые возвращают прообразы ожидания, в случаях, показанных в следующей таблице.

Генерируемые компилятором вызовы

Ситуация	Генерируемый вызов
Начало выполнения	`co_await prom.initial_suspend()`
`co_yield value`	`co_await prom.yield_value (value)`

[1] https://en.cppreference.com/w/cpp/coroutine/coroutine_traits.

Ситуация	Генерируемый вызов
co_return value	co_await prom.yield_value(value)
Окончание выполнения	co_await prom.final_suspend()

Аргументом оператора co_await должен быть прообраз ожидания. Оператор co_await преобразовывает их в контроллеры ожидания.

5.4.2. Общие требования к контроллерам ожидания

Чтобы считаться контроллером ожидания, объект должен поддерживать три функции-члена:

Интерфейс контроллера ожидания

Функция	Описание
await_ready	Оповещает о том, что результат вычислений готов
await_suspend	Вызывается при приостановке сопрограммы и управляет последующим возобновлением её работы
await_resume	Вызывается при возобновлении работы сопрограммы и устанавливает результат оператора co_await

В стандарте C++ 20 определены два простейших контроллера ожидания: std::suspend_always и std::suspend_never.

5.4.3. Стандартные контроллеры ожидания

Как и явствует из названия, контроллер std::suspend_always всегда приостанавливает сопрограмму. Для этого его функция await_ready всегда возвращает значение false.

Контроллер ожидания std::suspend_always

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

Контроллер ожидания std::suspend_never ведёт себя прямо противоположным образом. С ним сопрограмма никогда не приостанавливается, потому что функция await_ready всегда возвращает значение true.

Контроллер ожидания std::suspend_never

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
```

```
    constexpr void await_resume() const noexcept {}
};
```

Контроллеры ожидания играют роль строительных блоков при создании таких функций, как `initial_suspend` и `final_suspend`. Эти функции вызываются автоматически на соответствующих этапах жизни сопрограммы: в начале и в конце её выполнения.

5.4.4. Функция `initial_suspend`

Если функция `initial_suspend`, член объекта-обещания, возвращает объект `std::suspend_always`, сопрограмма приостанавливается сразу после запуска. Если же она возвращает объект `std::suspend_never`, сопрограмма не делает остановки. Отсюда получаем две следующие реализации.

Ленивая сопрограмма

```
std::suspend_always initial_suspend() {
    return {};
}
Жадная сопрограмма
std::suspend_never initial_suspend() {
    return {};
}
```

5.4.5. Функция `final_suspend`

Если функция `final_suspend`, член объекта-обещания, возвращает объект `std::suspend_always`, сопрограмма приостанавливается непосредственно перед завершением. Если же она возвращает объект `std::suspend_never`, сопрограмма не делает остановки. Отсюда получаем две следующие реализации.

Ленивая сопрограмма

```
std::suspend_always final_suspend() noexcept {
    return {};
}
```

Жадная сопрограмма

```
std::suspend_never final_suspend() noexcept {
    return {};
}
```

Таким образом, теперь в наших руках есть прообразы ожидания и контроллеры ожидания. Нужно ещё понять, как из прообраза получается контроллер.

5.4.6. Получение контроллера ожидания

Есть два способа из имеющегося прообраза ожидания получить контроллер ожидания:

- с помощью перегруженного оператора co_await, который в качестве аргумента принимает прообраз ожидания и возвращает контроллер ожидания;
- если подходящей перегрузки этого оператора нет, контроллером ожидания становится сам прообраз ожидания.

Вспомним, что в выражениях вида co_await expression выражение expression является прообразом ожидания. Это выражение обычно представляет собой вызов функции-члена у объекта-обещания: prom.yield_value(value), prom.initial_suspend() или prom.final_suspend(). Обозначим через awatiable этот объект-прообраз. Компилятору предстоит каким-то способом из прообраза построить контроллер ожидания, чтобы затем вызвать его функции-члены await_ready, await_suspend и await_resume.

Для этого компилятор выполняет поиск подходящего преобразования в таком порядке:

1. Сначала он пытается найти перегрузку оператора co_await среди функций-членов объекта awatiable:

```
awaiter = awaitable.operator co_await();
```

2. Также компилятор пытается найти перегрузку оператора co_await среди свободно стоящих функций (т. е. не членов класса):

```
awaiter = operator co_await(awaitable);
```

3. Если это не удаётся, объект awatiable становится контроллером ожидания:

```
awaiter = awaitable;
```

Прообраз и есть контроллер ожидания

Изучая приводимые далее примеры, читатель наверняка заметит, что в них почти всегда имеет место последний случай: прообраз ожидания неявно сам становится контроллером ожидания. Лишь в примере, посвящённом синхронизации потоков, используется перегруженный оператор co_await, чтобы в явном виде получить контроллер ожидания.

Теперь, когда статический аспект сопрограмм вполне изучен, пришла пора разобраться с динамическими аспектами.

5.5. Процесс функционирования сопрограммы

Компилятор преобразовывает сопрограмму и создаёт два механизма: механизм управления объектом-обещанием, находящимся снаружи от кода сопрограммы, и механизм управления ожиданием, работающий внутри сопрограммы.

5.5.1. Управление обещанием

Если в теле функции используются ключевые слова `co_yield`, `co_await` или `co_return`, эта функция становится сопрограммой, и компилятор преобразовывает её тело в нечто наподобие следующего кода.

Преобразованная сопрограмма

```
1  {
2      Promise prom;
3      co_await prom.initial_suspend();
4      try {
5          <тело функции с операторами co_return, co_yield, or co_await>
6      }
7      catch (...) {
8          prom.unhandled_exception();
9      }
10     FinalSuspend:
11         co_await prom.final_suspend();
12 }
```

Таким образом, преобразованный код работает с объектом-обещанием. Ниже перечислены основные этапы процедуры управления обещанием.

- Начинается выполнение сопрограммы:
  - выделяется память для кадра;
  - значения параметров функции копируются в кадр;
  - создаётся объект-обещание `prom` (строка 2);
  - совершается вызов `prom.get_return_object()`, чтобы получить дескриптор сопрограммы и сохранить его в локальной переменной. Результат этого вызова возвращается вызывающей стороне, когда сопрограмма в первый раз приостанавливает выполнение;
  - вызывается функция `prom.initial_suspend()`, её результат используется для ожидания с помощью оператора `co_await`, как показано в строке 3. Напомним, что функция `initial_suspend` объекта-обещания может возвращать объекты `suspend_never` и `suspend_always` для жадной и ленивой стратегий вычислений соответственно;
  - когда начальное ожидание закончено, выполняется тело сопрограммы.
- Выполнение сопрограммы достигает точки приостанова:
  - значение `prom.get_return_object()` возвращается функции, которая пробудила сопрограмму.

- Выполнение сопрограммы достигает оператора `co_return`:
  - вызывается функция `prom.return_void()`, если оператор употреблён без аргумента или с выражением типа `void`;
  - вызывается функция `prom.return_value()`, если оператор употреблён с выражением-аргументом типа, отличного от `void`;
  - уничтожаются все переменные, созданные в стеке;
  - вызывается функция `prom.final_suspend()`, затем к полученному результату применяется оператор `co_await`.
- Выполнение сопрограммы завершается (например, после оператора `co_return`, вследствие необработанного исключения, или по запросу через дескриптор сопрограммы):
  - вызывается деструктор объекта-обещания;
  - вызываются деструкторы объектов-аргументов функции;
  - освобождается память, выделенная для кадра сопрограммы;
  - управление возвращается вызвавшему контексту.

Если сопрограмма завершается из-за необработанного исключения, дополнительно выполняются следующие действия:

- исключение перехватывается, затем из обработчика вызывается функция `prom.unhandled_exception()`;
- вызывается функция `prom.final_suspend()`, затем к полученному результату применяется оператор `co_await`.

Если оператор `co_await` в явном виде используется в коде сопрограммы или если компилятор неявно подставляет этот оператор вместе с вызовом функций `initial_suspend`, `final_suspend` или `yield_value`, запускается ещё один механизм – управление ожиданием.

5.5.2. Управление ожиданием

Наличие в сопрограмме оператора `co_await` заставляет компилятор преобразовать его, добавив вызовы функций `await_ready`, `await_suspend` и `await_resume` контроллера ожидания `awaiter`. Основное назначение преобразованного кода можно охарактеризовать как управление ожиданием.

В первом приближении результат работы компилятора можно представить следующим псевдокодом.

Генерируемый код управления ожиданием

```
1  если awaiter.await_ready() возвращает false:
2
3      приостановить сопрограмму
4
5      если awaiter.await_suspend(coroutineHandle) возвращает тип:
6
7          void:
8              awaiter.await_suspend(coroutineHandle);
9              сопрограмма остаётся приостановленной
10             вернуть управление вызвавшему коду
```

```
11
12         bool:
13             bool result = awaiter.await_suspend(coroutineHandle);
14             если result имеет значение “истина”:
15                 сопрограмма остаётся приостановленной
16                 вернуть управление вызвавшему коду
17             иначе:
18                 goto возобновление
19
20         дескриптор другой сопрограммы:
21             auto anotherHandle = awaiter.await_suspend(coroutineHandle);
22             anotherHandle.resume();
23             вернуть управление вызвавшему коду
24
25 возобновление:
26
27 return awaiter.await_resume();
```

Все эти действия выполняются только в случае, если функция `awaiter.await_ready()` возвращает значение `false` (строка 1). Если же этот вызов возвращает значение `true`, это означает, что сопрограмма уже вычислила свой результат – тогда остаётся лишь вызвать функцию `awaiter.await_resume` (строка 27).

Предположим, что выражение `awaiter.await_ready()` даёт значение `false`. В первую очередь сопрограмма приостанавливается (строка 3), вслед за этим сразу вызывается функция `awaiter.await_suspend`. Её возвращаемым типом может быть тип `void` (строка 7), логический (строка 12) или дескриптор сопрограммы (строка 20). Механизма управления ожиданием работает различным образом в каждом из этих трёх случаев.

Управление ожиданием в зависимости от типа функции await_suspend

Тип	Описание
`void`	Сопрограмма приостанавливается и отдаёт управление вызвавшему коду
`bool`	При значении `true`: сопрограмма приостанавливается и отдаёт управление вызвавшему коду. При значении `false`: сопрограмма продолжает работу и не отдаёт управление
`std::coroutine_handle`	Другая сопрограмма пробуждается и продолжает выполнение. Текущая сопрограмма отдаёт управление

Что должно произойти при выбросе исключения? Это зависит от того, где возникает исключение: в функции `await_ready`, `await_suspend` или `await_resume`:

- функция `await_ready`: сопрограмма не приостанавливается, функции `await_suspend` и `await_resume` не вызываются;
- функция `await_suspend`: исключение перехватывается, сопрограмма продолжает свою работу, функция `await_resume` не вызывается;
- функция `await_resume`: вызовы функций `await_ready` и `await_suspend` уже отработали и вернули свои значения, но вызов функции `await_resume`, конечно, значения не возвращает.

Теперь пора показать, как эта теория работает на практике.

5.6. Оператор co_return и жадный фьючерс

Здесь речь пойдёт о случае, когда для возврата значения из сопрограммы используется оператор co_return. Вероятно, следующая программа даёт простейший из возможных примеров сопрограмм, которые всё ещё делают что-то осмысленное: эта сопрограмма автоматически сохраняет значение в переменную.

Жадный фьючерс

```
1  // eagerFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      std::shared_ptr<T> value;
10     MyFuture(std::shared_ptr<T> p): value(p) {}
11     ~MyFuture() { }
12     T get() {
13         return *value;
14     }
15
16     struct promise_type {
17         std::shared_ptr<T> ptr = std::make_shared<T>();
18         ~promise_type() { }
19         MyFuture<T> get_return_object() {
20             return ptr;
21         }
22         void return_value(T v) {
23             *ptr = v;
24         }
25         std::suspend_never initial_suspend() {
26             return {};
27         }
28         std::suspend_never final_suspend() noexcept {
29             return {};
30         }
31         void unhandled_exception() {
32             std::exit(1);
33         }
34     };
35 };
```

```
36
37 MyFuture<int> createFuture() {
38     co_return 2021;
39 }
40
41 int main() {
42     std::cout << '\n';
43
44     auto fut = createFuture();
45     std::cout << "fut.get(): " << fut.get() << '\n';
46
47     std::cout << '\n';
48 }
```

Объект класса `MyFuture` ведёт себя как фьючерс[1], который начинает работу немедленно после создания. Вызов сопрограммы `createFuture` в строке 44 возвращает фьючерс, а выражение `fut.get()` в строке 45 получает результат из соответствующего обещания.

Поведение этой сопрограммы отличается от поведения обычных фьючерсов в одном аспекте: возвращаемое значение сопрограммы `createFuture` доступно сразу после вызова. Для удобства управлением временем жизни объект, с которым работает сопрограмма, находится под управлением умного указателя (строки 9 и 17). Сопрограмма возвращает объекты типа `std::suspend_never` из обеих функций, запрашивающих приостановку (строки 25 и 28), и, следовательно, не приостанавливается ни в начале, и в конце выполнения. Это означает, что сопрограмма отрабатывает при вызове функции `createFuture`. Функция-член `get_return_object` в строке 19 создаёт объект-фьючерс, который отныне связан с сопрограммой, а функция-член `return_value` в строке 22 сохраняет в переменную результат работы сопрограммы, который поступает из оператора `co_return 2021` в строке 38. Клиентский код вызывает функцию `fut.get` в строке 45 и использует фьючерс, чтобы получить доступ к данным, хранящимся в объекте-обещании. Функция-член `get` просто возвращает клиенту значение, хранящееся по умному указателю (строка 13). Таким образом, программа печатает следующий результат:

```
fut.get(): 2021
```

Работа жадного фьючерса

Читатель мог бы возразить, что создание сопрограммы, которая работает в точности как обычная функция, не стоит затраченных усилий. И это правда! Однако эта простейшая сопрограмма представляет хорошую отправную точку, чтобы впоследствии создавать разнообразные более сложные фьючерсы. В разделе 6.5 будут разобраны и другие примеры фьючерсов.

[1] https://en.cppreference.com/w/cpp/thread/future.

5.7. Оператор co_yield и бесконечный поток данных

Благодаря оператору co_yield становится возможным реализовать генератор бесконечного потока данных, из которого клиент может получить одно за другим сколь угодно много значений. Тип возвращаемого значения функции generatorForNumbers (страница 5) указан как generator<int>. Объект этого типа должен содержать в себе объект-обещание p, причём оператор co_yield i превращается компилятором в выражение co_await p.yield_value(i). Оператор co_yield i может быть выполнен сколь угодно много раз, и каждый раз выполнение сопрограммы должно приостанавливаться.

Следующая программа генерирует бесконечный поток данных. Сопрограмма getNext посредством оператора co_yield выдаёт числа, начиная со значения start, увеличивая каждое следующее значение на величину step.

Бесконечный поток данных

```
1  // infiniteDataStream.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9      struct promise_type;
10     using handle_type = std::coroutine_handle<promise_type>;
11
12     Generator(handle_type h): coro(h) {}
13     handle_type coro;
14     std::shared_ptr<T> value;
15
16     ~Generator() {
17         if ( coro ) coro.destroy();
18     }
19     Generator(const Generator&) = delete;
20     Generator& operator = (const Generator&) = delete;
21     Generator(Generator&& oth): coro(oth.coro) {
22         oth.coro = nullptr;
23     }
24     Generator& operator = (Generator&& oth) {
25         coro = oth.coro;
26         oth.coro = nullptr;
27         return *this;
28     }
29     int getValue() {
30         return coro.promise().current_value;
31     }
32     bool next() {
```

```
33          coro.resume();
34          return not coro.done();
35      }
36      struct promise_type {
37          promise_type()  = default;
38
39          ~promise_type() = default;
40
41          auto initial_suspend() {
42              return std::suspend_always{};
43          }
44          auto final_suspend() noexcept {
45              return std::suspend_always{};
46          }
47          auto get_return_object() {
48              return Generator{handle_type::from_promise(*this)};
49          }
50          auto return_void() {
51              return std::suspend_never{};
52          }
53
54          auto yield_value(int value) {
55              current_value = value;
56              return std::suspend_always{};
57          }
58          void unhandled_exception() {
59              std::exit(1);
60          }
61          T current_value;
62      };
63  };
64
65  Generator<int> getNext(int start = 0, int step = 1) {
66      auto value = start;
67      while (true) {
68          co_yield value;
69          value += step;
70      }
71  }
72
73  int main() {
74      std::cout << '\n';
75
76      std::cout << "getNext():";
77      auto gen = getNext();
78      for (int i = 0; i <= 10; ++i) {
79          gen.next();
80          std::cout << " " << gen.getValue();
81      }
82
83      std::cout << "\n\n";
84
85      std::cout << "getNext(100, -10):";
```

```
86      auto gen2 = getNext(100, -10);
87      for (int i = 0; i <= 20; ++i) {
88          gen2.next();
89          std::cout << “ “ << gen2.getValue();
90      }
91
92      std::cout << ‘\n’;
93  }
```

В функции `main` создаются две сопрограммы. Первая из них, названная `gen` (строка 77), возвращает значения от 0 и далее по возрастанию, а программа выбирает из них первые 11 элементов: до числа 10 включительно. Вторая, названная `gen2` (строка 86), выдаёт значения, начиная со 100 и с шагом −10, программа из них выбирает 21 элемент, т. е. значения от 100 до −100. Прежде чем погрузиться в разбор того, как работает эта программа, посмотрим на результат её работы, полученный благодаря интерактивной системе `Wandbox`[1].

```
Start

getNext(): 0 1 2 3 4 5 6 7 8 9 10

getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100

0

Finish
```

Бесконечные потоки данных

Функционирование программы начинается с такой последовательности шагов.

1. Создаётся объект-обещание.
2. Вызывается функция `promise.get_return_object()`, её результат сохраняется в переменной.
3. Создаётся генератор `gen` или `gen2`.
4. Вызывается функция `promise.initial_suspend()`. Генератор работает по ленивой стратегии, поэтому всегда начинает свою работу с приостановки. Управление при этом передаётся клиентскому коду.
5. Клиентский код запрашивает у генератора следующее значение. Управление передаётся сопрограмме.
6. Сопрограмма выполняет оператор `co_yield`. Он вырабатывает значение для возврата, выполнение сопрограммы приостанавливается, а управление передаётся клиентскому коду.
7. Клиентский код обрабатывает полученное от генератора значение.

На последующих итерациях цикла выполняются только шаги 5–7. В разделе 6.6 будет показано дальнейшее развитие и ряд усовершенствований этой программы.

---

[1] https://wandbox.org/.

5.8. Оператор co_await

Оператор `co_await` управляет приостановкой и последующим возобновлением работы сопрограммы. В операторе `co_await exp` выражение `exp` должно быть так называемым прообразом ожидания (англ. *awaitable*). Он должен поддерживать преобразование в контроллер ожидания (англ. *awaiter*), который, в свою очередь, должен реализовывать определённый интерфейс, состоящий из трёх функций: `await_ready`, `await_suspend` и `await_resume`. Типичный пример ситуации, когда стоит использовать оператор `co_await`, даёт сервер, ожидающий входящих запросов и отвечающий на них. Традиционная реализация могла бы выглядеть так:

Блокирующий сервер

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = acceptor.accept();      // блокирует
4      auto request = socket.read();           // блокирует
5      auto response = handleRequest(request);
6      socket.write(response);                 // блокирует
7  }
```

Этот сервер очень прост, так как он всего лишь отвечает на все запросы последовательно в одном и том же потоке. Сервер слушает порт 443, принимает соединение (строка 3), читает данные, присланные клиентом (строка 4), и посылает клиенту ответ (строка 6). Вызовы в строках 3, 4 и 6 блокируют поток.

Благодаря оператору `co_await` эти блокирующие вызовы можно заменить сопрограммами с приостановкой и возобновлением работы.

Ожидающий сервер

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = co_await acceptor.accept();
4      auto request = co_await socket.read();
5      auto response = handleRequest(request);
6      co_await socket.write(response);
7  }
```

Перед тем как представить усложнённый пример синхронизации потоков с использованием сопрограмм, начнём с чего-то более простого: с запуска задания по запросу.

5.8.1. Запуск задания по запросу

Следующая сопрограмма проста настолько, насколько это вообще возможно. Она приостанавливается и ожидает возможности продолжить выполнение, причём в роли контроллера ожидания выступает объект предопределённого типа `std::suspend_never()`.

Запуск задания по запросу

```
1  // startJob.cpp
2  
3  #include <coroutine>
4  #include <iostream>
5  
6  struct Job {
7      struct promise_type;
8      using handle_type = std::coroutine_handle<promise_type>;
9      handle_type coro;
10 
11     Job(handle_type h): coro(h){}
12 
13     ~Job() {
14         if ( coro ) coro.destroy();
15     }
16 
17     void start() {
18         coro.resume();
19     }
20 
21     struct promise_type {
22         auto get_return_object() {
23             return Job{handle_type::from_promise(*this)};
24         }
25 
26         std::suspend_always initial_suspend() {
27             std::cout << "    Preparing job" << '\n';
28             return {};
29         }
30 
31         std::suspend_always final_suspend() noexcept {
32             std::cout << "    Performing job" << '\n';
33             return {};
34         }
35 
36         void return_void() {}
37         void unhandled_exception() {}
38     };
39 };
40 
41 Job prepareJob() {
42     co_await std::suspend_never();
43 }
44 
45 int main() {
46     std::cout <<  "Before job" << '\n';
47 
48     auto job = prepareJob();
49     job.start();
50 
51     std::cout <<  "After job" <<  '\n';
52 }
```

Читатель может подумать, что сопрограмма prepareJob, объявленная в строке 41, совершенно бесполезна, потому что она якобы пытается ожидать чего-то, но контроллер ожидания велит ей не приостанавливать выполнения. Однако она вовсе не бесполезна! Функция prepareJob – это по меньшей мере фабрика сопрограмм, которая выполняет оператор co_yield (строка 42) и возвращает объект-сопрограмму. Вызов функции prepareJob в строке 48 приводит к созданию объекта-сопрограммы типа Job. Если же теперь присмотреться к реализации класса Job, легко убедиться, что объект-сопрограмма сразу после создания приостанавливает своё выполнение, поскольку функция-член initial_suspend объекта-обещания возвращает контроллер ожидания std::suspend_always (строка 26). Именно поэтому для фактического запуска сопрограммы нужен вызов в строке 49 функции-члена job.start (см. строку 18). Функция-член final_suspend, объявленная в строке 31, также возвращает контроллер ожидания std::suspend_always. Ниже представлен результат запуска программы.

```
Before job
    Preparing job
    Performing job
After job
```

Запуск задания по запросу

В разделе 6.7 эта программа будет взята за отправную точку для дальнейших экспериментов.

5.9. Синхронизация потоков

Для потоков вполне обычное дело – синхронизироваться между собой. Один поток подготавливает порцию данных, которые нужны другому потоку. Для организации взаимодействия между отправителем и получателем можно использовать переменные условия, обещания и фьючерсы, а также атомарные переменные логического типа. Однако сопрограммы позволяют сделать синхронизацию потоков ещё более простой, свободной от присущего переменным условия риска потерянного и ложного пробуждения.

Синхронизация потоков

```
1   // senderReceiver.cpp
2
3   #include <coroutine>
4   #include <chrono>
5   #include <iostream>
6   #include <functional>
7   #include <string>
```

```
8   #include <stdexcept>
9   #include <atomic>
10  #include <thread>
11
12  class Event {
13   public:
14     Event() = default;
15
16     Event(const Event&) = delete;
17     Event(Event&&) = delete;
18     Event& operator=(const Event&) = delete;
19     Event& operator=(Event&&) = delete;
20
21     class Awaiter;
22     Awaiter operator co_await() const noexcept;
23
24     void notify() noexcept;
25
26   private:
27     friend class Awaiter;
28
29     mutable std::atomic<void*> suspendedWaiter{nullptr};
30     mutable std::atomic<bool> notified{false};
31  };
32
33  class Event::Awaiter {
34   public:
35     Awaiter(const Event& eve): event(eve) {}
36
37     bool await_ready() const;
38     bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
39     void await_resume() noexcept {}
40
41   private:
42     friend class Event;
43
44     const Event& event;
45     std::coroutine_handle<> coroutineHandle;
46  };
47
48  bool Event::Awaiter::await_ready() const {
49     // allow at most one waiter
50     if (event.suspendedWaiter.load() != nullptr) {
51         throw std::runtime_error("More than one waiter is not valid");
52     }
53
54     // false - приостановка сопрограммы
55     // true  - сопрограмма выполняется как обычная функция
56     return event.notified;
57  }
58
59  bool Event::Awaiter::await_suspend(std::coroutine_handle<> corHandle)
60  noexcept {
```

```
61      coroutineHandle = corHandle;
62
63      if (event.notified) return false;
64
65      // store the waiter for later notification
66      event.suspendedWaiter.store(this);
67
68      return true;
69  }
70
71  void Event::notify() noexcept {
72      notified = true;
73
74      // try to load the waiter
75      auto* waiter = static_cast<Awaiter*>(suspendedWaiter.load());
76
77      // check if a waiter is available
78      if (waiter != nullptr) {
79          // resume the coroutine => await_resume
80          waiter->coroutineHandle.resume();
81      }
82  }
83
84  Event::Awaiter Event::operator co_await() const noexcept {
85      return Awaiter{ *this };
86  }
87
88  struct Task {
89      struct promise_type {
90          Task get_return_object() { return {}; }
91          std::suspend_never initial_suspend() { return {}; }
92          std::suspend_never final_suspend() noexcept { return {}; }
93          void return_void() {}
94          void unhandled_exception() {}
95      };
96  };
97
98  Task receiver(Event& event) {
99      auto start = std::chrono::high_resolution_clock::now();
100     co_await event;
101     std::cout << "Got the notification! " << '\n';
102     auto end = std::chrono::high_resolution_clock::now();
103     std::chrono::duration<double> elapsed = end - start;
104     std::cout << "Waited " << elapsed.count() << " seconds." << '\n';
105 }
106
107 using namespace std::chrono_literals;
108
109 int main() {
110     std::cout << '\n';
111
112     std::cout << "Notification before waiting" << '\n';
113     Event event1{};
```

```
114     // оповестить
115     auto senderThread1 = std::thread([&event1]{ event1.notify(); });
116     auto receiverThread1 = std::thread(receiver, std::ref(event1));
117 
118     receiverThread1.join();
119     senderThread1.join();
120 
121     std::cout << '\n';
122 
123     std::cout << "Notification after 2 seconds waiting" << '\n';
124     Event event2{};
125     auto receiverThread2 = std::thread(receiver, std::ref(event2));
126     auto senderThread2 = std::thread([&event2]{
127         std::this_thread::sleep_for(2s);
128         event2.notify();        // оповестить
129     });
130 
131     receiverThread2.join();
132     senderThread2.join();
133 
134     std::cout << '\n';
135 }
```

С точки зрения использования, синхронизация потоков посредством сопрограмм устроена вполне очевидно. Рассмотрим этот код в подробностях. Потоки-отправители `senderThread1` (строка 115) и `senderThread2` (строка 125) используют события (объекты класса `Event`) в строках 115 и 128, чтобы известить потоки-получатели. Функция `receiver`, объявленная в строках 98–105, представляет собой сопрограмму, которая выполняется в потоках `receiverThread1` (строка 116) и `receiverThread2` (строка 126). Программа измеряет длительность выполнения сопрограммы и выводит на консоль. Это и есть длительность ожидания события сопрограммой. Результат работы программы представлен на следующем рисунке.

```
Start


Notification before waiting
Got the notification!
Waited 1.5738e-05 seconds.

Notification after 2 seconds waiting
Got the notification!
Waited 2.00019 seconds.


0

Finish
```

Синхронизация потоков

Если сопоставить класс Generator из примера с бесконечной последовательностью данных и класс Event из последнего примера, можно обнаружить небольшое различие. В первом случае класс Generator играет роли как прообраза ожидания, так и контроллера ожидания; во втором же случае класс Event обладает перегруженной операцией co_await, которая возвращает контроллер ожидания. Разделение ответственностей на прообраз и контроллер ожидания делает код более структурированным.

Результат работы программы свидетельствует, что выполнение второй сопрограммы занимает около двух секунд. Причина состоит в том, что событие event1 оповещает ожидающий поток немедленно (см. строку 115), тогда как событие event2 сначала выдерживает двухсекундную паузу (строка 128).

Теперь посмотрим на этот пример с позиции разработчика. Жизненный цикл сопрограммы не так просто понять. У класса Event есть два интересных члена: suspendedWaiter и notified. Переменная suspendedWaiter, объявленная в строке 29, идентифицирует контроллер ожидания, которому адресовано оповещение, а переменная notified, объявленная в строке 30, позволяет различить, состоялось ли оповещение.

При разборе принципа работы оповещений будем предполагать, что в первом случае событие происходит до того, как сопрограмма начинает его ожидать, а во втором случае – наоборот.

Рассмотрим судьбу первого события. Объект event1 успевает послать оповещение до того, как поток-получатель receiverThread1 начинает выполнение. В строке 115 вызывается функция notify, объявленная в строках 71– 82. В первую очередь устанавливается флаг, означающий, что оповещение уже произошло. Сразу после этого из атомарной переменной-члена извлекается адрес контроллера ожидания, которому предназначено оповещение. В данном случае это будет значение nullptr, поскольку поток-получатель ещё не начал выполнение. Следовательно, вызов функции resume в строке 80 выполнен не будет. Вслед за этим выполняется функция await_ready, объявленная в строках 48–57. Она начинает с проверки, не подписано ли на событие более одного получателя, – в этом случае функция выбрасывает исключение. Важнейшее для понимания этой функции обстоятельство состоит в том, что её возвращаемое значение есть значение переменной-члена event.notification, а оно уже установлено в true функцией notify. Когда функция await_ready возвращает значение true, это приводит к тому, что сопрограмма не приостанавливается, а продолжает выполняться, как обычная функция.

Во втором случае оператор co_await event2 выполняется до того, как объект event2 посылает своё оповещение. Выполнение оператора co_await начинается с вызова функции await_ready (строка 48 и далее). Важное отличие от предыдущего случая состоит в том, что теперь она возвращает значение false. Это приводит к приостановке сопрограммы. Для этого вызывается функция await_suspend (строки 59–69). В качестве аргумента она получает дескриптор сопрограммы и сохраняет его для дальнейшего использования в переменной-члене coroutineHandle (см. строку 61). Под дальнейшим использованием, конечно, следует понимать возобновление работы сопрограммы. Затем в строке 66 указатель на контроллер ожидания сохраняется в переменную-член suspendedWaiter. Когда в дальнейшем происходит оповещение о событии

(строка 128), вызывается функция `notify` (строки 71– 82). Однако на этот раз переменная-член `waiter` имеет значение, отличное от `nullptr`. Вследствие этого будет выполнен вызов в строке 80, который возобновит выполнение сопрограммы.

5.10. Краткие итоги

- Сопрограммы представляют собой обобщение понятия функции. Они умеют приостанавливать и возобновлять своё выполнение, удерживая в промежутках своё состояние.
- Со стандартом C++ 20 программист получает в своё распоряжение не конкретные сопрограммы, а общий каркас для реализации сопрограмм. Этот каркас состоит из двух десятков функций, из которых одни нужно, а другие можно определить своими руками.
- Благодаря появлению новых ключевых слов `co_await` и `co_yield` жизненный цикл выполнения функций расширяется двумя новыми возможностями.
- Оператор `co_await` позволяет приостанавливать и возобновлять вычисление. Так, если оператор вида `co_await expr` используется в теле функции `func`, вызов вида `auto result = func()` не блокирует выполнение кода, если результат работы функции ещё не готов. Вместо жадной до ресурсов блокировки происходит ресурсосберегающее ожидание.
- Оператор `co_yield` позволяет создавать бесконечные потоки данных.

6. Учебные примеры

Теперь, когда изучен теоретический материал о моделях памяти, интерфейсе управления потоками и о новейшем стандарте C++ 20, можно применить его на практике и численно измерить производительность этих средств.

Об интерпретации показателей производительности

Численные показатели производительности следует воспринимать с известной осторожностью. Здесь не ставится цель получить точные значения для каждого алгоритма на обеих операционных системах: Windows и Linux. Важнее составить представление о том, какие алгоритмы работают хорошо, а какие – нет. Вместо цели сравнить между собой числовые показатели, полученные автором на компьютерах с ОС Windows и Linux, ставится цель узнать, какой из алгоритмов лучше работает в каждой из этих ОС.

6.1. Вычисление суммы элементов вектора

Какой способ сложения элементов контейнера `std::vector` самый быстрый? Чтобы получить ответ, заполним контейнер `std::vector` миллионом случайных, равномерно распределённых чисел от 1 до 10. Задача будет состоять в том, чтобы вычислить суммы этих чисел множеством различных способов. Производительность однопоточного суммирования берётся в качестве основы для сравнения. В последующих разделах будут разобраны алгоритмы на основе атомарных переменных, блокировщиков, локальных данных потока и заданий. Начнём с простейшего однопоточного алгоритма.

6.1.1. Суммирование элементов вектора в одном потоке

Даже однопоточный алгоритм можно реализовать несколькими способами.

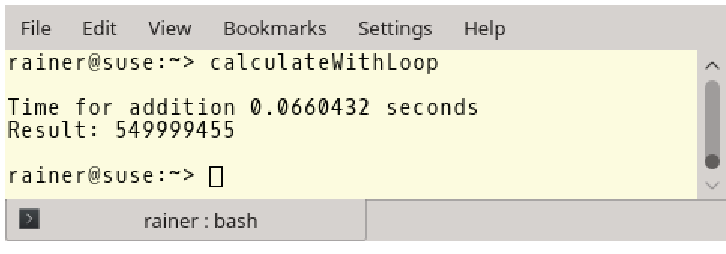
6.1.1.1. Суммирование в цикле по диапазону

Самый очевидный способ суммирования состоит в том, чтобы прибавлять очередной элемент к накопителю в цикле по диапазону. Само суммирование происходит в строке 26.

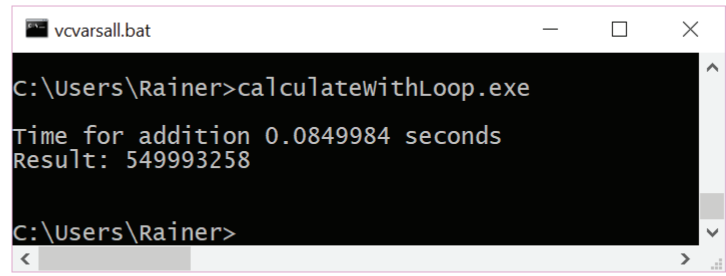
Суммирование в цикле по диапазону

```
1  // calculateWithLoop.cpp
2  
3  #include <chrono>
4  #include <iostream>
5  #include <random>
6  #include <vector>
7  
8  constexpr long long size = 100000000;
9  
10 int main(){
11   std::cout << std::endl;
12 
13   std::vector<int> randValues;
14   randValues.reserve(size);
15 
16   // random values
17   std::random_device seed;
18   std::mt19937 engine(seed());
19   std::uniform_int_distribution<> uniformDist(1, 10);
20   for (long long i = 0 ; i < size ; ++i)
21        randValues.push_back(uniformDist(engine));
22 
23   const auto sta = std::chrono::steady_clock::now();
24 
25   unsigned long long sum = {};
26   for (auto n: randValues) sum += n;
27 
28   const std::chrono::duration<double> dur =
29         std::chrono::steady_clock::now() - sta;
30 
31   std::cout << "Time for addition " << dur.count()
32             << " seconds" << std::endl;
33   std::cout << "Result: " << sum << std::endl;
34 
35   std::cout << std::endl;
36 }
```

Сколь быстро работает эта программа? На компьютерах автора получились результаты, представленные на следующих двух рисунках.



Суммирование в цикле по диапазону в системе Linux



Суммирование в цикле по диапазону в системе Windows

Однако не стоит использовать циклы в явном виде. В большинстве случаев лучше пользоваться алгоритмами из стандартной библиотеки.

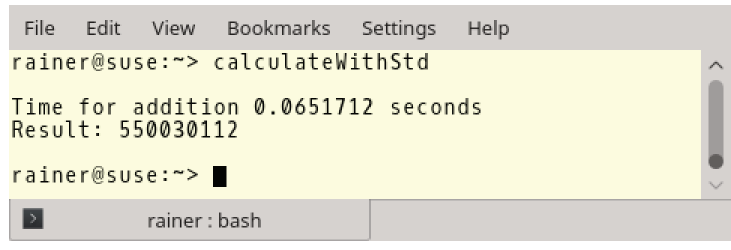
6.1.1.2. Суммирование алгоритмом std::accumulate

Алгоритм std::accumulate представляет собой наиболее правильное средство для суммирования элементов вектора. Для экономии места ниже показан лишь вызов функции, весь остальной код идентичен предыдущему.

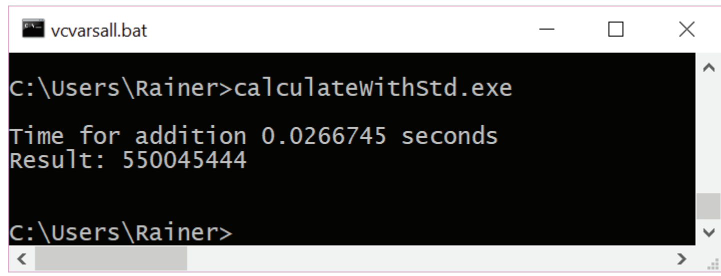
Суммирование алгоритмом std::accumulate

```
// calculateWithStd.cpp
...
const unsigned long long sum = std::accumulate(
    randValues.begin(),
    randValues.end(),
    0);
```

В системе Linux алгоритм std::accumulate показывает примерно такую же производительность, что и цикл по диапазону. Однако в системе Windows различие значительно, и функция std::accumulate явно выигрывает у цикла, как видно из следующих рисунков.



Суммирование алгоритмом `std::accumulate` в системе Linux



Суммирование алгоритмом `std::accumulate` в системе Windows

Теперь у нас есть опорные значения. Прежде чем переходить к параллельным алгоритмам, посмотрим ещё на две однопоточные реализации: одну с блокировщиком и другую – с атомарной переменной. Тем самым мы сможем точно измерить накладные расходы, которые влечёт за собой использование этих средств синхронизации, в отсутствие коллизий.

6.1.1.3. Использование блокировщика

Если доступ к переменной-накопителю суммы защитить блокировщиком, можно получить ответы на следующие вопросы.

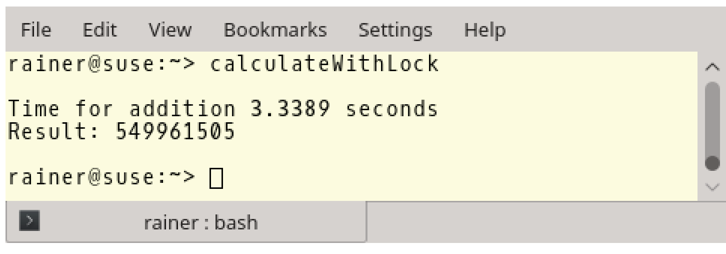
1. Сколь затратна синхронизация посредством блокировщика при отсутствии коллизий?
2. Сколь быстрым может быть блокировщик в оптимальном случае?

Ниже показан лишь фрагмент кода, непосредственно связанный с использованием блокировщика `std::lock_guard`. Полный исходный код можно загрузить на сайте книги.

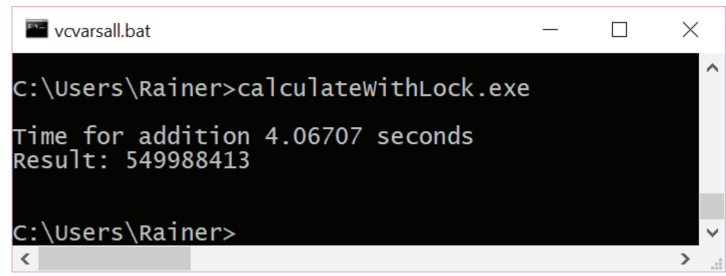
Суммирование с блокировкой доступа к переменной-накопителю

```
// calculateWithLock.cpp
...
  std::mutex myMutex;

  for (auto i: randValues){
      std::lock_guard<std::mutex> myLockGuard(myMutex);
      sum += i;
  }
```

Время выполнения алгоритма изменилось ожидаемым образом: доступ к переменной `sum` теперь осуществляется медленнее, как видно из следующих рисунков.



Суммирование с блокировкой в системе Linux



Суммирование с блокировкой в системе Windows

Таким образом, использование блокировщика `std::lock_guard`, даже при полном отсутствии коллизий, обходится в 50–150 раз дороже, чем использование стандартного алгоритма `std::accumulate`. Посмотрим теперь, как обстоит дело с атомарными переменными.

6.1.1.4. Использование атомарной переменной

Для атомарных переменных нужно дать ответы на те же два вопроса, которые интересовали нас в связи с блокировщиками.

1. Насколько дорого обходится использование атомарных переменных?
2. Сколь быстрой может быть атомарная переменная в отсутствие коллизий?

Кроме того, поставим ещё один вопрос: каково различие в производительности блокировщиков и атомарных переменных?

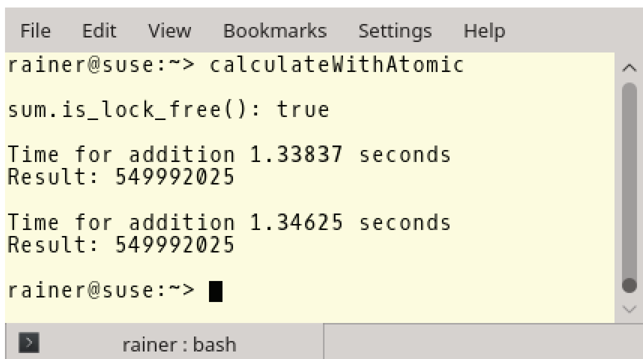
Суммирование с использованием атомарной переменной-накопителя

```
1  // calculateWithAtomic.cpp
2
3  #include <atomic>
4  #include <chrono>
```

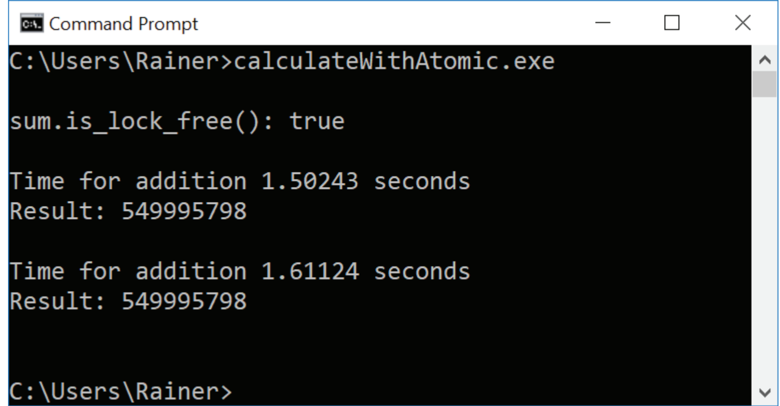
```
5  #include <iostream>
6  #include <numeric>
7  #include <random>
8  #include <vector>
9
10 constexpr long long size = 100000000;
11
12 int main(){
13   std::cout << std::endl;
14
15   std::vector<int> randValues;
16   randValues.reserve(size);
17
18   // random values
19   std::random_device seed;
20   std::mt19937 engine(seed());
21   std::uniform_int_distribution<> uniformDist(1, 10);
22   for (long long i = 0 ; i < size ; ++i)
23       randValues.push_back(uniformDist(engine));
24
25   std::atomic<unsigned long long> sum = {};
26   std::cout << std::boolalpha << "sum.is_lock_free(): "
27             << sum.is_lock_free() << std::endl;
28   std::cout << std::endl;
29
30   auto sta = std::chrono::steady_clock::now();
31
32   for (auto i: randValues) sum += i;
33
34   std::chrono::duration<double> dur =
35     std::chrono::steady_clock::now() - sta;
36   std::cout << "Time for addition " << dur.count()
37             << " seconds" << std::endl;
38   std::cout << "Result: " << sum << std::endl;
39
40   std::cout << std::endl;
41
42   sum=0;
43   sta = std::chrono::steady_clock::now();
44
45   for (auto i: randValues) sum.fetch_add(i);
46
47   dur = std::chrono::steady_clock::now() - sta;
48   std::cout << "Time for addition " << dur.count()
49             << " seconds" << std::endl;
50   std::cout << "Result: " << sum << std::endl;
51
52   std::cout << std::endl;
53 }
```

Сначала, в строке 27, выполняется проверка, не использует ли переменная `sum` блокировку. Эта проверка необходима, так как в худшем случае может не

оказаться разницы между использованием атомарных переменных и блокировок, как в предыдущей программе. На всех сколько-нибудь распространённых платформах, известных автору, атомарные переменные свободны от блокировок. Сумма в этой программе вычисляется двумя способами. В строке 32 используется перегруженная операция `+=`, тогда как в строке 45 – функция `std::fetch_add`. При выполнении в один поток оба варианта демонстрируют примерно одинаковую производительность, однако функция `std::fetch_add` позволяет в явном виде указывать порядок доступа к памяти, что будет использоваться в дальнейших разделах. Результаты работы программы показаны ниже.



Суммирование с использованием атомарной переменной в системе Linux



Суммирование с использованием атомарной переменной в системе Windows

6.1.1.5. Сводные данные по однопоточным алгоритмам

Анализ полученных данных позволяет прийти к следующим трём выводам.

1. В системах Windows и Linux атомарные переменные ведут себя в 12–50 раз медленнее, чем алгоритм `std::accumulate` без синхронизации.

2. Атомарные переменные в 2–3 раза быстрее блокировок в системах Windows и Linux.
3. Реализация функции `std::accumulate`, доступная в системе Windows, по-видимому, чрезвычайно оптимизирована.

Прежде чем рассмотреть многопоточные реализации, соберём в таблицу результаты работы всех однопоточных алгоритмов. Все значения даны в секундах.

Производительность однопоточных алгоритмов

Операционная система и компилятор	Цикл по диапазону	Функция accumulate	Блокировщик	Атомарная переменная
Linux (GCC)	0,07	0,07	3,34	1,34 1,33
Windows (cl.exe)	0,08	0,03	4,07	1,50 1,61

6.1.2. Многопоточное суммирование с общей переменной

Читатель наверняка уже может предвидеть результат этого раздела. Использование общей переменной из четырёх потоков неоптимально, так как накладные расходы на синхронизацию доступа к ней превышают выигрыш от распараллеливания вычислений. В последующих разделах будут получены соответствующие числовые данные. Вопросы, на которые предстоит найти ответ, остаются прежними.

1. Насколько различается производительность алгоритмов суммирования с использованием блокировки и атомарной переменной?
2. Насколько различается производительность однопоточного и многопоточного выполнений функции `std::accumulate`?

6.1.2.1. Использование блокировщика

Простейший способ сделать суммирование потокобезопасным – использовать блокировщик `std::lock_guard`.

Многопоточное суммирование с блокировкой

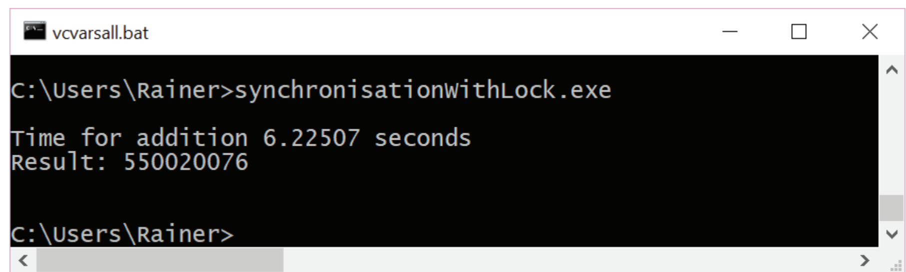
```
1  // synchronisationWithLock.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <mutex>
6  #include <random>
7  #include <thread>
8  #include <utility>
9  #include <vector>
```

```
10
11 constexpr long long size = 100000000;
12
13 constexpr long long fir =  25000000;
14 constexpr long long sec =  50000000;
15 constexpr long long thi =  75000000;
16 constexpr long long fou = 100000000;
17
18 std::mutex myMutex;
19
20 void sumUp(unsigned long long& sum, const std::vector<int>& val,
21            unsigned long long beg, unsigned long long end){
22     for (auto it = beg; it < end; ++it){
23         std::lock_guard<std::mutex> myLock(myMutex);
24         sum += val[it];
25     }
26 }
27
28 int main(){
29   std::cout << std::endl;
30
31   std::vector<int> randValues;
32   randValues.reserve(size);
33
34   std::mt19937 engine;
35   std::uniform_int_distribution<> uniformDist(1,10);
36   for (long long i = 0 ; i < size ; ++i)
37       randValues.push_back(uniformDist(engine));
38
39   unsigned long long sum= 0;
40   const auto sta = std::chrono::steady_clock::now();
41
42   std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
43   std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
44   std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
45   std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
46
47   t1.join();
48   t2.join();
49   t3.join();
50   t4.join();
51
52   const std::chrono::duration<double> dur =
53     std::chrono::steady_clock::now() - sta;
54   std::cout << "Time for addition " << dur.count()
55             << " seconds" << std::endl;
56   std::cout << "Result: " << sum << std::endl;
57
58   std::cout << std::endl;
59 }
```

Принцип работы этой программы объяснить легко. Функция `sumUp`, объявленная в строках 20–26, выполняется одновременно из нескольких потоков. Эта функция получает по ссылке переменную-накопитель `sum` и контейнер `val`. Параметры `beg` и `end` задают отрезок контейнера, в котором нужно выполнить суммирование. Блокировщик типа `std::lock_guard` в строке 23 используется для того, чтобы защитить общую переменную от гонки данных. Каждый из четырёх потоков (строки 42–45) выполняет четверть общего объёма вычислений. Результаты измерения производительности этой программы показаны ниже.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> synchronisationWithLock

Time for addition 20.8111 seconds
Result: 549996948

rainer@suse:~>
rainer : bash
```

Суммирование с общей переменной и блокировщиком в системе Linux



Суммирование с общей переменной и блокировщиком в системе Windows

Столь низкая производительность объясняется наличием узкого места – общей переменной `sum`, доступ к которой синхронизируется тяжеловесным блокировщиком `std::lock_guard`. Сразу приходит в голову очевидное решение: заменить тяжеловесный блокировщик быстрой атомарной переменной.

Сокращённые примеры кода

Для экономии места в оставшейся части этого раздела вместо полного кода программы будет приводиться лишь функция `SumUp`, поскольку остальные части программы остаются почти неизменными. Полный текст примеров можно найти на сайте книги.

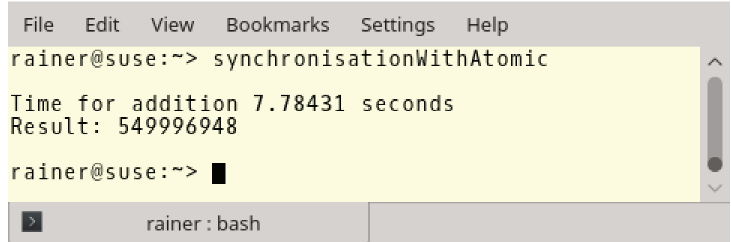
6.1.2.2. Использование атомарной переменной

Сделаем теперь переменную-накопитель суммы атомарной. Это означает, что блокировщик `std::lock_guard` более не нужен. Ниже показана видоизменённая функция `sumUp`.

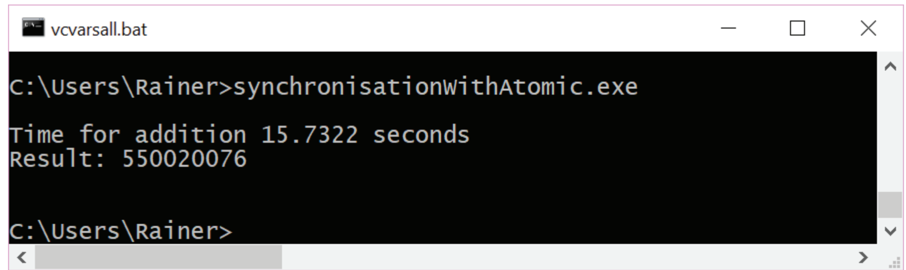
Суммирование элементов вектора с помощью атомарной переменной

```
// synchronisationWithAtomic.cpp
...
void sumUp(
    std::atomic<unsigned long long>& sum,
    const std::vector<int>& val,
    unsigned long long beg,
    unsigned long long end)
{
    for (auto it = beg; it < end; ++it) {
        sum += val[it];
    }
}
```

В операционной системе Windows эта программа показала довольно странный результат: синхронизация с помощью блокировщика оказалась быстрее решения не основе атомарной переменной более чем вдвое. Результат работы программы показан на рисунке.



Суммирование с помощью атомарной переменной в системе Linux



Суммирование с помощью атомарной переменной в системе Windows

Помимо операции `+=`, к атомарной переменной можно применять функцию-член `fetch_add`. Посмотрим, что из этого получится.

6.1.2.3. Использование атомарной переменной с функцией fetch_add

Исходный код почти не подвергся изменениям. Лишь в одной строке пришлось изменить операцию сложения на вызов функции-члена fetch_add.

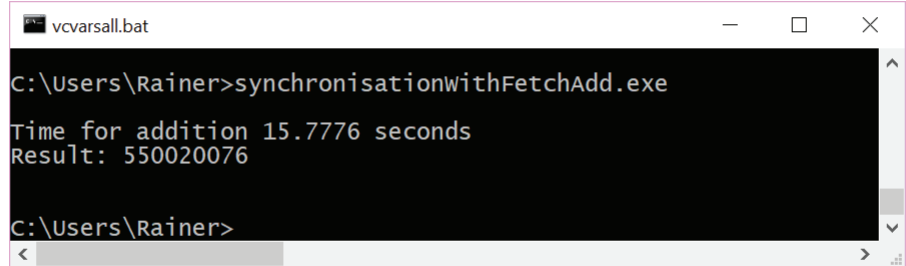
Суммирование с атомарной переменной с функцией fetch_add

```
// synchronisationWithFetchAdd.cpp
...
void sumUp(
    std::atomic<unsigned long long>& sum,
    const std::vector<int>& val,
    unsigned long long beg,
    unsigned long long end)
{
    for (auto it = beg; it < end; ++it) {
        sum.fetch_add(val[it]);
    }
}
```

Эта программа обладает почти такой же производительностью, как и предыдущий пример. Таким образом, различие между операцией += и функцией fetch_add оказалось незначительным.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> synchronisationWithFetchAdd

Time for addition 7.87254 seconds
Result: 549996948

rainer@suse:~>
rainer : bash
```

Суммирование с функцией fetch_add в системе Linux



Суммирование с функцией fetch_add в системе Windows

Хотя замена перегруженной операции += функцией fetch_add никак не сказалась на производительности программы, последний вариант всё же обладает одним преимуществом: он позволяет в явном виде ослабить требования к упорядочению доступа к памяти, то есть применить ослабленную семантику.

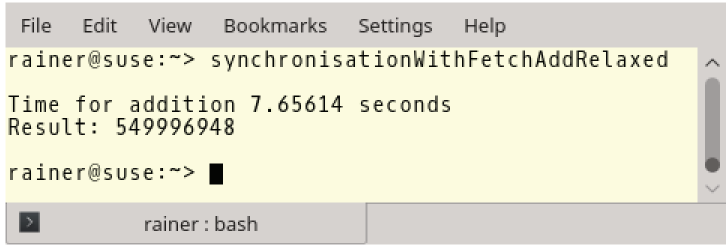
6.1.2.4. Использование ослабленной семантики

Изменение в исходном коде снова затрагивает лишь одну строку.

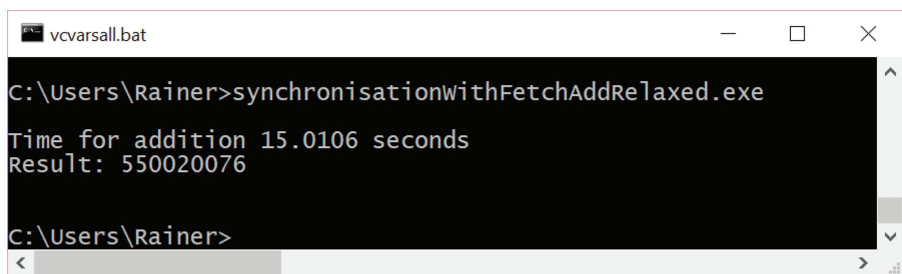
```
// synchronisationWithFetchAddRelaxed.cpp
...
void sumUp(
    std::atomic<unsigned long long>& sum,
    const std::vector<int>& val,
    unsigned long long beg,
    unsigned long long end)
{
    for (auto it = beg; it < end; ++it) {
        sum.fetch_add(val[it], std::memory_order_relaxed);
    }
}
```

По умолчанию атомарные переменные работают в модели последовательной согласованности. Это справедливо для комбинированной операции сложения с присваиванием и, конечно, для функции fetch_add, но поведение последней можно оптимизировать. В этой программе порядок доступа к памяти заменён с используемого по умолчанию на ослабленный. Это слабейшая из всех моделей памяти и тем самым конечная точка оптимизации решения на базе общей переменной.

Ослабленная семантика допустима в данном случае потому, что предоставляет две гарантии: во-первых, каждая из операций fetch_add происходит атомарно, во-вторых – операции, выполняемые из различных потоков, окончательно синхронизируются между собой вызовом функции join. Слабейшая модель памяти позволяет добиться наилучшей (среди решений, основанных на общей переменной) производительности, как явствует из следующих рисунков.



Суммирование с ослабленной семантикой в системе Linux



Суммирование с ослабленной семантикой в системе Windows

6.1.2.5. Сводные данные по алгоритмам с общей переменной

Полученные в описанных экспериментах данные сведены в следующую таблицу. Все величины даны в секундах.

Производительность многопоточных алгоритмов с общей переменной

Операционная система и компилятор	lock_guard	+=	fetch_add	Ослабленная семантика
Linux (GCC)	20,81	7,78	7,87	7,66
Windows (cl.exe)	6,22	15,73	15,78	15,01

Результаты экспериментов выглядят не слишком обнадёживающими. Вариант с атомарной общей переменной и ослабленной семантикой операций над ней работает примерно в сто раз медленнее, чем однопоточный алгоритм `std::accumulate`.

Попробуем теперь соединить две предыдущие стратегии суммирования. Будем использовать четыре потока, но сведём к минимуму синхронизацию между ними.

6.1.3. Раздельное суммирование в потоках

Есть несколько способов минимизировать синхронизацию потоков. Так, можно воспользоваться локальными переменными, переменными потоков и заданиями.

6.1.3.1. Использование локальной переменной

Если каждый поток пользуется собственной локальной переменной для подсчёта суммы своего участка контейнера, он может выполнять свою работу без какой бы то ни было синхронизации. Синхронизация понадобится только для того, чтобы сложить между собой результаты работы отдельных потоков. Последнее сложение играет в алгоритме ключевую роль, и его, конечно, нужно защитить от гонки данных. Есть разные способы сделать это. Отметим, что поскольку подсчёт окончательного результата требует сложения лишь

четырёх чисел, для производительности алгоритма не имеет значения, какой способ синхронизации при этом используется. В следующих примерах применяются: блокировщик `std::lock_guard`, атомарная переменная с последовательной согласованностью операций и с ослабленной семантикой.

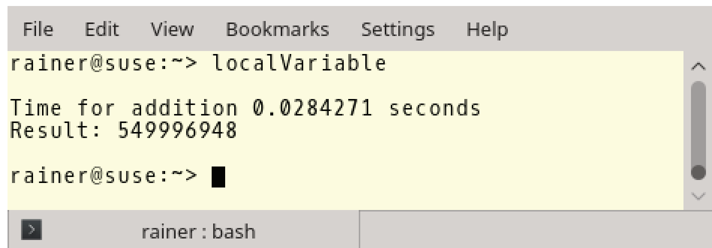
6.1.3.1.1. Синхронизация итоговой суммы блокировщиком std::lock_guard

Раздельное суммирование и минимальная синхронизация блокировщиком

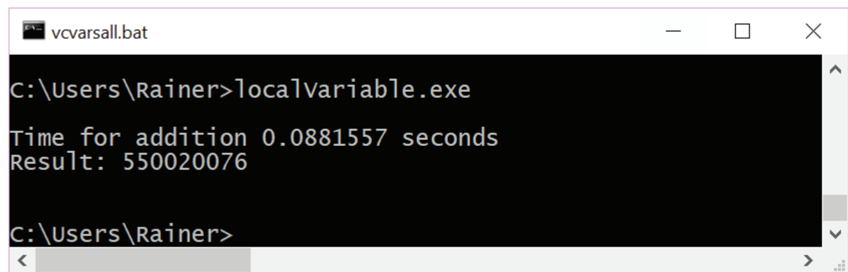
```
1  // localVariable.cpp
2
3  #include <mutex>
4  #include <chrono>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <utility>
9  #include <vector>
10
11 constexpr long long size = 100000000;
12
13 constexpr long long fir =  25000000;
14 constexpr long long sec =  50000000;
15 constexpr long long thi =  75000000;
16 constexpr long long fou = 100000000;
17
18 std::mutex myMutex;
19
20 void sumUp(unsigned long long& sum, const std::vector<int>& val,
21            unsigned long long beg, unsigned long long end){
22     unsigned long long tmpSum{};
23     for (auto i = beg; i < end; ++i){
24         tmpSum += val[i];
25     }
26     std::lock_guard<std::mutex> lockGuard(myMutex);
27     sum += tmpSum;
28 }
29
30 int main(){
31   std::cout << std::endl;
32
33   std::vector<int> randValues;
34   randValues.reserve(size);
35
36   std::mt19937 engine;
37   std::uniform_int_distribution<> uniformDist(1, 10);
38   for (long long i = 0; i < size; ++i)
39        randValues.push_back(uniformDist(engine));
40
41   unsigned long long sum{};
42   const auto sta = std::chrono::system_clock::now();
43
44   std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
```

```
45   std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
46   std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
47   std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
48
49   t1.join();
50   t2.join();
51   t3.join();
52   t4.join();
53
54   const std::chrono::duration<double> dur =
55     std::chrono::system_clock::now() - sta;
56
57
58   std::cout << "Time for addition " << dur.count()
59             << " seconds" << std::endl;
60   std::cout << "Result: " << sum << std::endl;
61
62   std::cout << std::endl;
63 }
```

Здесь представляют интерес строки 26 и 27. В них локальная сумма `tmpSum` прибавляется к глобальному накопителю `sum`. Производительность этой программы показана на следующих рисунках.



Суммирование на основе локальной переменной в системе Linux



Суммирование на основе локальной переменной в системе Windows

В следующих двух вариантах решения с локальной переменной меняется лишь функция `sumUp`, поэтому только её текст и будет показан. Полный исходный текст примеров можно найти на сайте.

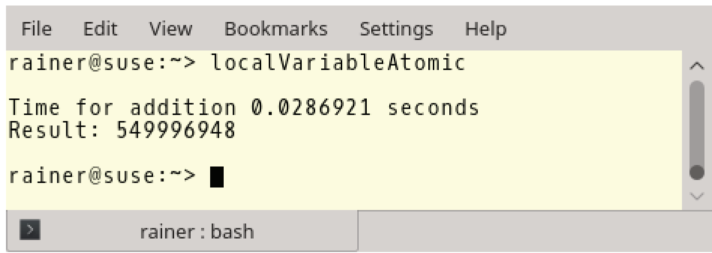
6.1.3.1.2. Сумма в атомарной переменной с последовательной согласованностью

Изменим тип глобальной переменной `sum`, в которой накапливается итоговая сумма, на атомарный.

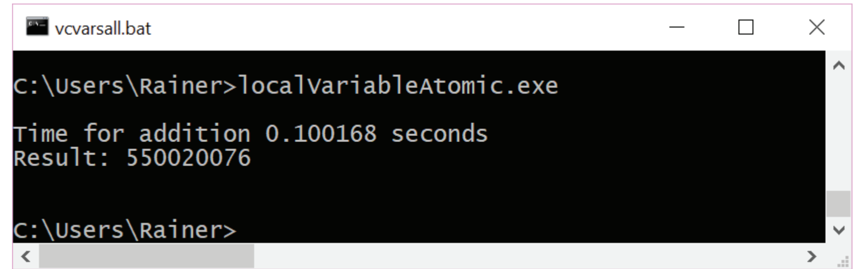
Раздельное суммирование с итоговой суммой в атомарной переменной

```
1  // localVariableAtomic.cpp
2  ...
3  void sumUp(
4      std::atomic<unsigned long long>& sum,
5      const std::vector<int>& val,
6      unsigned long long beg,
7      unsigned long long end)
8  {
9      unsigned long long tmpSum{};
10     for (auto i = beg; i < end; ++i){
11         tmpSum += val[i];
12     }
13     sum += tmpSum;
14 }
```

Измерение производительности даёт следующие результаты.



Суммирование на основе локальной переменной в системе Linux



Суммирование на основе локальной переменной в системе Windows

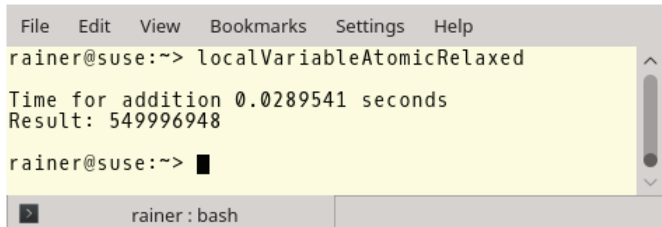
6.1.3.1.3. Использование атомарной переменной с ослабленной семантикой

Предыдущее решение можно ещё немного улучшить. Воспользуемся ослабленной семантикой вместо порядка доступа к памяти по умолчанию. Поведение программы останется вполне определённым, так как для её правильного функционирования нужны лишь две гарантии: что все операции прибавления к итоговой сумме произойдут и будут атомарными.

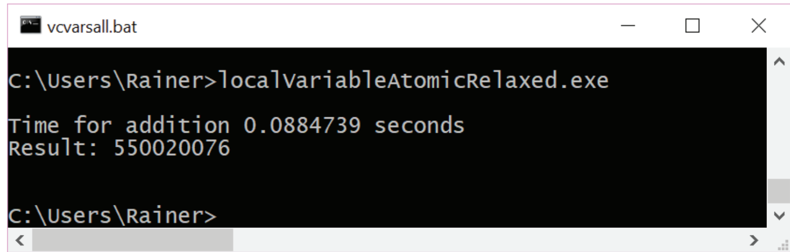
Раздельное суммирование вектора с ослабленной семантикой

```
1  // localVariableAtomicRelaxed.cpp
2  ...
3  void sumUp(
4      std::atomic<unsigned long long>& sum,
5      const std::vector<int>& val,
6      unsigned long long beg,
7      unsigned long long end)
8  {
9      unsigned long long tmpSum{};
10     for (auto i = beg; i < end; ++i){
11         tmpSum += val[i];
12     }
13     sum.fetch_add(tmpSum, std::memory_order_relaxed);
14 }
```

Как и можно было ожидать, производительность алгоритма с раздельным суммированием никак не зависит от того, используется в ней блокировщик, атомарная переменная с последовательной согласованностью операций или же ослабленная семантика.



Суммирование на основе локальной переменной в системе Linux



Суммирование на основе локальной переменной в системе Windows

Данные потоков – это частный случай локальных данных. Их время жизни ограничено временем работы владеющих ими потоков, а не временем работы одной функции, как у обычных локальных переменных наподобие переменной `tmpSum` из предыдущего примера.

6.1.3.2. Использование переменных с потоковым временем жизни

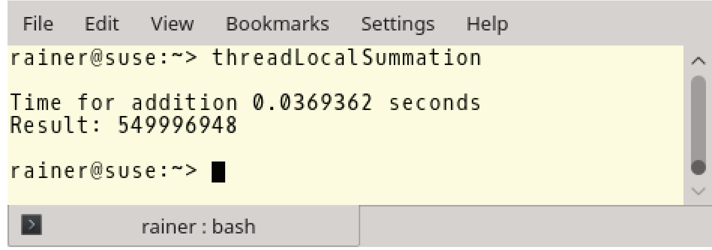
Переменная с потоковым временем жизни (также называемая потоковой переменной) принадлежит тому потоку, в котором была создана. Такая переменная появляется в памяти только тогда, когда становится нужна. Поэтому потоковая длительность хранения как нельзя лучше подходит для переменной `tmpSum`.

Раздельное суммирование с использованием потоковой переменной

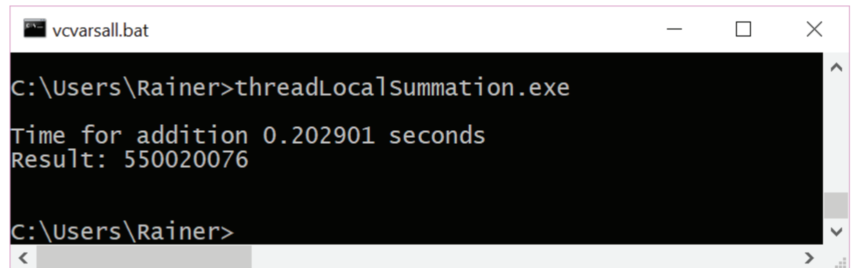
```
1  // threadLocalSummation.cpp
2
3  #include <atomic>
4  #include <chrono>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <utility>
9  #include <vector>
10
11 constexpr long long size = 100000000;
12
13 constexpr long long fir =  25000000;
14 constexpr long long sec =  50000000;
15 constexpr long long thi =  75000000;
16 constexpr long long fou = 100000000;
17
18 thread_local unsigned long long tmpSum = 0;
19
20 void sumUp(
21     std::atomic<unsigned long long>& sum,
22     const std::vector<int>& val,
23     unsigned long long beg,
24     unsigned long long end)
25 {
26     for (auto i = beg; i < end; ++i){
27         tmpSum += val[i];
28     }
29     sum.fetch_add(tmpSum, std::memory_order_relaxed);
30 }
31
```

```
32 int main(){
33   std::cout << std::endl;
34 
35   std::vector<int> randValues;
36   randValues.reserve(size);
37 
38   std::mt19937 engine;
39   std::uniform_int_distribution<> uniformDist(1, 10);
40   for (long long i = 0; i < size; ++i)
41       randValues.push_back(uniformDist(engine));
42 
43   std::atomic<unsigned long long> sum{};
44   const auto sta = std::chrono::system_clock::now();
45 
46   std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
47   std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
48   std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
49   std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
50 
51   t1.join();
52   t2.join();
53   t3.join();
54   t4.join();
55 
56   const std::chrono::duration<double> dur=
57         std::chrono::system_clock::now() - sta;
58 
59   std::cout << "Time for addition " << dur.count()
60             << " seconds" << std::endl;
61   std::cout << "Result: " << sum << std::endl;
62 
63   std::cout << std::endl;
64 }
```

Переменная `tmpSum` с потоковым временем жизни объявляется в строке 18 и используется в строках 27 и 29. Ниже представлены результаты измерения производительности этой программы.



Суммирование на основе локальной переменной в системе Linux



Суммирование на основе локальной переменной в системе Windows

Наконец, остаётся рассмотреть решение на основе асинхронных заданий.

6.1.3.3. Использование асинхронных заданий

Воспользовавшись заданиями, можно сделать всю работу, не прибегая к явной синхронизации потоков. Каждый фрагмент контейнера суммируется в отельном потоке, а окончательный результат подсчитывается в главном потоке. Полный текст программы представлен ниже.

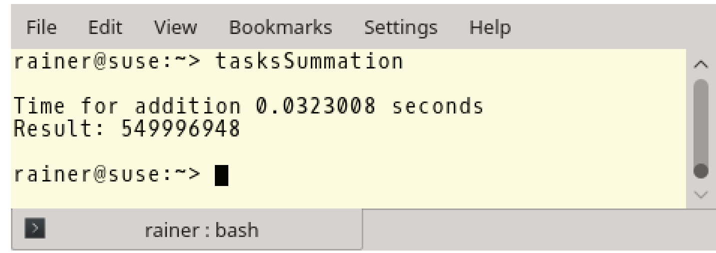
Суммирование вектора на основе заданий

```
1  // tasksSummation.cpp
2  
3  #include <chrono>
4  #include <future>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <utility>
9  #include <vector>
10 
11 constexpr long long size = 100000000;
12 
13 constexpr long long fir =  25000000;
14 constexpr long long sec =  50000000;
15 constexpr long long thi =  75000000;
16 constexpr long long fou = 100000000;
17 
18 void sumUp(
19     std::promise<unsigned long long>&& prom,
20     const std::vector<int>& val,
21     unsigned long long beg,
22     unsigned long long end)
23 {
24     unsigned long long sum={};
25     for (auto i = beg; i < end; ++i){
26         sum += val[i];
27     }
28     prom.set_value(sum);
```

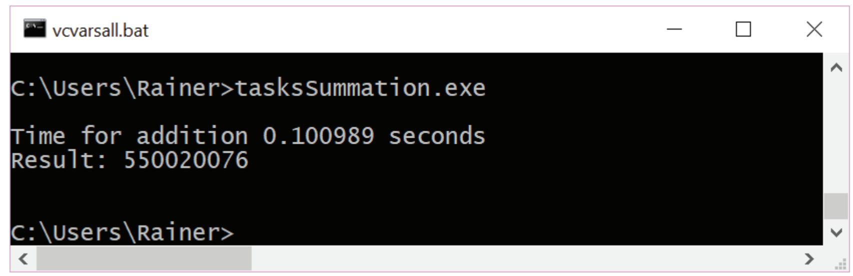
```
29 }
30
31 int main(){
32   std::cout << std::endl;
33
34   std::vector<int> randValues;
35   randValues.reserve(size);
36
37   std::mt19937 engine;
38   std::uniform_int_distribution<> uniformDist(1,10);
39   for (long long i = 0; i < size; ++i)
40       randValues.push_back(uniformDist(engine));
41
42   std::promise<unsigned long long> prom1;
43   std::promise<unsigned long long> prom2;
44   std::promise<unsigned long long> prom3;
45   std::promise<unsigned long long> prom4;
46
47   auto fut1= prom1.get_future();
48   auto fut2= prom2.get_future();
49   auto fut3= prom3.get_future();
50   auto fut4= prom4.get_future();
51
52   auto sta = std::chrono::system_clock::now();
53
54   std::thread t1(
55     sumUp, std::move(prom1), std::ref(randValues), 0, fir);
56   std::thread t2(
57     sumUp, std::move(prom2), std::ref(randValues), fir, sec);
58   std::thread t3(
59     sumUp, std::move(prom3), std::ref(randValues), sec, thi);
60   std::thread t4(
61     sumUp, std::move(prom4), std::ref(randValues), thi, fou);
62
63   auto sum= fut1.get() + fut2.get() + fut3.get() + fut4.get();
64
65   const std::chrono::duration<double> dur =
66     std::chrono::system_clock::now() - sta;
67   std::cout << "Time for addition " << dur.count()
68             << " seconds" << std::endl;
69   std::cout << "Result: " << sum << std::endl;
70
71   t1.join();
72   t2.join();
73   t3.join();
74   t4.join();
75
76   std::cout << std::endl;
77 }
```

В строках 42–45 объявляются четыре объекта-обещания, а в строках 47–50 создаются связанные с ними фьючерсы. В строках 42–45 обещания пере-

даются запускающимся потокам. Следует отметить, что обещания не могут копироваться – их можно только перемещать. Четыре потока выполняют функцию `sumUp`, реализованную в строках 18–29. Функция `sumUp` принимает свой первый аргумент – обещание – по ссылке rvalue. В строке 63 главный поток запрашивает у фьючерсов (посредством блокирующего вызова `get`) значения, подсчитанные четырьмя потоками. На следующих рисунках показаны результаты запуска этой программы.



Суммирование на основе заданий в системе Linux



Суммирование на основе заданий в системе Linux

В заключение раздела приведём сводку показателей производительности алгоритмов, основанных на использовании локальных переменных.

6.1.3.4. Сводные данные

Как явствует из результатов измерений, нет большой разницы между использованием асинхронных заданий и локальных переменных (со всевозможными механизмами синхронизации для подсчёта итогового значения, например атомарных переменных). Лишь потоковые переменные, по-видимому, несколько замедляют работу программы. Это замедление наблюдается на обоих операционных системах: Windows и Linux. Различие времени работы одного и того же алгоритма в ОС Windows и Linux не говорит ни о чём, так как измерения проводились на различных компьютерах: программа оптимизирована для четырёх ядер, тогда как имеющийся у автора компьютер с операционной системой Windows имеет лишь два. Величины в следующей таблице даны в секундах.

Производительность алгоритмов с локальными переменными

ОС и компилятор	`lock_guard`	Атомарные переменные с последовательной согласованностью	Атомарные переменные с ослабленной семантикой	Переменные потоков	Задания
Linux (GCC)	0,03	0,03	0,03	0,04	0,03
Windows (cl.exe)	0,10	0,10	0,10	0,20	0,10

Внимание привлекает одна интересная деталь. Многопоточная реализация с локальными переменными работает примерно вдвое быстрее однопоточного суммирования. Между тем можно было бы ожидать четырёхкратного увеличения производительности, ведь четыре потока выполняются на четырёх ядрах, на требуя никакой синхронизации. Почему же производительность оказывается вдвое ниже ожидаемой?

6.1.4. Суммирование вектора: подведение итогов

6.1.4.1. Однопоточные алгоритмы

Цикл по диапазону и алгоритм `std::accumulate` из стандартной библиотеки обладают сопоставимой производительностью. При включенной оптимизации компилятор может использовать для суммирования быстродействующие инструкции расширенных наборов ОКМД[1] (англ. *SIMD*) – SSE или AVX. В этом случае счётчик цикла увеличивается всякий раз на 4 (при использовании расширения SSE) или на 8 (в случае расширения AVX).

6.1.4.2. Многопоточные алгоритмы с общей переменной

Опыт использования общей переменной в качестве накопителя суммы наглядно демонстрирует, что синхронизация обходится дорого, и поэтому её следует избегать, гда только можно. Если использовать атомарную переменную и даже ослабить требование последовательной согласованности операций над ней, четыре параллельных потока выполняются примерно в сотню раз дольше, чем один поток. Когда речь идёт о быстродействии, первой целью разработчика должно стать сведение к минимуму дорогостоящих синхронизаций.

6.1.4.3. Многопоточные алгоритмы с локальными переменными

Накопление каждым из четырёх потоков своей частичной суммы в локальной переменной оказывается лишь вдвое быстрее цикла по диапазону в единственном потоке или алгоритма `std::accumulate`. Этот результат кажется удивительным, ведь от параллельной работы четырёх потоков было

[1] https://ru.wikipedia.org/wiki/SIMD.

бы естественно ожидать четырёхкратного выигрыша производительности. Ещё удивительнее оказался график использования ядер, показывающий, что каждое ядро загружено далеко не полностью.

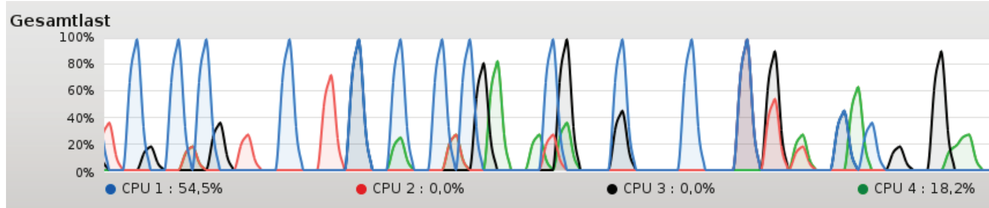
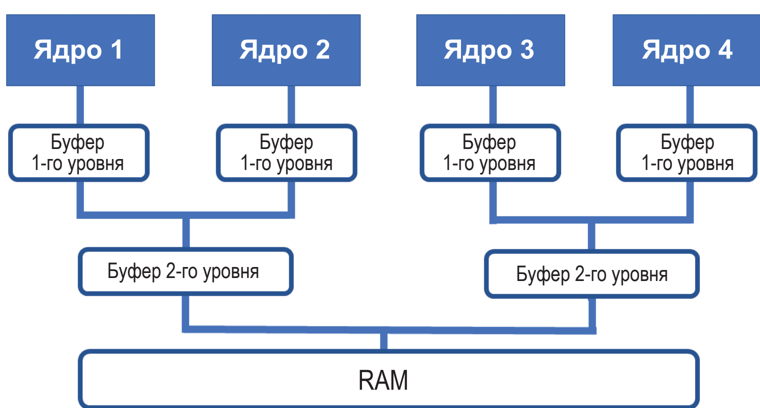


График использования четырёх ядер

Причина этого, однако, проста. Ядра не могут достаточно быстро извлекать данные из оперативной памяти. Выполнение этого алгоритма ограничено возможностями памяти[1] (англ. *memory bound*). Иначе говоря, это память замедляет работу процессорных ядер. Следующий рисунок поясняет, где находится узкое место.



Узкое место при работе с памятью

Наглядная модель производительности систем, известная как модель кровли[2], позволяет оценивать производительность приложений, работающих на многоядерных или многопроцессорных архитектурах. В этой модели учитываются пиковая производительность, пиковая пропускная способность и интенсивность использования системы[3].

---

1 https://en.wikipedia.org/wiki/Memory_bound_function.
2 https://en.wikipedia.org/wiki/Roofline_model.
3 Имеется в виду аппроксимированный график производительности, состоящий из двух участков: линейного при относительно малых нагрузках и горизонтального при нагрузках, превышающих известный предел. Пока нагрузка на систему невелика, производительность системы растёт линейно с ростом интенсивности её

6.2. Потокобезопасное создание объекта-одиночки

Перед тем как погрузиться в указанную тему, следует подчеркнуть: автор вовсе не пропагандирует использование шаблона «Одиночка». Разбор потокобезопасных способов инициализации одиночки необходимо начать с предостережения.

Несколько мыслей об объектах-одиночках

Единственная причина, по которой шаблон «Одиночка» используется в этой книге, состоит в том, что он представляет собой классический пример переменной, которая должна быть проинициализирована потокобезопасным образом. Как шаблон проектирования, объект-одиночка обладает рядом серьёзных недостатков. Ниже приведены некоторые его проблемы.

- Объект-одиночка есть, в сущности, глобальная переменная. Это обстоятельство заметно затрудняет тестирование программ, так как их поведение зависит от глобального состояния.
- Для того чтобы использовать объект-одиночку в некоторой функции, чаще всего вызывают статическую функцию-член его класса – например, `MySingleton::getInstance()`. Это означает, что интерфейс функции никак не сообщает её пользователю, что внутри неё используется объект этого класса. Зависимость от одиночного объекта оказывается скрытой в деталях реализации.
- Если в разных исходных файлах определены два статических объекта, `x` и `y`, и конструктор каждого из них зависит от другого объекта, имеет место ситуация, известная как фиаско статической инициализации[1], поскольку нет никакой гарантии, какой из статических объектов будет инициализирован первым. Объекты-одиночки представляют собой именно статические объекты.
- Шаблон «Одиночка» управляет отложенным созданием объекта, но не его уничтожением[2]. Ситуация, когда ставший ненужным объект не уничтожается, называется утечкой ресурсов (в частности, памяти).
- Предположим, программист захочет объявить подкласс класса-одиночки. Как этот подкласс должен быть устроен? Как им пользоваться в программе? Можно ли такое допускать?
- Потокобезопасная и быстрая инициализация одиночек – довольно сложное дело.

использования. Однако при дальнейшем повышении интенсивности наступает насыщение – система не может выдавать более определённого числа результатов в единицу времени, и график производительности обращается в горизонтальную прямую. В случае многопроцессорной или многоядерной системы, рассматриваемой в данном разделе, насыщение наступает при достижении предела пропускной способности шины данных. Ядра процессора могли бы обработать и больше данных, но они не успевают поступать с такой скоростью из оперативной памяти. – *Прим. перев.*

[1] https://isocpp.org/wiki/faq/ctors.

[2] Это утверждение спорно. Не составляет труда создать расширенную реализацию шаблона «Одиночка», в которой помимо статической функции `getInstance` присутствовала бы противоположная ей статическая функция `destroyInstance`, ответственная за потокобезопасное уничтожение объекта-одиночки, освобождение всех выделенных ему ресурсов и обнуление статической переменной-указателя. – *Прим. перев.*

> Более подробное обсуждение преимуществ и недостатков шаблона проектирования «Одиночка» можно найти по ссылкам из обзорной статьи о нём[1].

Разбор подходов к потокобезопасной инициализации объектов-одиночек начнём с одного вырожденного случая.

6.2.1. Шаблон «Блокировка с двойной проверкой»

Блокировка с двойной проверкой[2] считается классическим способом потокобезопасной инициализации объектов-одиночек. Однако то, что выглядит устоявшейся общепринятой практикой и шаблоном проектирования, на деле представляет собой антишаблон[3]. Этот подход предполагает гарантии относительно модели памяти, которые имелись в традиционных реализациях, но более не предоставляются языками Java, C# или C++. Ложное предположение состоит в атомарности операции создания объекта. Как следствие решение, кажущееся потокобезопасным, таковым на самом деле не является.

Что представляет собой блокировка с двойной проверкой? Этот шаблон придуман для того, чтобы улучшить производительность простейшего потокобезопасного метода инициализации одиночки, где инициализация выполняется под блокировкой.

Потокобезопасная инициализация с простой блокировкой

```
1  std::mutex myMutex;
2
3  class MySingleton{
4  public:
5    static MySingleton& getInstance() {
6      std::lock_guard<mutex> myLock(myMutex);
7      if(!instance) instance = new MySingleton();
8      return *instance;
9    }
10
11 private:
12   MySingleton() = default;
13   ~MySingleton() = default;
14   MySingleton(const MySingleton&) = delete;
15   MySingleton& operator= (const MySingleton&) = delete;
16   static MySingleton* instance;
17 };
18
19 MySingleton* MySingleton::instance = nullptr;
```

Хорошее ли это решение? И да, и нет. Да, поскольку оно совершенно потокобезопасно. Нет – потому что его быстродействие оставляет желать луч-

---

[1] https://ru.wikipedia.org/wiki/Одиночка_%28шаблон_проектирования%29.

[2] https://www.dre.vanderbilt.edu/~schmidt/PDF/DC-Locking.pdf.

[3] https://ru.wikipedia.org/wiki/Антипаттерн.

шего. Каждое обращение к объекту-одиночке упирается в тяжеловесную блокировку в строке 7. Однако все обращения после инициализации происходят исключительно в режиме чтения, и блокировка становится не нужна. Для борьбы с этой проблемой и был предложен шаблон с двойной проверкой. Посмотрим на функцию `getInstance` после небольшого изменения.

Блокировка с двойной проверкой

```
1 static MySingleton& getInstance(){
2   if (!instance){                               // проверка
3     lock_guard<mutex> myLock(myMutex);          // блокировка
4     if(!instance) instance = new MySingleton(); // проверка
5   }
6   return *instance;
7 }
```

Прежде чем использовать тяжеловесную блокировку, в строке 2 используется быстрая операция сравнения указателей. Только если указатель `instance` пуст, захватывается мьютекс (строка 3). Поскольку существует возможность того, что какой-то другой поток успел проинициализировать указатель между операцией сравнения в строке 2 и блокировкой в строке 3, значение указателя необходимо проверить ещё раз, это делается в строке 4. Таким образом, шаблон вполне оправдывает своё название: два раза выполняется проверка и один раз – блокировка.

Остроумно? Да. Потокобезопасно? Нет. В чём проблема? Операция присваивания в строке 4 состоит из по меньшей мере трёх шагов:

1) выделить память для объекта `MySingleton`;
2) проинициализировать объект `MySingleton`;
3) присвоить адрес объекта `MySingleton` в переменную `instance`.

Проблема состоит в том, что реализация языка C++ никоим образом не гарантирует, что эти шаги будут выполнены именно в таком порядке. Процессор вполне может поменять их местами, получив в итоге последовательность 1, 3, 2. В этом случае сначала выделяется память, затем переменной `instance` присваивается адрес неинициализированного объекта. Если в этот момент другой поток попытается получить доступ к единственному экземпляру, проверка в строке 2 покажет, что экземпляр уже создан. Следовательно, второй поток получит ссылку на неинициализированный объект, что делает поведение программы неопределённым.

6.2.2. Измерение производительности

Хотелось бы измерить накладные расходы на доступ к объекту-одиночке. За основу для сравнения различных подходов берётся алгоритм, который в одном потоке получает доступ к объекту-одиночке 40 миллионов раз подряд. Конечно, первое из этих обращений инициализирует объект. С этим базовым случаем будем сравнивать программу, которая обращается к одиночному объекту из четырёх параллельных потоков. Чтобы получить общий показа-

тель производительности, нужно сложить время выполнения всех четырёх потоков. Таким способом сравним производительность реализаций, основанных на статической переменной с ограниченной областью видимости, известную как реализация Мейерса, на блокировщике `std::lock_guard`, функции `std::call_once` в сочетании с флагом `std::once_flag`, а также на атомарных переменных с семантикой последовательной согласованности и с семантикой захвата и освобождения.

Измерения скорости работы программ проводятся на двух компьютерах. Компьютер под управлением ОС Linux с компилятором GCC имеет четыре процессорных ядра, а компьютер с ОС Windows и компилятором cl.exe – два. Во всех экспериментах программы компилируются с максимальным уровнем оптимизации. Читателю рекомендуется освежить в памяти замечание, сделанное в самом начале главы о сравнении производительности программы на разных компьютерах.

Эксперименты должны дать ответы на два вопроса:

1. Как соотносится производительность различных реализаций объекта-одиночки?
2. Имеются ли существенные различия между работой этих алгоритмов в ОС Linux с компилятором GCC и в ОС Windows с компилятором cl.exe?

Результаты всех измерений будут сведены в таблицу.

Как было указано выше, измерению производительности различных многопоточных реализаций должна предшествовать однопоточная. Представленная в следующем коде функция `getInstance` не является потокобезопасной согласно стандарту C++ 03.

Однопоточная реализация объекта-одиночки по Мейерсу

```
1  // singletonSingleThreaded.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  constexpr auto tenMill = 10000000;
7
8  class MySingleton {
9  public:
10   static MySingleton& getInstance() {
11     static MySingleton instance;
12     volatile int dummy{};
13     return instance;
14   }
15 private:
16   MySingleton() = default;
17   ~MySingleton() = default;
18   MySingleton(const MySingleton&) = delete;
19   MySingleton& operator=(const MySingleton&) = delete;
20 };
21
22 int main() {
```

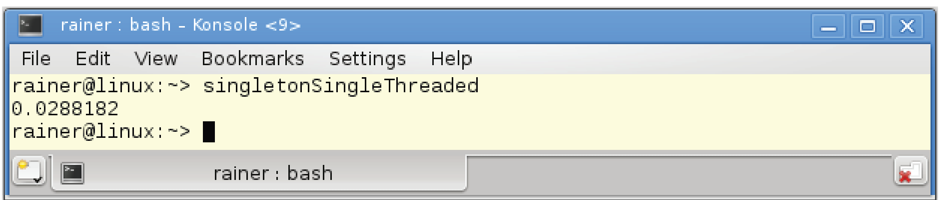
```
23   constexpr auto fourtyMill = 4 * tenMill;
24
25   auto begin= std::chrono::system_clock::now();
26
27   for ( size_t i = 0; i <= fourtyMill; ++i) {
28     MySingleton::getInstance();
29   }
30
31   auto end = std::chrono::system_clock::now() - begin;
32
33   std::cout << std::chrono::duration<double>(end).count() << std::endl;
34 }
```

В этой базовой реализации используется так называемый объект-одиночка Мейерса, названный так в честь автора, Скотта Мейерса[1]. Красота этого подхода состоит в том, что единственный экземпляр в строке 11 объявлен как статическая переменная с ограниченной областью видимости. Стандарт языка гарантирует, что инициализация выполняется лишь один раз – а именно когда статическая функция-член `getInstance` (строки 10–14) вызывается в первый раз.

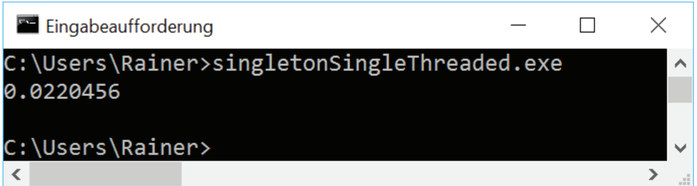
Зачем нужна переменная dummy

Если бы не было объявления переменной `dummy` с квалификатором `volatile` в строке 12, компилятор просто убрал бы вызов функции `getInstance` в строке 28, потому что результат этого вызова никак не используется. Однако наличие `volatile`-переменной запрещает оптимизатору устранять вызов функции в строке 28.

Ниже показан результат запуска этой однопоточной программы, дающий базу для сравнения многопоточных реализаций.



Объект-одиночка Мейерса при работе в один поток в ОС Linux



Объект-одиночка Мейерса при работе в один поток в ОС Windows

[1] https://en.wikipedia.org/wiki/Scott_Meyers.

6.2.3. Потокобезопасный вариант реализации Мейерса

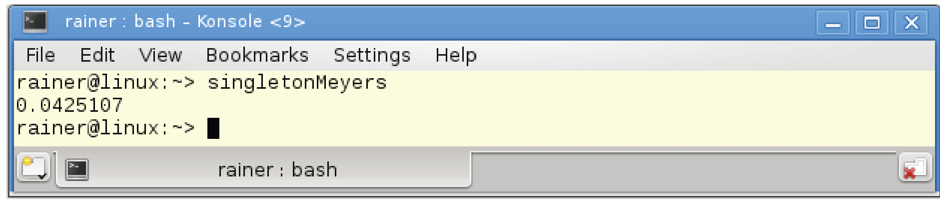
Стандарт C++ 11 гарантирует, что статические переменные с областью видимости, ограниченной блоком, инициализируются потокобезопасным образом. В представленной выше реализации Мейерса статическая переменная объявлена внутри блока, поэтому никаких усилий от программиста более не требуется. Единственное, что нужно сделать, – это изменить главную функцию программы, чтобы она обращалась к объекту-одиночке из нескольких параллельных потоков.

Реализация объекта-одиночки по Мейерсу в многопоточной среде

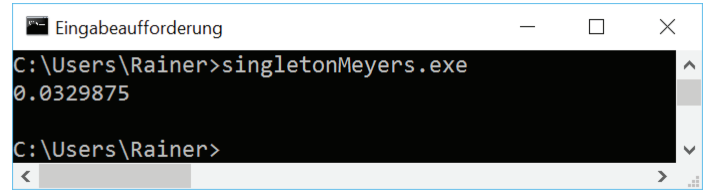
```
1  // singletonMeyers.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <future>
6
7  constexpr auto tenMill = 10000000;
8
9  class MySingleton{
10 public:
11   static MySingleton& getInstance() {
12     static MySingleton instance;
13     volatile int dummy{};
14     return instance;
15   }
16 private:
17   MySingleton() = default;
18   ~MySingleton() = default;
19   MySingleton(const MySingleton&) = delete;
20   MySingleton& operator=(const MySingleton&) = delete;
21 };
22
23 std::chrono::duration<double> getTime() {
24   const auto begin = std::chrono::system_clock::now();
25   for (size_t i = 0; i <= tenMill; ++i) {
26       MySingleton::getInstance();
27   }
28   return std::chrono::system_clock::now() - begin;
29 };
30
31 int main() {
32     auto fut1= std::async(std::launch::async, getTime);
33     auto fut2= std::async(std::launch::async, getTime);
34     auto fut3= std::async(std::launch::async, getTime);
35     auto fut4= std::async(std::launch::async, getTime);
36
37     const auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
```

```
38
39     std::cout << total.count() << std::endl;
40 }
```

Обращение к объекту-одиночке происходит из функции `getTime`, объявленной в строках 23–29. Эта функция запускается из четырёх обещаний, которые создаются в строках 32–35. Результаты, полученные из фьючерсов, суммируются в строке 37. Вот и всё: остаётся лишь посмотреть на результат выполнения программы.



Объект-одиночка Мейерса в многопоточной среде в ОС Linux



Объект-одиночка Мейерса в многопоточной среде в ОС Windows

Сокращение текста примеров

Функция `getTime`, которая измеряет время выполнения потока, и функция `main` во всех примерах практически одинаковы. Поэтому будем опускать их в последующих разделах. Полные тексты примеров можно найти на сайте книги.

Перейдём теперь к наиболее очевидной из реализаций с явной синхронизацией, а именно к реализации на основе блокировщика.

6.2.4. Реализации на основе блокировщика

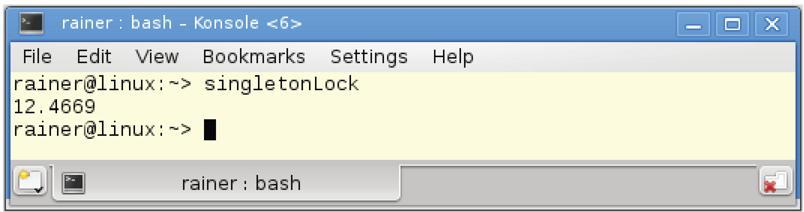
Мьютекс, завёрнутый в блокировщик `std::lock_guard`, гарантирует потокобезопасную реализацию объекта-одиночки.

Объект-одиночка с блокировщиком

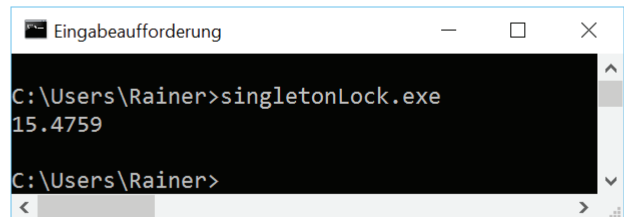
```
1  // singletonLock.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <future>
```

```
6  #include <mutex>
7
8  constexpr auto tenMill = 10000000;
9
10 std::mutex myMutex;
11
12 class MySingleton{
13 public:
14   static MySingleton& getInstance(){
15     std::lock_guard<std::mutex> myLock(myMutex);
16     if (!instance){
17         instance= new MySingleton();
18     }
19     volatile int dummy{};
20     return *instance;
21   }
22 private:
23   MySingleton() = default;
24   ~MySingleton() = default;
25   MySingleton(const MySingleton&) = delete;
26   MySingleton& operator=(const MySingleton&) = delete;
27
28   static MySingleton* instance;
29 };
30
31 MySingleton* MySingleton::instance = nullptr;
```

Читатель может догадаться, что эта реализация работает довольно медленно.



Объект-одиночка с блокировщиком в системе Linux



Объект-одиночка с блокировщиком в системе Linux

Следующая версия потокобезопасного объекта-одиночки также основывается на средствах стандартной библиотеки – в ней используется функция `std::call_once` в сочетании с флагом `std::once_flag`.

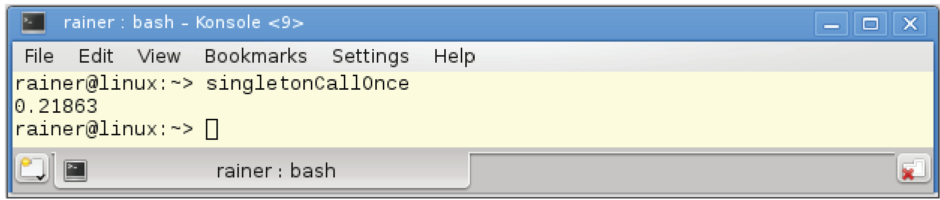
6.2.5. Реализация на основе функции std::call_once

С помощью функции `std::call_once` вместе с флагом `std::once_flag` можно регистрировать вызываемый объект, чтобы в нужный момент он был вызван ровно один раз потокобезопасным образом.

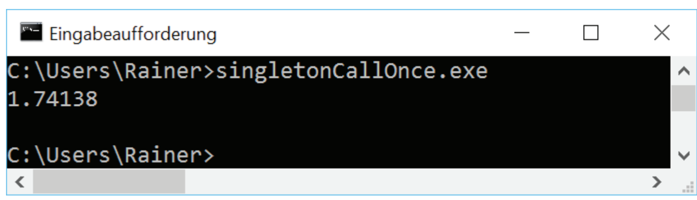
Реализация на основе функции std::call_once

```
1  // singletonCallOnce.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <future>
6  #include <mutex>
7  #include <thread>
8
9  constexpr auto tenMill = 10000000;
10
11 class MySingleton{
12 public:
13   static MySingleton& getInstance(){
14     std::call_once(initInstanceFlag, &MySingleton::initSingleton);
15     volatile int dummy{};
16     return *instance;
17   }
18 private:
19   MySingleton() = default;
20   ~MySingleton() = default;
21   MySingleton(const MySingleton&) = delete;
22   MySingleton& operator=(const MySingleton&) = delete;
23
24   static MySingleton* instance;
25   static std::once_flag initInstanceFlag;
26
27   static void initSingleton() {
28     instance= new MySingleton;
29   }
30 };
31
32 MySingleton* MySingleton::instance = nullptr;
33 std::once_flag MySingleton::initInstanceFlag;
```

Результаты работы программы показаны ниже.



Реализация на основе функции `std::call_once` в системе Linux



Реализация на основе функции `std::call_once` в системе Windows

Разбор различных реализаций объекта-одиночки продолжим решением, основанным на атомарных переменных.

6.2.6. Решение на основе атомарных переменных

Использование атомарных переменных делает задачу значительно более сложной. Так, для атомарных переменных можно даже задать порядок доступа к памяти. Следующие две реализации потокобезопасного объекта-одиночки основываются на разобранном ранее шаблоне блокировки с двойной проверкой.

6.2.6.1. Семантика последовательной согласованности

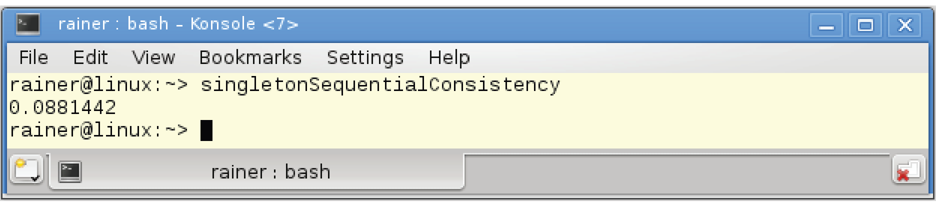
В первой версии решения с атомарными переменными воспользуемся атомарными операциями, не задавая в явном виде порядок доступа к памяти. Это означает, что по умолчанию будет использоваться семантика последовательной согласованности.

Решение на атомарных переменных с последовательной согласованностью

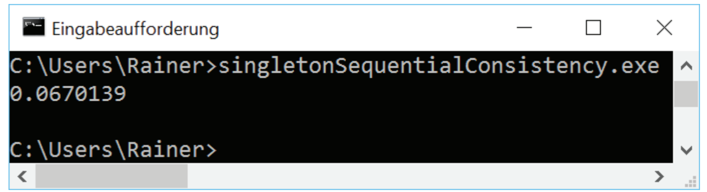
```
1  // singletonSequentialConsistency.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <future>
6  #include <mutex>
7  #include <thread>
8
```

```
9  constexpr auto tenMill = 10000000;
10
11 class MySingleton{
12 public:
13   static MySingleton* getInstance() {
14     MySingleton* sin = instance.load();
15     if (!sin) {
16       std::lock_guard<std::mutex> myLock(myMutex);
17       sin = instance.load(std::memory_order_relaxed);
18       if(!sin){
19         sin= new MySingleton();
20         instance.store(sin);
21       }
22     }
23     volatile int dummy{};
24     return sin;
25   }
26 private:
27   MySingleton() = default;
28   ~MySingleton() = default;
29   MySingleton(const MySingleton&) = delete;
30   MySingleton& operator=(const MySingleton&) = delete;
31
32   static std::atomic<MySingleton*> instance;
33   static std::mutex myMutex;
34 };
35
36 std::atomic<MySingleton*> MySingleton::instance;
37 std::mutex MySingleton::myMutex;
```

В отличие от классической блокировки с двойной проверкой, эта реализация действительно гарантирует, что операция присваивания в строке 19 выполняется раньше, чем функция `store` в строке 20. Эта гарантия следует из последовательной согласованности атомарных операций как порядка доступа к памяти по умолчанию. Нужно обратить внимание на операцию `load` в строке 17 с ослабленным порядком `std::memory_order_relaxed`. Эта операция нужна потому, что между первой операцией `load` в строке 14 и входом в критическую секцию в строке 16 какой-то другой поток может вклиниться и изменить значение переменной `instance`.



Решение на атомарных переменных в системе Linux



Решение на атомарных переменных в системе Windows

Эту программу можно оптимизировать ещё сильнее.

6.2.6.2. Семантика захвата и освобождения

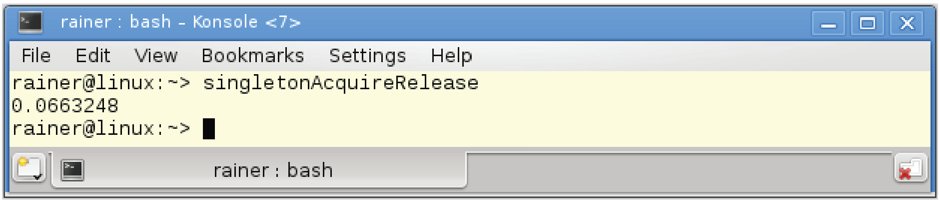
Рассмотрим пристальнее последний пример потокобезопасной реализации объекта-одиночки на основе атомарных переменных. Чтение значения переменной в строке 14 представляет собой операцию захвата, а запись этой переменной в строке 20 – операцию освобождения. Обе операции работают с одной и той же атомарной переменной. Поэтому семантика последовательной согласованности здесь чрезмерна. Стандарт C++ 11 гарантирует, что операция освобождения синхронизируется с операцией захвата той же самой атомарной переменной, чем устанавливается определённый порядок операций. А именно никакие последующие операции чтения и записи не могут ставиться перед операцией захвата. В этом и состоит минимум гарантий, необходимый для реализации потокобезопасного объекта-одиночки.

Реализация на атомарных переменных с семантикой захвата и освобождения

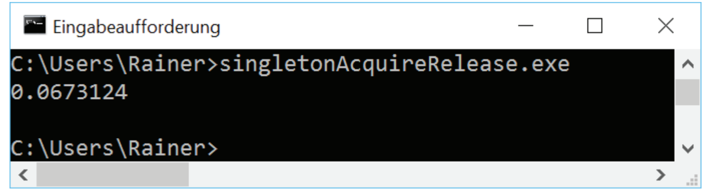
```
1  // singletonAcquireRelease.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <future>
6  #include <mutex>
7  #include <thread>
8
9  constexpr auto tenMill = 10000000;
10
11 class MySingleton{
12 public:
13   static MySingleton* getInstance() {
14     MySingleton* sin = instance.load(std::memory_order_acquire);
15     if (!sin) {
16       std::lock_guard<std::mutex> myLock(myMutex);
17       sin = instance.load(std::memory_order_relaxed);
18       if(!sin) {
19         sin = new MySingleton();
20         instance.store(sin, std::memory_order_release);
21       }
22     }
23     volatile int dummy{};
```

```
24      return sin;
25    }
26  private:
27    MySingleton() = default;
28    ~MySingleton() = default;
29    MySingleton(const MySingleton&) = delete;
30    MySingleton& operator=(const MySingleton&) = delete;
31
32    static std::atomic<MySingleton*> instance;
33    static std::mutex myMutex;
34  };
35
36  std::atomic<MySingleton*> MySingleton::instance;
37  std::mutex MySingleton::myMutex;
```

Производительность реализации, основанной на семантике захвата и освобождения, незначительно отличается от реализации с семантикой последовательной согласованности.



Решение на атомарных переменных в системе Linux



Решение на атомарных переменных в системе Windows

Близость показателей не должна удивлять, поскольку в архитектуре x86 эти два порядка доступа к памяти работают весьма сходным образом. Более существенного выигрыша производительности можно было бы ожидать на архитектурах ARMv7[1] или PowerPC[2]. Более подробное изложение этих вопросов можно найти в блоге Джеффа Прешингса[3].

В завершение темы остаётся лишь сравнить все полученные результаты.

[1] https://en.wikipedia.org/wiki/ARM_architecture.

[2] https://en.wikipedia.org/wiki/PowerPC.

[3] http://preshing.com/.

6.2.7. Сводные данные

Результаты измерений говорят сами за себя. Реализация Мейерса оказалась самой быстрой. Более того, она ещё наиболее проста в исполнении. Она работает примерно вдвое быстрее реализаций на основе атомарных переменных. Как и ожидалось, синхронизация с помощью блокировщика – самая неповоротливая из всех. Функция `std::call_once` довольно медленно работает в системе Windows.

Операционная система и компилятор	Linux (GCC)	Windows (cl.exe)
Однопоточная	0,03	0,02
Реализация Мейерса	0,04	0,03
std::lock_guard	12,47	15,48
std::call_once	0,22	1,74
Последовательная согласованность	0,09	0,07
Захват и освобождение	0,07	0,07

Следует подчеркнуть, что представленные в таблице числа – это сумма длительностей выполнения по всем четырём потокам. Это значит, что вариант Мейерса лучше всего работает в многопоточной среде, так как он демонстрирует почти такую же скорость, как и однопоточная реализация.

6.3. Поэтапная оптимизация с использованием инструмента CppMem

В этом разделе начнём с небольшой программы и будем последовательно улучшать её. Каждый этап её совершенствования будем проверять на анализаторе CppMem. CppMem[1] представляет собой интерактивный инструмент для исследования того, как небольшие участки кода работают в модели памяти C++.

Ниже представлен начальный вариант программы.

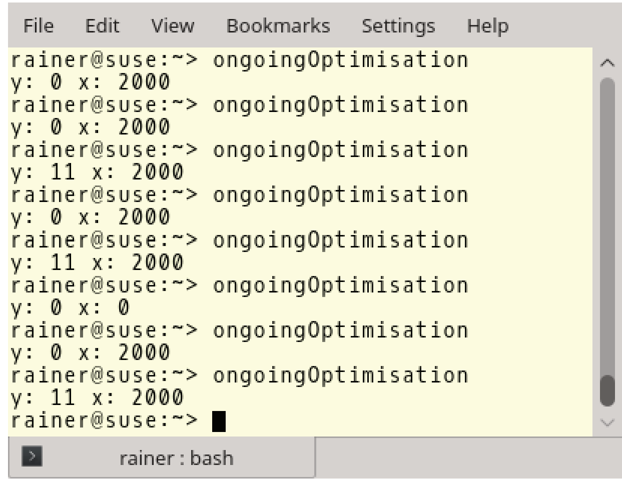
Начальная версия программы для последующей оптимизации

```
1  // ongoingOptimisation.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  int x = 0;
7  int y = 0;
```

[1] http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/.

```
8
9  void writing() {
10   x = 2000;
11   y = 11;
12 }
13
14 void reading() {
15   std::cout << “y: “ << y << “ “;
16   std::cout << “x: “ << x << std::endl;
17 }
18
19 int main() {
20   std::thread thread1(writing);
21   std::thread thread2(reading);
22   thread1.join();
23   thread2.join();
24 }
```

Эта программа устроена весьма просто. Она состоит из двух потоков, `thread1` и `thread2`. Поток `thread1` присваивает значения переменным `x` и `y`, а поток `thread1` вычитывает эти значения в *обратном порядке*. Принцип работы программы кажется очевидным, но даже столь простая программа может давать три различных результата, как свидетельствует следующий рисунок.



Базовый вариант программы

Приступая к поэтапной оптимизации этой программы, нужно иметь в виду два вопроса:

1. Обладает ли эта программа хорошо определённым поведением? В частности, присутствует ли в ней гонка данных?
2. Какие значения могут принимать переменные `x` и `y`?

На первый из этих вопросов обычно бывает сложно ответить. В первую очередь подумаем над этим вопросом, а затем проверим свои рассуждения с помощью инструмента CppMem. Когда ответ на первый вопрос будет найден, из него сам собой получится ответ и на второй вопрос. Возможные значения переменных `x` и `y` можно будет представить в виде таблицы.

Между прочим, следует ещё пояснить, что здесь имеется в виду под пошаговой оптимизацией. Это довольно просто: нам предстоит оптимизировать программу, с каждым разом всё более ослабляя порядок доступа к памяти. Полный список шагов по оптимизации программы будет таким:

1) неатомарные переменные;
2) синхронизация на основе блокировщика;
3) атомарные переменные с последовательной согласованностью;
4) атомарные переменные с семантикой захвата и освобождения;
5) атомарные переменные с ослабленной семантикой;
6) переменные с квалификатором `volatile`.

Прежде чем начать поэтапную оптимизацию программы, нужно получить хотя бы начальное представление об инструменте CppMem. Глава 15 содержит такое введение.

6.3.1. Неатомарные переменные

Одного нажатия на кнопку **Run** в окне CppMem достаточно, чтобы убедиться в наличии гонки данных. Если точнее, в программе есть даже две гонки данных. Не защищен доступ ни к переменной `x`, ни к переменной `y`. Как следствие программа в целом обладает неопределённым поведением. О такой программе можно сказать, что она обладает семантикой русской рулетки: в результате её работы может произойти что угодно, даже самоуничтожение компьютера. Поэтому мы не можем строить никаких предположений о значениях переменных `x` и `y`.

Гарантии для целочисленных переменных

В большинстве распространённых архитектур гарантируется атомарность доступа к переменным типа `int`, при условии что эти переменные выровнены в памяти естественным образом. Естественным в 32-битных или 64-битных архитектурах называется выравнивание, при котором адрес переменной типа `int` кратен четырём. Причина, по которой есть смысл так подробно останавливаться на этом правиле, состоит в том, что стандарт C++ 11 позволяет программисту в явном виде управлять выравниванием данных.

Конечно, это вовсе не означает совета использовать обычный тип `int` вместо атомарного. Это означает лишь, что компилятор предоставляет более сильные гарантии, чем стандарт языка C++. Если программист строит свою программу с опорой на гарантии компилятора, программа не будет отвечать стандарту языка C++ 11. Следовательно, такая программа может работать некорректно на иных архитектурах или на той же архитектуре с будущей версией компилятора.

Посмотрим теперь, как инструмент CppMem сообщает о неопределённом поведении программы. Инструмент CppMem позволяет сократить текст программы до полного минимума.

Несинхронизировнный доступ в анализаторе CppMem

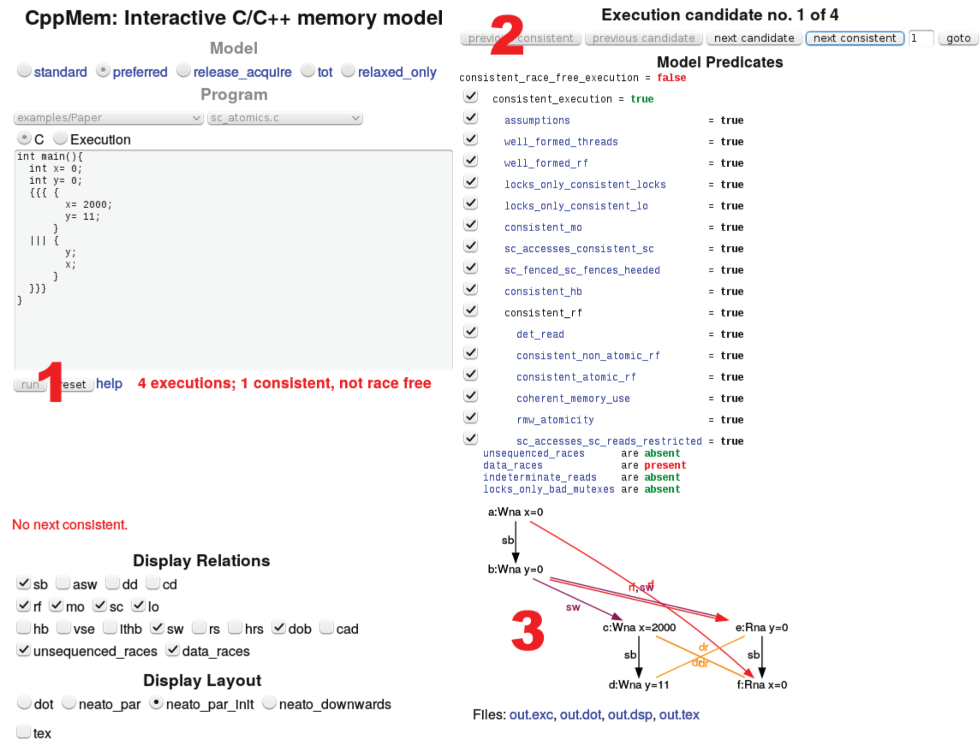
```
1  int main() {
2    int x = 0;
3    int y = 0;
4    {{{ {
5          x = 2000;
6          y = 11;
7        }
8    ||| {
9          y;
10         x;
11       }
12   }}}
13 }
```

Потоки на встроенном языке CppMem можно обозначить посредством фигурных скобок (строки 4 и 12) и символа трубопровода (строка 8). Дополнительные пары фигурных скобок, как в строках 4 и 7 или в строках 8 и 11, служат для обозначения действий, которые должны выполняться потоком. Поскольку вывод значений переменных x и y нам на самом деле не нужен, в строках 9 и 10 стоит лишь вычитание их значений.

Это было отступление от теории работы с инструментом CppMem. Пора переходить к практике.

6.3.1.1. Анализ программы

Если запустить программу в среде CppMem, среда покажет сообщение, помеченное на рисунке цифрой 1, о том, что из четырёх возможных последовательностей выполнения программы лишь одно оказалось согласованным, но и оно содержит гонку данных. Теперь пользователь может переключаться между вариантами выполнения программы с помощью кнопок, помеченных цифрой 2, и анализировать графическую схему процесса выполнения, снабжённую текстовыми метками, на рисунке она обозначена цифрой 3.

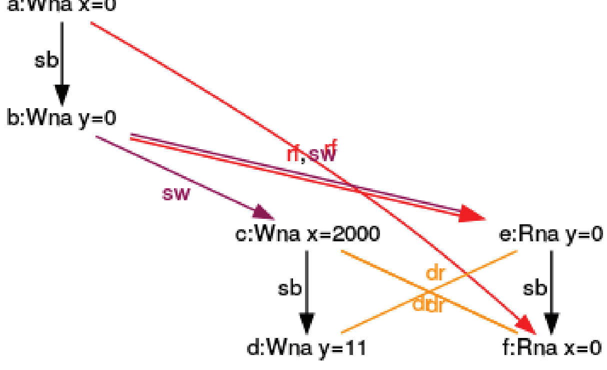


Гонка данных в случае неатомарных переменных

Больше всего информации о программе можно получить именно из анализа этих графов.

6.3.1.1.1. Первый вариант выполнения

На какие выводы может натолкнуть граф выполнения программы, показанный на следующем рисунке?



Первый вариант выполнения

Вершины графа соответствуют выражениям в тексте программы, а рёбра – отношениям между выражениями. Выражения помечены на графе буквами от a до f, они и будут использоваться в следующем далее пояснении. Что же должен говорить пользователю этот граф?

- a:Wna x = 0: выполнение программы начинается с неатомарной записи значения в переменную x;
- sb (sequenced-before – находится перед): операция записи (a) выполняется перед операцией записи (b). Подобное же отношение имеет место между операциями (c) и (d), а также между операциями (e) и (f);
- rf (read from – чтение из): выражение (e) читает значение переменной y, присвоенное выражением (b); подобным же образом выражение (f) читает значение, присвоенное выражением (a);
- sw (synchronizes-with – синхронизируется с): выражение (a) синхронизируется с выражением (f). Это утверждение справедливо потому, что выражение (f) выполняется в отдельном потоке, а момент создания потока представляет собой точку синхронизации: результаты всех операций, которые произошли до этой точки, должны быть видимы из потока. Из соображений симметрии такое же отношение имеет место и между выражениями (b) и (e);
- dr (data race – гонка данных): здесь имеют место гонки данных между попытками чтения и записи как для переменной x, так и для переменной y. Следовательно, программа обладает неопределённым поведением.

Почему это выполнение считается согласованным

Разобранный здесь вариант выполнения программы считается согласованным потому, что значения переменных x и y получили свои начальные значения в главном потоке, в выражениях (a) и (b). Присваивание этим переменным новых значений в выражениях (c) и (d) не является согласованным с точки зрения модели памяти[1].

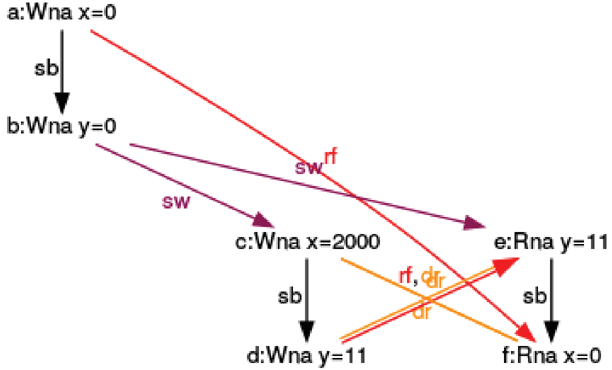
Следующие три варианта выполнения программы не являются согласованными.

6.3.1.1.2. Второй вариант выполнения

Последовательность выполнения операций показана на рисунке на следующей странице.

---

[1] Здесь имеется в виду, что, несмотря на явную гонку данных, выполнение программы сложилось таким образом, что поток thread2 успел вычитать вполне установленные главной функцией значения переменных до того, как поток thread1 начал менять их значения, поэтому результат работы такой последовательности операций вполне определён, т. е. выполнение программы носит согласованный характер. В то же время эта согласованность стала результатом не гарантии, а удачного стечения обстоятельств: того, что поток thread2 обошёл в гонке поток thread1. – *Прим. перев.*

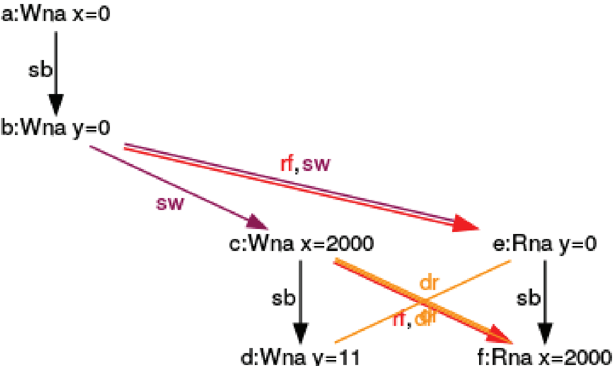


Второй вариант выполнения

Теперь выражение (e) читает значение, которое пишет выражение (d). Операция записи (d) происходит параллельно с операцией чтения (e).

6.3.1.1.3. Третий вариант выполнения

Ещё один возможный сценарий выполнения параллельной программы представлен на рисунке.

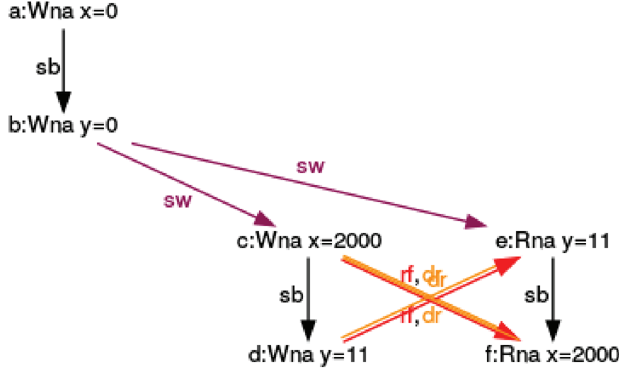


Третий вариант выполнения

Он симметричен предыдущему варианту: операция чтения в выражении (f) выполняется одновременно с записью в выражении (c).

6.3.1.1.4. Четвёртый вариант выполнения

В последнем из логически возможных сценариев неправильно всё. Обе операции чтения, (e) и (f), выполняются одновременно с операциями записи в те же переменные: (d) и (c).



Четвёртый вариант выполнения

6.3.1.1.5. Выводы

Даже с конфигурацией по умолчанию инструмент CppMem позволил получить много ценной информации и глубоко заглянуть в процесс выполнения программы. Построенные системой графики помогли выявить следующее:

- в результате работы программы могут возникнуть четыре комбинации значений переменных: (0, 0), (11, 0), (0, 2000) и (11, 2000);
- в программе имеется по меньшей мере одна гонка данных – следовательно, её поведение не определено;
- лишь один из четырёх возможных вариантов выполнения согласован.

Использование квалификатора `volatile`

С точки зрения модели памяти доступ к переменным, объявленным с квалификатором `volatile`, ничем не отличается от несинхронизированного доступа к обычным переменным x и y.

Несинхронизированный доступ к `volatile` -переменным

```
1  int main() {
2   volatile int x = 0;
3   volatile int y = 0;
4   {{{ {
5       x = 2000;
6       y = 11;
7     }
8   ||| {
9       y;
10      x;
11    }
12  }}}
13 }
```

Для этой программы инструмент CppMem генерирует точно такие же граф-схемы, как для предыдущего примера. Причина этого проста и состоит в том, что семантика квалификатора `volatile` никак не связана с параллельным режимом работы программы.

В рассмотренном выше примере доступ к переменным x и y никак не был синхронизирован, отсюда и гонка данных, а с нею и неопределённое поведение. Наиболее очевидный способ синхронизации – использование блокировщиков.

6.3.2. Анализ программы с блокировкой

Пусть теперь оба потока, thread1 и thread2, используют один и тот же мьютекс, заворачивая его в блокировщик std::lock_guard.

Поэтапная оптимизация: использование блокировщика

```
1  // ongoingOptimisationLock.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6
7  int x = 0;
8  int y = 0;
9
10 std::mutex mut;
11
12 void writing(){
13   std::lock_guard<std::mutex> guard(mut);
14   x = 2000;
15   y = 11;
16 }
17
18 void reading(){
19   std::lock_guard<std::mutex> guard(mut);
20   std::cout << "y: " << y << " ";
21   std::cout << "x: " << x << std::endl;
22 }
23
24 int main(){
25   std::thread thread1(writing);
26   std::thread thread2(reading);
27   thread1.join();
28   thread2.join();
29 };
```

Поведение этой программы вполне определено. В зависимости от порядка выполнения потоков (сначала поток thread1, затем thread2, или наоборот) либо сначала из обеих переменных вычитаются их первоначальные значения, затем обеим присваиваются новые, либо сначала обеим присваиваются новые значения, после чего они вычитаются. Из четырёх мыслимых комбинаций значений переменных лишь две возможны, как показано в таблице.

Возможные значения переменных при использовании блокировщика

y	x	Возможность
0	0	Да
11	0	Нет
0	2000	Нет
11	2000	Да

Блокировщики и инструмент CppMem

Автору не удалось применить анализатор CppMem к программе, содержащей блокировщик. Если читателю это удастся, автор будет признателен за информацию.

Блокировщики просты в использовании, однако такая синхронизация выходит слишком тяжеловесной. Рассмотрим более проворные механизмы синхронизации, основанные на атомарных переменных.

6.3.3. Атомарные переменные с последовательной согласованностью

Если программист не указывает явно порядок доступа к памяти, по умолчанию применяется семантика последовательной согласованности. Последовательная согласованность влечёт за собой две гарантии. Во-первых, в каждом потоке операции над атомарными переменными выполняются в том порядке, в котором они записаны в исходном коде. Во-вторых, операции всех потоков образуют глобально упорядоченную последовательность. Ниже представлена оптимизированная версия предыдущей программы.

Поэтапная оптимизация: последовательная согласованность

```
1  // ongoingOptimisationSequentialConsistency.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6
7  std::atomic<int> x{0};
8  std::atomic<int> y{0};
9
10 void writing(){
11   x.store(2000);
12   y.store(11);
13 }
14
15 void reading(){
16   std::cout << y.load() << " ";
17   std::cout << x.load() << std::endl;
18 }
19
```

```
20 int main(){
21   std::thread thread1(writing);
22   std::thread thread2(reading);
23   thread1.join();
24   thread2.join();
25 };
```

Рассмотрим эту программу подробнее. Она не содержит гонки данных, так как переменные x и y атомарны. Поэтому остаётся ответить лишь на один вопрос: какие значения переменных возможны. Благодаря последовательной согласованности все потоки должны образовывать единый глобальный порядок выполнения операций. Очевидны следующие соотношения.

- Операция `store`, записывающая в переменную x значение 2000, выполняется ранее операции `store`, которая записывает значение 11 в переменную y.
- Операция `load` над переменной y выполняется ранее операции `load` над переменной x.

Следовательно, вычитание значения переменной x не может дать значение 0, если из переменной y уже прочитано значение 11.

Все остальные комбинации значений возможны. Ниже перечислены три возможных сценария перемежающегося выполнения потоков, приводящих к трём различным комбинациям значений переменных.

1. Поток `thread1` целиком выполняется до потока `thread2`.
2. Поток `thread2` целиком выполняется до потока `thread1`.
3. В потоке `thread1` выполняется первая операция `store`, записывающая в переменную x значение 2000, затем начинается выполнение потока `thread2`.

Возможные комбинации значений переменных показаны в таблице.

Возможные значения переменных в семантике последовательной согласованности

y	x	Возможность
0	0	Да
11	0	Нет
0	2000	Да
11	2000	Да

Сравним наши рассуждения с тем, что даст анализатор CppMem.

6.3.3.1. Анализ программы инструментом CppMem

Ниже представлен перевод последней программы на язык системы CppMem.

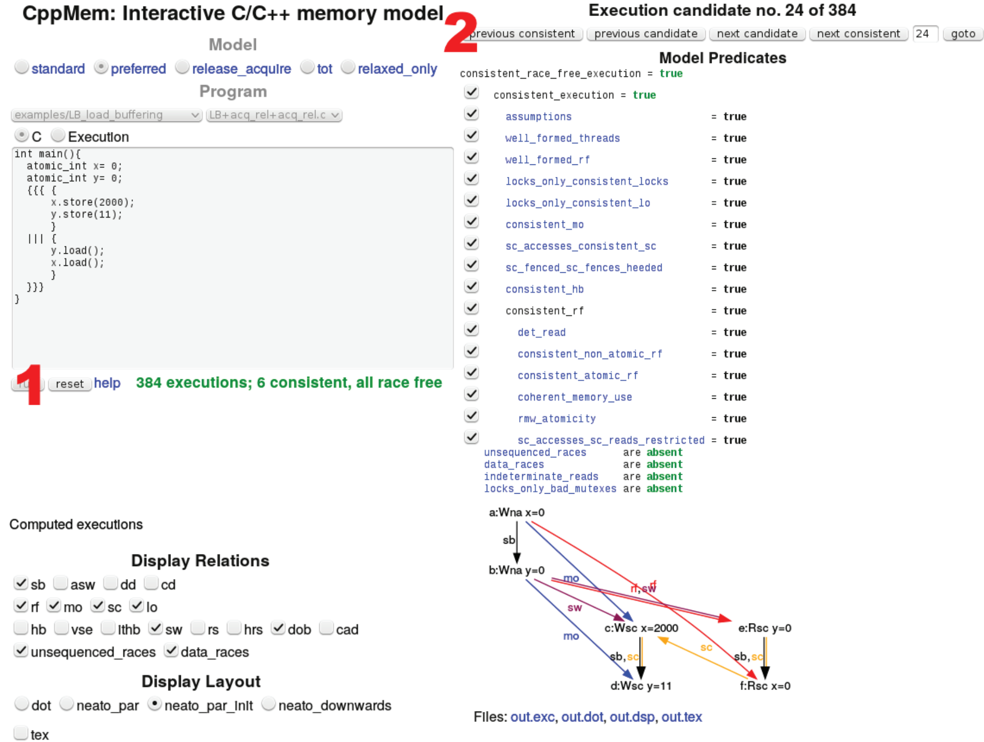
Атомарные переменные в системе CppMem

```
1 int main() {
2   atomic_int x = 0;
3   atomic_int y = 0;
```

```
4    {{{ {
5          x.store(2000);
6          y.store(11);
7        }
8    ||| {
9          y.load();
10         x.load();
11       }
12   }}}
13 }
```

Прежде всего нужно сказать несколько слов о синтаксисе. В строках 2 и 3 используется тип `atomic_int` – это предопределённый в системе CppMem псевдоним для типа `std::atomic<int>`.

Если теперь запустить программу под управлением анализатора, число получившихся вариантов-кандидатов оказывается удивительно большим.



Поэтапная оптимизация: последовательная согласованность

Система обнаруживает 384 возможных кандидата (цифра 1 на рисунке), только 6 из которых согласованы. Ни один вариант-кандидат не содержит гонки данных. Нас будут интересовать только шесть согласованных вариантов выполнения программы, на остальные 378 вариантов можно не обращать внимания. Несогласованность означает, что они, например, не отве-

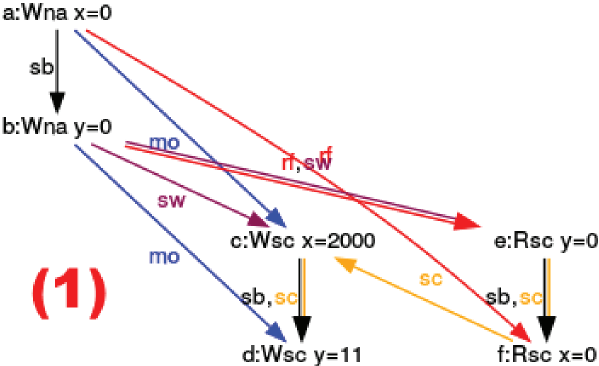
чают ограничениям на порядок модификации переменных, накладываемым моделью памяти.

Воспользуемся панелью визуального интерфейса, помеченной на рисунке цифрой 2, чтобы исследовать шесть размеченных схем.

Как мы уже знаем, возможны все комбинации значений переменных, кроме одной: когда с `y=11` и `x=0`. Три оставшихся результата возможны в силу семантики последовательной согласованности. Теперь хочется выяснить, какие последовательности перемежающихся операций ведут к тем или иным возможным комбинациям значений переменных.

6.3.3.1.1. Вариант выполнения для y = 0, x = 0

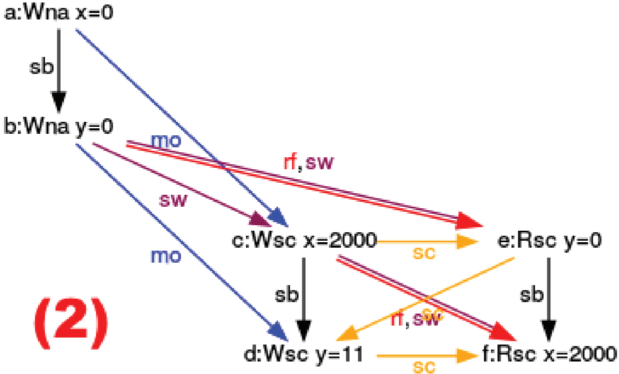
Эта комбинация значений реализуется при единственном варианте выполнения программы, показанном на следующем рисунке.



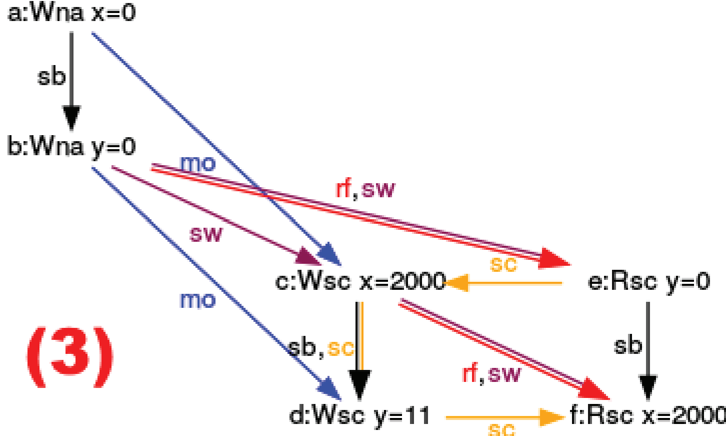
Вариант выполнения для y = 0, x = 0

6.3.3.1.2. Варианты выполнения для y = 0, x = 2000

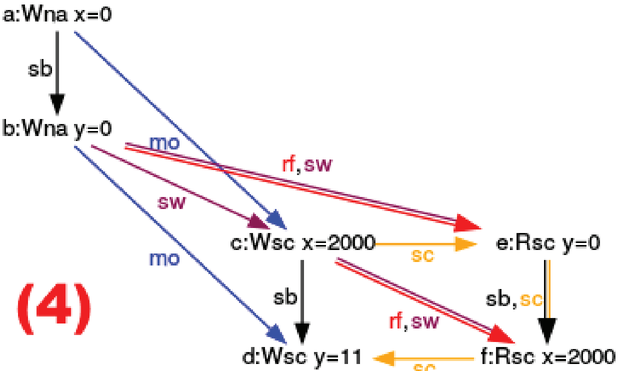
Четыре различных варианта выполнения программы, показанных на рисунках, ведут к одному результату.



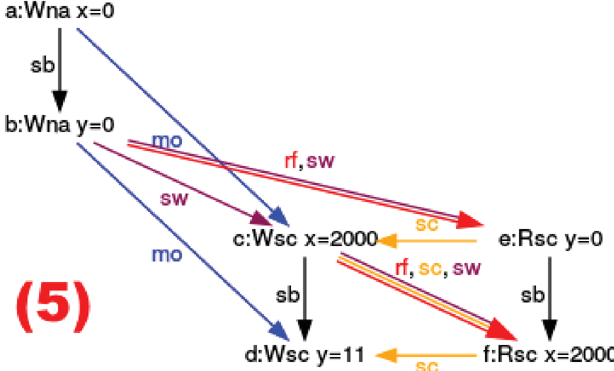
Вариант выполнения для y = 0, x = 2000



Вариант выполнения для y = 0, x = 2000



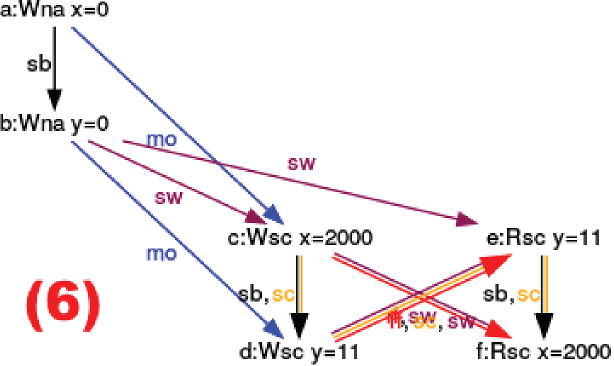
Вариант выполнения для y = 0, x = 2000



Вариант выполнения для y = 0, x = 2000

6.3.3.1.3. Вариант выполнения для y = 11, x = 2000

Последняя возможная комбинация значений переменных реализуется лишь в одном варианте выполнения программы.

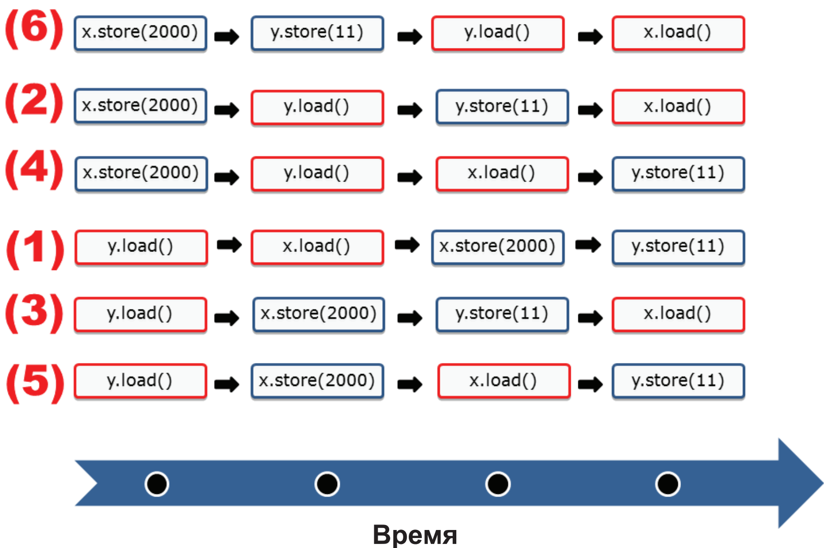


Вариант выполнения для y = 11, x = 2000

Анализ программы этим не исчерпывается. Нужно ответить ещё на один вопрос: какая последовательность выполнения операций соответствует каждому из шести графов?

6.3.3.2. Последовательность операций

На следующем рисунке каждой возможной последовательности операций поставлен в соответствие один из показанных выше графов.



Последовательности выполнения операций

Начнём с наиболее очевидных случаев.

- Вариант (1) довольно просто сопоставить с графом под тем же номером. При такой последовательности операций из обеих переменных вычитаются начальные значения 0, поскольку обе операции чтения происходят раньше, чем в переменные попадают новые значения 2000 и 11.
- Вариант (6) можно связать с соответствующим графом с помощью похожего рассуждения. Из переменной `y` вычитается значение 11, а из переменной `x` – значение 2000, поскольку обе операции чтения выполняются после операций записи.
- Варианты (2), (3), (4), (5) более интересны, в результате их из переменной `y` вычитается значение 0, а из переменной `x` – значение 2000. Желтые стрелки с меткой (sc) на графах дают ключ к пониманию происходящего, так как обозначают последовательность выполнения операций. Рассмотрим, например, вариант (2):
  – путь по жёлтым стрелкам (sc) на графе (2) означает такую последовательность операций: записать значение 2000 в переменную `x`, прочитать значение 0 из переменной `y`, затем записать значение 11 в переменную `y` и, наконец, прочитать из переменной `x` значение 2000. Этот путь на графе соответствует перемежающейся последовательности операций под номером (2) на последнем рисунке.

Посмотрим, что будет, если нарушить последовательную согласованность семантикой захвата и освобождения.

6.3.4. Атомарные переменные с семантикой захвата и освобождения

В семантике захвата и освобождения синхронизируются между собой только атомарные операции над одной и той же переменной. В этом состоит её отличие от семантики последовательной согласованности, при которой синхронизируются все атомарные операции между всеми потоками. Это отличие делает семантику захвата и освобождения более лёгкой для компьютера и, следовательно, быстрой. Текст модифицированной программы с семантикой захвата и освобождения приведён ниже.

Поэтапная оптимизация: семантика захвата и освобождения

```
1  // ongoingOptimizationAcquireRelease.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6
7  std::atomic<int> x{0};
8  std::atomic<int> y{0};
9
```

```
10 void writing(){
11   x.store(2000, std::memory_order_relaxed);
12   y.store(11, std::memory_order_release);
13 }
14
15 void reading(){
16   std::cout << y.load(std::memory_order_acquire) << " ";
17   std::cout << x.load(std::memory_order_relaxed) << std::endl;
18 }
19
20 int main(){
21   std::thread thread1(writing);
22   std::thread thread2(reading);
23   thread1.join();
24   thread2.join();
25 };
```

Одного взгляда на эту программу довольно, чтобы убедиться: все операции в ней атомарны – следовательно, поведение программы вполне определено. Однако второй взгляд обнаруживает интересную деталь: если операции над переменной `y` выполняются с флагами `std::memory_order_release` (строка 12) и `std::memory_order_acquire` (строка 16), то операции над переменной `x`, напротив, выполняются в ослабленной семантике `std::memory_order_relaxed` (строки 11 и 17). Таким образом, порядок выполнения операций над переменной `x` не связан никакими ограничениями. Поэтому ответ на вопрос о возможных комбинациях значений переменных `x` и `y` может дать только анализ переменной `y`. Имеют место следующие соотношения:

- операция записи в переменную `y` в строке 12 синхронизируется с операцией чтения из этой переменной в строке 16;
- операция записи в переменную `x` в строке 11 видима ранее операции записи в переменную `y` в строке 12;
- операция чтения из переменной `y` в строке 16 видима ранее операции чтения из переменной `x` в строке 17.

Остановимся подробнее на этих трёх утверждениях. Ключевое значение имеет то обстоятельство, что запись в переменную `y` в строке 12 синхронизируется с чтением из неё в строке 16. Это справедливо потому, что обе операции работают с одной и той же атомарной переменной и используют при этом семантику захвата и освобождения: запись работает как освобождение, а чтение – как захват. Парные операции над переменной `y` обладают ещё одним важным свойством. Они делают эту переменную своеобразным барьером, относительно которого распределяются другие операции. Так, операция записи значения 2000 в переменную `x` не может быть выполнена после операции записи в переменную `y`, а операция чтения переменной `x` не может выполниться ранее чтения переменной `y`.

Семантика захвата и освобождения требует более сложных рассуждений, чем ранее рассмотренная семантика последовательной согласованности, но комбинации возможных значений переменных `x` и `y` остаются без измене-

ний. Невозможной по-прежнему оказывается только ситуация, когда переменная y имеет значение 11, а переменная x – значение 0.

Возможны три разных порядка выполнения операций, которые приводят к трём комбинациям значений переменных:

- поток thread1 полностью выполняется до начала потока thread2;
- поток thread2 полностью выполняется до начала потока thread1;
- поток thread1 выполняется до операции записи в строке 11 до того, как начнёт выполняться поток thread2.

Иными словами, имеем комбинации значений переменных, показанные в следующей таблице.

Возможные значения переменных в семантике захвата и освобождения

y	x	Возможность
0	0	Да
11	0	Нет
0	2000	Да
11	2000	Да

Попробуем теперь проверить наши умозаключения с помощью инструмента CppMem.

6.3.4.1. Анализ программы инструментом CppMem

Ниже показана соответствующая программа на внутреннем языке CppMem.

Семантика захвата и освобождения в системе CppMem

```
1  int main() {
2    atomic_int x = 0;
3    atomic_int y = 0;
4    {{{ {
5          x.store(2000, memory_order_relaxed);
6          y.store(11, memory_order_release);
7       }
8    ||| {
9          y.load(memory_order_acquire);
10         x.load(memory_order_relaxed);
11      }
12   }}}
13 }
```

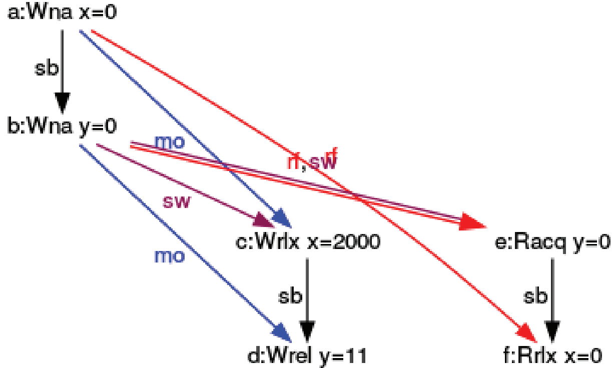
Мы уже знаем, что в результате выполнения этой программы возможны все исходы, кроме одного (y = 11, x = 0).

6.3.4.1.1. Возможные варианты выполнения

Рассматриваем здесь лишь три графа с согласованным выполнением. На всех схемах показано отношение захвата-освобождения между операцией записи

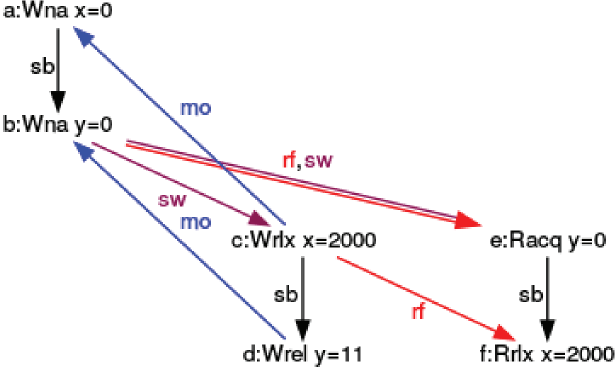
в переменную y (выступающей как освобождение) и операцией чтения из неё (работающей как захват). При этом не важно, выполняется ли чтение переменной y в главном потоке программы или в отдельном потоке. Соответствующие стрелки снабжены меткой rf. Кроме того, отношение синхронизации показано стрелками с меткой sw.

6.3.4.1.2. Вариант выполнения для случая y = 0, x = 0



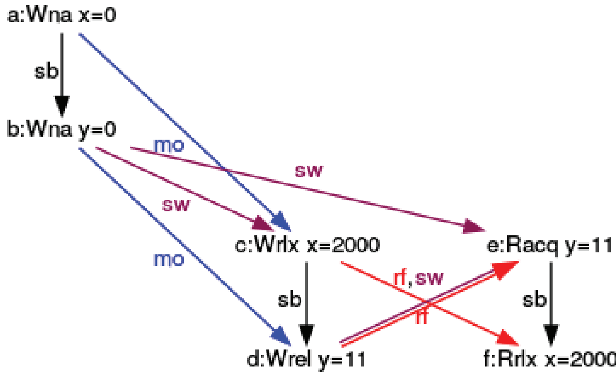
Вариант выполнения для случая y = 0, x = 0

6.3.4.1.3. Вариант выполнения для случая y = 0, x = 2000



Вариант выполнения для случая y = 0, x = 2000

6.3.4.1.4. Вариант выполнения для случая y = 11, x = 2000



Вариант выполнения для случая y = 11, x = 2000

Можно было бы предположить, что переменную x вообще не нужно делать атомарной. Это кажущееся очевидным предположение оказывается неверным. Разберёмся почему.

6.3.5. Смесь атомарных и неатомарных переменных

Типичная ошибка в понимании семантики захвата и освобождения – думать, будто операция захвата ждёт соответствующего освобождения. Из этого ложного предположения легко сделать столь же ложный вывод, что программу можно оптимизировать, сделав переменную x неатомарной. Рассмотрим следующую программу.

Поэтапная оптимизация: смесь атомарных и неатомарных переменных

```
1   // ongoingOptimisationAcquireReleaseBroken.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6
7   int x = 0;
8   std::atomic<int> y{0};
9
10  void writing(){
```

```
11   x= 2000;
12   y.store(11, std::memory_order_release);
13 }
14
15 void reading() {
16   std::cout << y.load(std::memory_order_acquire) << " ";
17   std::cout << x << std::endl;
18 }
19
20 int main(){
21   std::thread thread1(writing);
22   std::thread thread2(reading);
23   thread1.join();
24   thread2.join();
25 };
```

В этой программе имеется гонка данных по переменной x и, следовательно, её поведение не определено. Семантика захвата и освобождения гарантирует, что если запись в переменную y в строке 12 выполняется ранее операции чтения из этой переменной в строке 16, то присваивание нового значения переменной x в строке 11 выполнится до чтения из этой переменной в строке 17. Однако если это условие не выполнено, чтение и запись переменной x могут произойти одновременно. Итак, имеем одновременный доступ к общей переменной, и одна из операций при этом представляет собой запись. Это и есть, по определению, гонка данных.

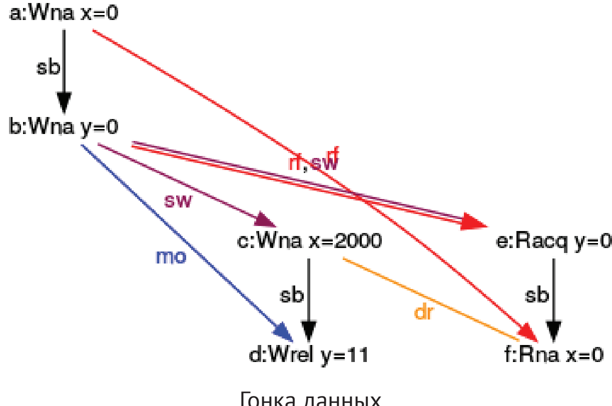
Чтобы отчётливее понять поведение программы, воспользуемся инструментом CppMem.

6.3.5.1. Анализ программы инструментом CppMem

Смешивание атомарных и неатомарных переменных в системе CppMem

```
1  int main() {
2    int x = 0;
3    atomic_int y = 0;
4    {{{ {
5          X = 2000;
6          y.store(11, memory_order_release);
7        }
8    ||| {
9          y.load(memory_order_acquire);
10         x;
11       }
12   }}}
13 }
```

Гонка данных возникает, когда один поток выполняет присваивание `x = 2000`, в то время как другой поток читает значение переменной x. На схеме это изображено жёлтой стрелкой с меткой dr (data race – «гонка данных»).



Гонка данных

Процессу пошаговой оптимизации программы не хватает последнего шага – ослабленной семантики атомарных операций над обеими переменными.

6.3.6. Атомарные переменные с ослабленной семантикой

При ослабленной семантике не происходит никакой синхронизации или упорядочивания операций – гарантируется лишь атомарность каждой из них.

Поэтапная оптимизация: ослабленная семантика

```
1  // ongoingOptimisationRelaxedSemantic.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6
7  std::atomic<int> x{0};
8  std::atomic<int> y{0};
9
10 void writing() {
11   x.store(2000, std::memory_order_relaxed);
12   y.store(11, std::memory_order_relaxed);
13 }
14
15 void reading() {
16   std::cout << y.load(std::memory_order_relaxed) << " ";
17   std::cout << x.load(std::memory_order_relaxed) << std::endl;
18 }
19
20 int main() {
21   std::thread thread1(writing);
```

```
22   std::thread thread2(reading);
23   thread1.join();
24   thread2.join();
25 };
```

Как и в предыдущих случаях, нужно найти ответы на два основных вопроса:

1. Обладает ли программа вполне определённым поведением?
2. Какие комбинации значений переменных возможны в результате выполнения программы?

В ослабленной семантике ответ на два основных вопроса очевиден. Поскольку все операции над обеими переменными атомарны, поведение программы вполне определено. С другой стороны, ослабленная семантика не налагает никаких ограничений на порядок выполнения операций в разных потоках. Это может привести к тому, что поток `thread2` увидит операции записи не в том порядке, в котором они выполняются потоком `thread1`. Впервые в ходе поэтапной оптимизации программы становится возможной ситуация, когда второй поток увидит в переменных `x` и `y` значения 0 и 11 соответственно. Таким образом, возможными становятся все четыре комбинации, как показано в следующей таблице.

Возможные значения переменных в ослабленной семантике

y	x	Возможность
0	0	Да
11	0	Да
0	2000	Да
11	2000	Да

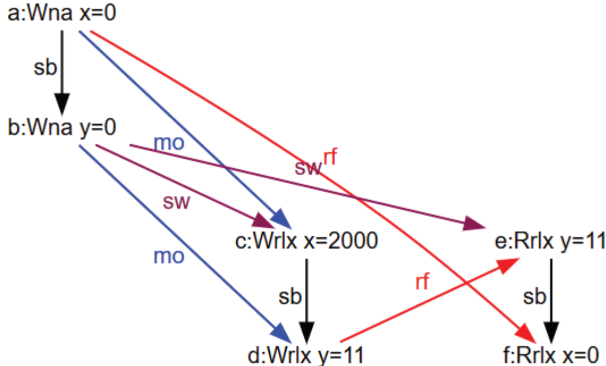
Интересно узнать, как инструмент CppMem изобразит граф выполнения программы для случая (x = 0, y = 11).

6.3.6.1. Анализ инструментом CppMem

Семантика захвата и освобождения в системе CppMem

```
1  int main() {
2    atomic_int x = 0;
3    atomic_int y = 0;
4    {{{ {
5          x.store(2000, memory_order_relaxed);
6          y.store(11, memory_order_relaxed);
7        }
8    ||| {
9          y.load(memory_order_relaxed);
10         x.load(memory_order_relaxed);
11       }
12   }}}
13 }
```

По представленной выше программе на внутреннем языке инструмент CppMem строит следующий граф, описывающий её необычное поведение.



Вариант выполнения для случая (x = 0, y = 11)

На схеме показано, что из переменной x вычитается значение 0, а из переменной y – значение 11, несмотря на то что в другом потоке запись в переменную x располагается перед записью в переменную y.

6.3.7. Итоги

Взять за основу небольшую программу и улучшать её шаг за шагом оказалось довольно поучительным упражнением. Во-первых, с каждым шагом оптимизации всё больше становилось возможных способов чередования операций между потоками. Как следствие у двух переменных появлялось всё больше возможных комбинаций значений. Во-вторых, каждое улучшение делало поведение программы всё более сложным. Даже для такой миниатюрной программы инструмент CppMem выявляет значительный массив результатов.

6.4. Быстрая синхронизация потоков

Если синхронизировать выполнение потоков требуется более одного раза, можно воспользоваться переменными условия, флагом `std::atomic_flag`, атомарным типом `std::atomic<bool>` или семафорами. Цель этого раздела – выяснить, какой из вариантов самый быстрый.

Чтобы получить числовые данные для сравнения, реализуем игру в пинг-понг. В одном потоке будет выполняться функция `ping`, а в другом – функция `pong`. Для простоты будем в дальнейшем называть эти потоки, соответственно, ping-потоком и pong-потоком. Ping-поток ждёт оповещения от pong-потока и, в свою очередь, отсылает оповещение ему. Игра заканчивается после миллиона таких оповещений. Чтобы получить корректные данные, каждый эксперимент будем повторять пять раз.

Об интерпретации числовых данных

Автор выполнял измерение производительности в конце 2020 года с использованием новейшей на тот момент версии 19.28 компилятора из среды Visual Studio, поскольку он уже поддерживал синхронизацию посредством атомарных переменных и семафоров. Примеры компилировались с максимальным уровнем оптимизации (ключ /Ox). Полученные показатели характеризуют лишь относительную производительность методов синхронизации потоков. Абсолютные показатели производительности сильно зависят от платформы. Чтобы получить их, читателю стоит повторить эксперименты на своей системе.

Сравнение механизмов синхронизации начнём с переменных условия.

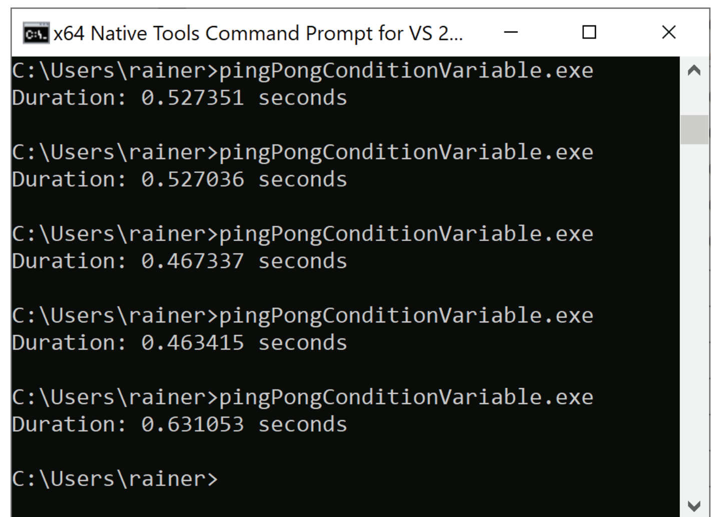
6.4.1. Переменные условия

Многократная синхронизация на основе переменных условия

```
1  // pingPongConditionVariable.cpp
2
3  #include <condition_variable>
4  #include <iostream>
5  #include <atomic>
6  #include <thread>
7
8  bool dataReady{false};
9
10 std::mutex mutex_;
11 std::condition_variable condVar1;
12 std::condition_variable condVar2;
13
14 std::atomic<int> counter{};
15 constexpr int countlimit = 1'000'000;
16
17 void ping() {
18     while(counter <= countlimit) {
19         {
20             std::unique_lock<std::mutex> lck(mutex_);
21             condVar1.wait(lck, []{return dataReady == false;});
22             dataReady = true;
23         }
24         ++counter;
25         condVar2.notify_one();
26   }
27 }
28
29 void pong() {
30     while(counter <= countlimit) {
31         {
32             std::unique_lock<std::mutex> lck(mutex_);
33             condVar2.wait(lck, []{return dataReady == true;});
34             dataReady = false;
35         }
```

```
36         condVar1.notify_one();
37     }
38 }
39
40 int main(){
41     auto start = std::chrono::system_clock::now();
42
43     std::thread t1(ping);
44     std::thread t2(pong);
45
46     t1.join();
47     t2.join();
48
49     std::chrono::duration<double> dur =
50         std::chrono::system_clock::now() - start;
51     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
52 }
```

В этой программе используются две переменные условия: condVar1 и condVar2. Ping-поток ждёт оповещения через переменную condVar1 и посылает через переменную condVar2. Переменная dataReady помогает предотвратить ложное пробуждение и потерю пробуждения. Игра в пинг-понг заканчивается, когда счётчик counter достигает значения countlimit. Вызовы функции-члена notify_one (строки 25 и 36) и операции над переменной counter потокобезопасны и поэтому находятся вне критических секций. Результаты выполнения программы представлены на следующем рисунке.



Многократная синхронизация на основе переменных условия

Среднее время выполнения составляет 0,52 с.

Замена механизма синхронизации с переменных условия на флаг std::atomic_flag вполне очевидна.

6.4.2. Решение на основе атомарного флага

Сначала покажем, как ту же логику функционирования воплотить на двух флагах, затем рассмотрим решение, в котором используется лишь один флаг.

6.4.2.1. Решение с двумя флагами

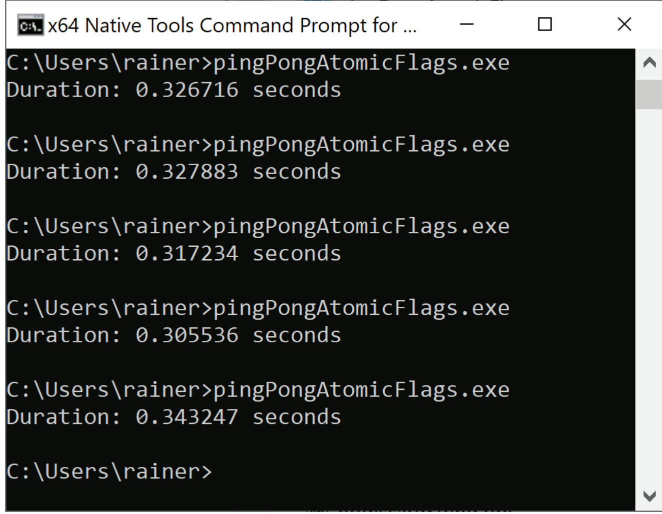
Следующая программа отличается от показанной ранее только тем, что ожидание переменной условия заменено ожиданием атомарного флага, а оповещение через переменную условия заменено, соответственно, установкой флага и последующим оповещением.

Многократная синхронизация с двумя атомарными флагами

```
1  // pingPongAtomicFlags.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag1{};
8  std::atomic_flag condAtomicFlag2{};
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1’000’000;
12
13 void ping() {
14     while(counter <= countlimit) {
15         condAtomicFlag1.wait(false);
16         condAtomicFlag1.clear();
17
18         ++counter;
19
20         condAtomicFlag2.test_and_set();
21         condAtomicFlag2.notify_one();
22     }
23 }
24
25 void pong() {
26     while(counter <= countlimit) {
27         condAtomicFlag2.wait(false);
28         condAtomicFlag2.clear();
29
30         condAtomicFlag1.test_and_set();
31         condAtomicFlag1.notify_one();
32     }
33 }
34
35 int main() {
```

```
36      auto start = std::chrono::system_clock::now();
37
38     condAtomicFlag1.test_and_set();
39     std::thread t1(ping);
40     std::thread t2(pong);
41
42     t1.join();
43     t2.join();
44
45     std::chrono::duration<double> dur =
46         std::chrono::system_clock::now() - start;
47     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
48 }
```

Вызов функции `wait` для переменной `condAtomicFlag1` в строке 15 блокирует поток, если атомарный флаг имеет значение `false`. Если же флаг находится в значении `true`, вызов этой функции сразу возвращает управление. Хранящееся в переменной логическое значение играет ту же роль, что и предикат для переменной условия, поэтому после разблокировки потока флаг нужно снова сбросить в `false` (строка 16). Симметричным образом, перед отсылкой оповещения pong-потоку в строке 21 второй флаг устанавливается в значение `true` (строка 20). Начальная установка флага `condAtomicFlag1` в значение `true` в главном потоке (строка 38) запускает игру. Благодаря атомарному флагу теперь она идёт заметно быстрее: средняя продолжительность игры составляет 0,32 с.



Многократная синхронизация с двумя атомарными флагами

Посмотрев на эту программу внимательнее, можно обнаружить, что для организации взаимодействия потоков довольно и одного атомарного флага.

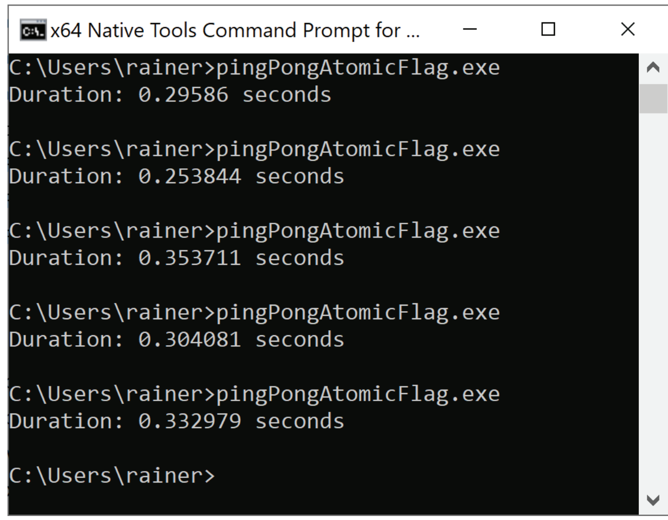
6.4.2.2. Решение с одним атомарным флагом

Использование одного атомарного флага делает процесс функционирования системы более очевидным.

Многократная синхронизация с одним атомарным флагом

```
1  // pingPongAtomicFlag.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag{};
8
9  std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         condAtomicFlag.wait(true);
15         condAtomicFlag.test_and_set();
16
17         ++counter;
18
19         condAtomicFlag.notify_one();
20     }
21 }
22
23 void pong() {
24     while(counter <= countlimit) {
25         condAtomicFlag.wait(false);
26         condAtomicFlag.clear();
27         condAtomicFlag.notify_one();
28     }
29 }
30
31 int main() {
32     auto start = std::chrono::system_clock::now();
33
34     condAtomicFlag.test_and_set();
35     std::thread t1(ping);
36     std::thread t2(pong);
37
38     t1.join();
39     t2.join();
40
41     std::chrono::duration<double> dur =
42         std::chrono::system_clock::now() - start;
43     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
44 }
```

В этой программе ping-поток блокируется до тех пор, пока флаг остаётся в значении `true`, а pong-поток, наоборот, блокируется, пока флаг имеет значение `false`. Использование одного флага вместо двух не оказывает существенного влияния на быстродействие. Среднее время выполнения программы составляет 0,31 с.



Многократная синхронизация с одним атомарным флагом

В этом примере переменная типа `std::atomic_flag` играла роль атомарной переменной логического типа. Сделаем ещё одну попытку – на этот раз с типом `std::atomic<bool>`.

6.4.3. Решение на основе атомарной логической переменной

Реализация на базе типа `std::atomic<bool>` представлена ниже.

Многократная синхронизация с атомарной логической переменной

```
1  // pingPongAtomicBool.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic<bool> atomicBool{};
8
9  std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
```

```
12 void ping() {
13     while(counter <= countlimit) {
14         atomicBool.wait(true);
15         atomicBool.store(true);
16 
17         ++counter;
18 
19         atomicBool.notify_one();
20     }
21 }
22 
23 void pong() {
24     while(counter <= countlimit) {
25         atomicBool.wait(false);
26         atomicBool.store(false);
27         atomicBool.notify_one();
28     }
29 }
30 
31 int main() {
32     std::cout << std::boolalpha << '\n';
33 
34     std::cout << "atomicBool.is_lock_free(): "
35               << atomicBool.is_lock_free()  << '\n';
36 
37     std::cout << '\n';
38 
39     auto start = std::chrono::system_clock::now();
40 
41     atomicBool.store(true);
42     std::thread t1(ping);
43     std::thread t2(pong);
44 
45     t1.join();
46     t2.join();
47 
48     std::chrono::duration<double> dur =
49         std::chrono::system_clock::now() - start;
50     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
51 }
```

Реализация типа `std::atomic<bool>` имеет право использовать тот или иной механизм блокировки, например мьютекс. Библиотека для системы Windows, которой пользовался автор, не содержит блокировок. Результат выполнения программы показан на рисунке. Среднее время выполнения программы составило 0,38 с.

```
x64 Native Tools Command Prompt for V...
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.424524 seconds
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.357399 seconds
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.38501 seconds
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.370447 seconds
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.400319 seconds
C:\Users\rainer>
```

Многократная синхронизация с атомарной логической переменной

С точки зрения устройства исходного текста, реализация на основе атомарной логической переменной чрезвычайно проста для понимания. Это справедливо и для следующей реализации, основанной на семафорах.

6.4.4. Реализация на семафорах

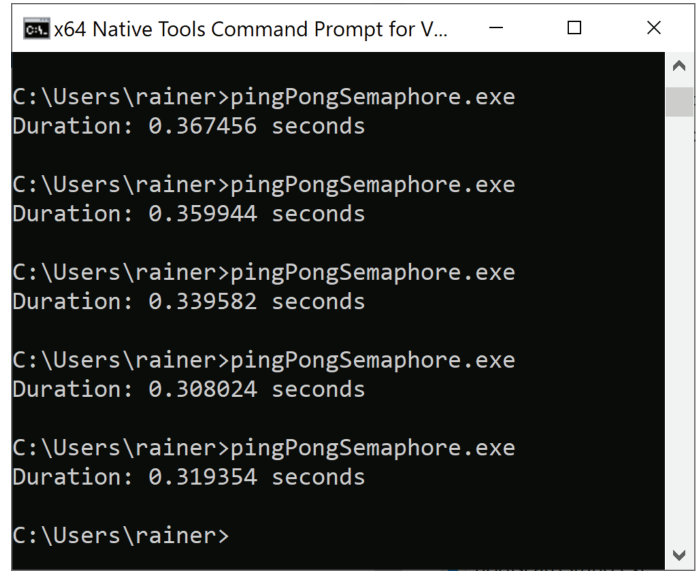
Можно ожидать, что семафоры будут работать быстрее, чем переменные условия. Проверим, так ли это.

Многократная синхронизация на основе семафоров

```
1  // pingPongSemaphore.cpp
2
3  #include <iostream>
```

```
4  #include <semaphore>
5  #include <thread>
6
7  std::counting_semaphore<1> signal2Ping(0);
8  std::counting_semaphore<1> signal2Pong(0);
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12
13 void ping() {
14     while(counter <= countlimit) {
15         signal2Ping.acquire();
16         ++counter;
17         signal2Pong.release();
18     }
19 }
20
21 void pong() {
22     while(counter <= countlimit) {
23         signal2Pong.acquire();
24         signal2Ping.release();
25     }
26 }
27
28 int main() {
29     auto start = std::chrono::system_clock::now();
30
31     signal2Ping.release();
32     std::thread t1(ping);
33     std::thread t2(pong);
34
35     t1.join();
36     t2.join();
37
38     std::chrono::duration<double> dur =
39         std::chrono::system_clock::now() - start;
40     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
41 }
```

В этой программе используются два семафора: `signal2Ping` и `signal2Pong` (строки 7 и 8). Оба могут иметь два возможных значения, 0 и 1, и инициализируются значением 0. Когда значение семафора равно 0, вызов функции `release` устанавливает его в 1 и разблокирует ожидающий поток, работая тем самым как оповещение. Вызов функции `acquire`, напротив, блокирует поток до тех пор, пока семафор не получит ненулевое значение. На следующем рисунке показан результат работы программы. Среднее время работы составляет 0,33 с.



Многократная синхронизация на основе семафоров

6.4.5. Сравнительный анализ

Как и следовало ожидать, переменные условия оказываются самым медленным механизмом синхронизации, а атомарные флаги – самым быстрым. Быстродействие атомарной переменной логического типа лежит посередине. У последнего решения есть свой недостаток: стандарт не гарантирует отсутствие блокировок в его реализации, в отличие от типа `std::atomic_flag`. Неожиданно высокой оказалась производительность семафоров: они почти не уступают атомарным флагам.

Время выполнения программы

Вариант реализации	Среднее время, с
Переменные условия	0,52
Два атомарных флага	0,32
Один атомарный флаг	0,31
Атомарная логическая переменная	0,38
Семафоры	0,33

6.5. Вариации на тему фьючерсов

Прежде чем разбирать разнообразные вариации программы из раздела 5.6, нужно понять принцип её работы. Текстовые сообщения в тексте программы облегчают её понимание.

Жадный фьючерс

```
1  // eagerFutureWithComments.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      std::shared_ptr<T> value;
10     MyFuture(std::shared_ptr<T> p): value(p) {
11         std::cout << "    MyFuture::MyFuture\n";
12     }
13     ~MyFuture() {
14         std::cout << "    MyFuture::~MyFuture\n";}
15     T get() {
16         std::cout << "    MyFuture::get\n";
17         return *value;
18     }
19
20     struct promise_type {
21         std::shared_ptr<T> ptr = std::make_shared<T>();
22         promise_type() {
23             std::cout << "        promise_type::promise_type\n";
24         }
25         ~promise_type() {
26             std::cout << "        promise_type::~promise_type\n";
27         }
28         MyFuture<T> get_return_object() {
29             std::cout << "        promise_type::get_return_object\n";
30             return ptr;
31         }
32
33         void return_value(T v) {
34             std::cout << "        promise_type::return_value\n";
35             *ptr = v;
36         }
37
38         std::suspend_never initial_suspend() {
39             std::cout << "        promise_type::initial_suspend\n";
40             return {};
41         }
42
43         std::suspend_never final_suspend() noexcept {
44             std::cout << "        promise_type::final_suspend\n";
45             return {};
46         }
47
48         void unhandled_exception() {
49             std::exit(1);
50         }
51     };
```

```
52 };
53
54 MyFuture<int> createFuture() {
55     std::cout << "createFuture\n";
56     co_return 2021;
57 }
58
59 int main() {
60     std::cout << '\n';
61
62     auto fut = createFuture();
63     auto res = fut.get();
64     std::cout << "res: " << res << '\n';
65
66     std::cout << '\n';
67 }
```

Вызов функции `createFuture` в строке 61 приводит к созданию экземпляра типа `MyFuture` (строка 8). Перед тем как завершится вызов конструктора, реализованного в строке 10, создаётся, отрабатывается и уничтожается объект-обещание типа `promise_type`, объявленного в строках 20–50. На каждом шаге своего жизненного цикла объект-обещание пользуется контроллером ожидания `std::suspend_never` (строки 38 и 43), поэтому выполнение не приостанавливается. Результат работы объекта-обещания нужно сохранить до будущего вызова функции `get` в строке 63, для этого нужно выделить память. Использование умных указателей типа `std::shared_ptr` в строках 9 и 21 гарантирует, что в программе не происходит утечки памяти. Когда выполнение программы выходит за область видимости локальной переменной `fut`, вызывается деструктор. Читатель может увидеть эту программу в действии с помощью интерактивного инструмента Compiler Explorer[1]. Результат работы программы показан на следующем рисунке.

```
        promise_type::promise_type
        promise_type::get_return_object
        promise_type::initial_suspend
createFuture
        promise_type::return_value
        promise_type::final_suspend
        promise_type::~promise_type
    MyFuture::MyFuture
    MyFuture::get
res: 2021

    MyFuture::~MyFuture
```

Жадный фьючерс

1 https://godbolt.org/z/Y9naEx.

Показанная в этом примере сопрограмма запускается немедленно после создания и тем самым работает по жадному принципу. Более того, эта сопрограмма выполняется в том же потоке, который её вызвал. Сделаем сопрограмму ленивой.

6.5.1. Ленивый фьючерс

Ленивый фьючерс – это фьючерс, который выполняется только тогда, когда запрошено его значение. Посмотрим, что для этого нужно изменить в реализации жадной сопрограммы из предыдущего раздела.

Ход выполнения жадного фьючерса

```
1  // lazyFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      struct promise_type;
10     using handle_type = std::coroutine_handle<promise_type>;
11
12     handle_type coro;
13
14     MyFuture(handle_type h): coro(h) {
15         std::cout << “    MyFuture::MyFuture\n”;
16     }
17     ~MyFuture() {
18         std::cout << “    MyFuture::~MyFuture\n”;
19         if (coro) coro.destroy();
20     }
21
22     T get() {
23         std::cout << “    MyFuture::get\n”;
24         coro.resume();
25         return coro.promise().result;
26     }
27
28     struct promise_type {
29         T result;
30
31         promise_type() {
32            std::cout << “        promise_type::promise_type\n”;
33         }
34
35         ~promise_type() {
36             std::cout << “        promise_type::~promise_type\n”;
37         }
38
```

```
39          auto get_return_object() {
40              std::cout << "        promise_type::get_return_object\n";
41              return MyFuture{handle_type::from_promise(*this)};
42          }
43
44          void return_value(T v) {
45              std::cout << "        promise_type::return_value\n";
46              result = v;
47          }
48
49          std::suspend_always initial_suspend() {
50              std::cout << "        promise_type::initial_suspend\n";
51              return {};
52          }
53
54          std::suspend_always final_suspend() noexcept {
55              std::cout << "        promise_type::final_suspend\n";
56              return {};
57          }
58
59          void unhandled_exception() {
60              std::exit(1);
61          }
62      };
63  };
64
65  MyFuture<int> createFuture() {
66      std::cout << "createFuture\n";
67      co_return 2021;
68  }
69
70  int main() {
71      std::cout << '\n';
72
73      auto fut = createFuture();
74      auto res = fut.get();
75      std::cout << "res: " << res << '\n';
76
77      std::cout << '\n';
78  }
```

Рассмотрим сначала объект-обещание. Обещание всегда приостанавливается в начале (строка 44) и в конце (строка 48). Далее функция-член `get_return_object` (строка 36) создаёт объект-фьючерс, который затем возвращается в функцию `createFuture`, которая запустила сопрограмму (строка 58). Класс `MyFuture` более интересен. В нём есть член данных `coro`, объявленный в строке 12, – дескриптор сопрограммы. Через этот дескриптор объект `MyFuture` управляет объектом-обещанием. В частности, он возобновляет работу обещания (строка 24), запрашивает у него результат (строка 25) и, наконец, уничтожает его (строка 19). Возобновление работы обещания необходимо, так как ранее, сразу после создания, оно было приостановлено (строка 44). Когда клиентский код вызывает функцию `get` объекта-фьючерса в строке 67,

чтобы узнать результат работы фьючерса, он тем самым неявно возобновляет работу обещания благодаря строке 24.

Работу этой программы можно наблюдать в среде Compiler Explorer[1]. В процессе работы программа выводит последовательность выполнения, как показано на рисунке.

```
        promise_type::promise_type
        promise_type::get_return_object
    MyFuture::MyFuture
        promise_type::initial_suspend
    MyFuture::get
createFuture
        promise_type::return_value
        promise_type::final_suspend
res: 2021

    MyFuture::~MyFuture
        promise_type::~promise_type
```

Ленивый фьючерс

Однако что произойдёт, если клиенту не нужен результат работы фьючерса? Проверим.

Создание сопрограммы без последующего пробуждения

```
1  int main() {
2      std::cout << '\n';
3
4      auto fut = createFuture();
5      auto res = fut.get();
6      std::cout << "res: " << res << '\n';
7
8      std::cout << '\n';
9  }
```

Как и следовало ожидать, объект-обещание не начинает своё выполнение, а функции-члены `return_value` и `final_suspend` не вызываются.

```
    promise_type::promise_type
    promise_type::get_return_object
MyFuture::MyFuture
    promise_type::initial_suspend

MyFuture::~MyFuture
    promise_type::~promise_type
```

Ленивый фьючерс без последующего пробуждения

[1] https://godbolt.org/z/EejWcj.

Сложности с временем жизни сопрограмм

Одна из сложностей, возникающих при работе с сопрограммами, касается управления временем жизни сопрограмм. В примере с жадным фьючерсом из предыдущего раздела результат работы сопрограммы пришлось сохранить в умном указателе `std::shared_ptr`. Это необходимо потому, что сопрограмма выполняется жадным образом, и обещание успевает полностью отработать до того, как клиент запрашивает результат у фьючерса.

В ленивой программе, напротив, выполнение сопрограммы всегда приостанавливается перед завершением, так как функция `final_suspend` возвращает значение типа `suspend_always` (строка 48). Следовательно, объект-обещание доживает до запроса значения клиентом, и умный указатель оказывается более не нужен. Если бы функция `final_suspend` возвращала значение типа `suspend_never`, это привело бы к неопределённому поведению, поскольку в этом случае объект-обещание живёт меньше времени, чем связанный с ним фьючерс. Следовательно, клиент пытался бы обратиться к уже уничтоженному обещанию.

Попробуем ещё больше изменить сопрограмму – так, чтобы она выполнялась в отдельном потоке.

6.5.2. Выполнение сопрограммы в отдельном потоке

Сопрограмма из следующего примера всегда создаётся в приостановленном состоянии, поскольку функция-член `final_suspend` возвращает значение типа `suspend_always`. Следовательно, обещание можно выполнять в отдельном потоке.

Запуск обещания в отдельном потоке

```
1  // lazyFutureOnOtherThread.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6  #include <thread>
7
8  template<typename T>
9  struct MyFuture {
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12     handle_type coro;
13
14     MyFuture(handle_type h): coro(h) {}
15     ~MyFuture() {
16         if ( coro ) coro.destroy();
17     }
18
19     T get() {
20         std::cout << "    MyFuture::get:  "
21                   << "std::this_thread::get_id(): "
```

```
22                  << std::this_thread::get_id() << ‘\n’;
23 
24         std::thread t([this] { coro.resume(); });
25         t.join();
26         return coro.promise().result;
27     }
28 
29     struct promise_type {
30         promise_type(){
31             std::cout << “        promise_type::promise_type:  “
32                       << “std::this_thread::get_id(): “
33                       << std::this_thread::get_id() << ‘\n’;
34         }
35 
36         ~promise_type(){
37             std::cout << “        promise_type::~promise_type:  “
38                       << “std::this_thread::get_id(): “
39                       << std::this_thread::get_id() << ‘\n’;
40         }
41 
42         T result;
43 
44         auto get_return_object() {
45             return MyFuture{handle_type::from_promise(*this)};
46         }
47 
48         void return_value(T v) {
49             std::cout << “        promise_type::return_value:  “
50                       << “std::this_thread::get_id(): “
51                       << std::this_thread::get_id() << ‘\n’;
52             std::cout << v << std::endl;
53             result = v;
54         }
55 
56         std::suspend_always initial_suspend() {
57             return {};
58         }
59 
60         std::suspend_always final_suspend() noexcept {
61             std::cout << “        promise_type::final_suspend:  “
62                       << “std::this_thread::get_id(): “
63                       << std::this_thread::get_id() << ‘\n’;
64             return {};
65         }
66 
67         void unhandled_exception() {
68             std::exit(1);
69         }
70     };
71 };
72 
73 MyFuture<int> createFuture() {
74     co_return 2021;
```

```
75 }
76
77 int main() {
78     std::cout << '\n';
79
80     std::cout << "main:  "
81               << "std::this_thread::get_id(): "
82               << std::this_thread::get_id() << '\n';
83
84     auto fut = createFuture();
85     auto res = fut.get();
86     std::cout << "res: " << res << '\n';
87
88     std::cout << '\n';
89 }
```

Эта программа дополнительно выводит идентификатор потока, в котором выполняется каждая функция. Эта программа очень похожа на предыдущую, где ленивый фьючерс запускался в вызывающем потоке. Главное различие – в функции-члене `get` (строка 19). Функция `resume`, возобновляющая выполнение приостановленного на старте обещания, запускается в новом потоке (строка 24). За выполнением этой программы можно понаблюдать с помощью интерактивной среды на сайте Wandbox[1]. Результат выполнения программы показан на рисунке.

```
main:  std::this_thread::get_id(): 139819561723776
        promise_type::promise_type:  std::this_thread::get_id(): 139819561723776
    MyFuture::get:  std::this_thread::get_id(): 139819561723776
        promise_type::return_value:  std::this_thread::get_id(): 139819456755456
        promise_type::final_suspend:  std::this_thread::get_id(): 139819456755456
res: 2021

        promise_type::~promise_type:  std::this_thread::get_id(): 139819561723776
```

Запуск обещания в отдельном потоке

Хотелось бы сказать ещё несколько слов о функции-члене `get`. Чрезвычайно важно, чтобы обещание, пробуждённое для выполнения в отдельном потоке, завершилось до того, как функция `get` вернёт результат обещания. В этом примере, если исключить вывод диагностических сообщений, функция имеет вид:

Функция get

```
T get() {
    std::thread t([this] { coro.resume(); });
    t.join();
    return coro.promise().result;
}
```

[1] https://wandbox.org/permlink/jFVVj80Gxu6bnNkc.

Если бы эта функция присоединялась к потоку `t` с помощью функции `join` после того, как взяла результат из объекта-обещания, поведение программы было бы неопределённым. Ниже показана соответствующим образом видоизменённая функция `get` с использованием типа `std::jthread`, который обеспечивает автоматическое присоединение потока в деструкторе. Однако в момент возврата значения уже слишком поздно ожидать завершения потока.

Модифицированная функция get с потоком типа std::jthread

```
T get() {
    std::jthread t([this] { coro.resume(); });
    return coro.promise().result;
}
```

В этом случае клиент, скорее всего, получит результат вычислений до того, как функция `return_value` его подготовит. Член `reault` объекта-обещания в момент взятия значения имеет произвольное неопределённое значение – оно и попадает в переменную `res` в главной функции. Ход выполнения программы показан на рисунке.

```
main:  std::this_thread::get_id(): 139913381070720
        promise_type::promise_type:  std::this_thread::get_id(): 139913381070720
    MyFuture::get:  std::this_thread::get_id(): 139913381070720
        promise_type::return_value:  std::this_thread::get_id(): 139913276102400
        promise_type::final_suspend:  std::this_thread::get_id(): 139913276102400
res: -1

        promise_type::~promise_type:  std::this_thread::get_id(): 139913381070720
```

Выполнение обещания в другом потоке

Есть несколько способов гарантировать завершение вычисляющего потока до возврата значения. Например, можно поместить объект `std::jthread` в ограниченную область видимости.

Функция get с потоком std::jthread в ограниченной области видимости

```
T get() {
    {
        std::jthread t([this] { coro.resume(); });
    }
    return coro.promise().result;
}
```

Также поток `std::jthread` можно сделать временным объектом[1].

---

[1] Такой объект уничтожается сразу после создания, при этом деструктор обеспечивает ожидание завершения потока. – *Прим. перев.*

Функция get с временным объектом типа std::jthread

```
T get() {
    std::jthread([this] { coro.resume(); });
    return coro.promise().result;
}
```

Этот последний вариант, впрочем, не стоит рассматривать из-за неудобочитаемости: может понадобиться несколько секунд, чтобы распознать в этом тексте вызов конструктора типа `std::jthread` с последующей деструкцией объекта.

6.6. Модификации и обобщения генераторов

Прежде чем видоизменить и обобщить генератор бесконечного потока данных из раздела 5.7.1, рассмотрим снова первоначальную реализацию как отправную точку долгого пути. В текст программы добавлен вывод многочисленных диагностических сообщений, а главная функция запрашивает из бесконечного потока только три элемента.

Генератор бесконечного потока данных

```
1  // infiniteDataStreamComments.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9      struct promise_type;
10     using handle_type = std::coroutine_handle<promise_type>;
11
12     Generator(handle_type h): coro(h) {
13         std::cout << "        Generator::Generator\n";
14     }
15
16     handle_type coro;
17
18     ~Generator() {
19         std::cout << "        Generator::~Generator\n";
20         if ( coro ) coro.destroy();
21     }
22
23     Generator(const Generator&) = delete;
24     Generator& operator = (const Generator&) = delete;
25
26     Generator(Generator&& oth): coro(oth.coro) {
```

```
27          oth.coro = nullptr;
28      }
29
30      Generator& operator= (Generator&& oth) {
31          coro = oth.coro;
32          oth.coro = nullptr;
33          return *this;
34      }
35
36      int getNextValue() {
37          std::cout << “        Generator::getNextValue\n”;
38          coro.resume();
39          return coro.promise().current_value;
40      }
41
42      struct promise_type {
43          promise_type() {
44              std::cout << “            promise_type::promise_type\n”;
45          }
46
47          ~promise_type() {
48              std::cout << “            promise_type::~promise_type\n”;
49          }
50
51          std::suspend_always initial_suspend() {
52              std::cout << “            promise_type::initial_suspend\n”;
53              return {};
54          }
55
56          std::suspend_always final_suspend() noexcept {
57              std::cout << “            promise_type::final_suspend\n”;
58              return {};
59          }
60
61          auto get_return_object() {
62              std::cout <<”            promise_type::get_return_object\n”;
63              return Generator{handle_type::from_promise(*this)};
64          }
65
66          std::suspend_always yield_value(int value) {
67              std::cout << “            promise_type::yield_value\n”;
68              current_value = value;
69              return {};
70          }
71
72          void return_void() {}
73
74          void unhandled_exception() {
75              std::exit(1);
76          }
77
78          T current_value;
79      };
80  };
```

```
81
82 Generator<int> getNext(int start = 10, int step = 10) {
83     std::cout << "    getNext: start\n";
84     auto value = start;
85     while (true) {
86         std::cout << "    getNext: before co_yield\n";
87         co_yield value;
88         std::cout << "    getNext: after co_yield\n";
89         value += step;
90     }
91 }
92
93 int main() {
94     auto gen = getNext();
95     for (int i = 0; i <= 2; ++i) {
96         auto val = gen.getNextValue();
97         std::cout << "main: " << val << '\n';
98     }
99 }
```

Если запустить эту программу (например, в среде Compiler Explorer[1]), она напечатает текст, делающий процесс её выполнения очевидным.

```
            promise_type::promise_type
            promise_type::get_return_object
        Generator::Generator
            promise_type::initial_suspend
        Generator::getNextValue
    getNext: start
    getNext: before co_yield
            promise_type::yield_value
main: 10
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 20
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 30
        Generator::~Generator
            promise_type::~promise_type
```

Генератор бесконечного потока данных

[1] https://godbolt.org/z/cTW9Gq.

Рассмотрим процесс выполнения программы подробнее. Вызов функции getNext в строке 94 приводит к созданию объекта класса Generator<int>. Для этого в первую очередь создаётся объект типа promise_type (строка 42), затем вызов функции get_return_object (строка 61) создаёт объект-генератор (строка 63) и сохраняет объект-обещание, завёрнутый в дескриптор сопрограммы, в переменной-члене coro объекта-генератора. Результат этого вызова возвращается клиенту, когда сопрограмма приостанавливается в первый раз. Первоначальная приостановка происходит сразу после создания программы (строка 51). Поскольку функция initial_suspend возвращает контроллер ожидания типа std::suspend_always, выполнение возвращается в функцию getNext и продолжается до оператора co_yield в строке 87. Этот оператор приводит в функцию yield_value (строка 66), которая запоминает очередное значение, которое в дальнейшем будет отдано клиенту. Функция yield_value возвращает контроллер ожидания типа std::suspend_always, поэтому выполнение сопрограммы приостанавливается, а управление передаётся в главную функцию, которая запускает цикл for в строке 95. Вызов функции getNextValue объекта-генератора gen в строке 96 возобновляет выполнение сопрограммы путём вызова функции resume для дескриптора сопрограммы (строка 38). Затем функция getNextValue возвращает текущее значение, которое было сохранено в объекте-обещании при предыдущем вызове функции yield_value (строка 66). Наконец, сгенерированное сопрограммой значение выводится на печать в строке 97, и цикл продолжает свою работу со следующей итерации. В конце работы программы уничтожаются генератор и обещания.

После подробного разбора этой программы можно внести в неё первое изменение.

6.6.1. Модификации программы

Код и нумерация строк в этом разделе остаются теми же, что и в предыдущем примере. Для краткости здесь будем показывать лишь изменившиеся участки кода.

6.6.1.1. Если сопрограмму не пробуждать

Если убрать из кода программы строки, вызывающие пробуждение сопрограммы (вызов функции getNextValue в строке 96 и печать значения в строке 97), сопрограмма останавливается сразу после создания.

Создание сопрограммы без последующего пробуждения

```
int main() {
    auto gen = getNext();
    for (int i = 0; i <= 2; ++i) {
        // auto val = gen.getNextValue();
        // std::cout << "main: " << val << '\n';
    }
}
```

Эта сопрограмма никогда ничего не делает. В конце концов, генератор и обещания уничтожаются в конце выполнения программы. Результат работы программы показан на рисунке.

```
    promise_type::promise_type
    promise_type::get_return_object
Generator::Generator
    promise_type::initial_suspend
Generator::~Generator
    promise_type::~promise_type
```

Создание сопрограммы без последующего пробуждения

6.6.1.2. Сопрограмма не приостанавливается на старте

В исходной версии программы функция-член `initial_suspend` объекта-обещания (строка 51) возвращает контроллер ожидания типа `std::suspend_always` (с англ. «приостанавливать всегда»). Как и явствует из названия, он вызывает приостановку сопрограммы сразу после старта. Попробуем вместо него возвращать контроллер ожидания `std::suspend_never`, который запрещает сопрограмму приостанавливаться.

Сопрограмма без первоначальной приостановки

```
std::suspend_never initial_suspend() {
    std::cout << "            promise_type::initial_suspend\n";
    return {};
}
```

В этом случае сопрограмма запускается непосредственно после создания и приостанавливается, только когда выполнение доходит до функции `yield_value` (строка 66). Последующий вызов функции `getNextValue` в строке 96 возобновляет работу сопрограммы и снова приводит к вызову функции `yield_value`. В результате этого значение 10, сгенерированное первым, игнорируется, и сопрограмма возвращает значения 20, 30 и 40. Ход выполнения программы показан на рисунке.

```
            promise_type::promise_type
            promise_type::get_return_object
        Generator::Generator
            promise_type::initial_suspend
    getNext: start
    getNext: before co_yield
            promise_type::yield_value
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 20
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 30
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 40
        Generator::~Generator
            promise_type::~promise_type
```

Не возобновляйте сопрограмму

6.6.1.3. Сопрограмма не приостанавливается при выдаче значения

Функция `yield_value` (строка 66) запускается оператором `co_yield` и запоминает очередное значение, которое затем должно быть отдано клиенту. Эта функция возвращает контроллер ожидания `std::suspend_always` и тем самым приостанавливает выполнение сопрограммы. Последующий вызов функции `getNextValue` в строке 96 должен возобновить её выполнение. Посмотрим, что получится, если контроллер ожидания, возвращаемый из функции `yield_value`, заменить на `std::suspend_never`.

Сопрограмма без приостановки при выдаче значения

```
std::suspend_never yield_value(int value) {
    std::cout << "            promise_type::yield_value\n";
    current_value = value;
    return {};
}
```

Как и следовало предполагать, теперь цикл в строках 85–90 выполняется вечно, а сопрограмма не возвращает клиенту никаких значений. Программа выдаёт диагностические сообщения, показанные на следующем рисунке.

```
        promise_type::promise_type
        promise_type::get_return_object
    Generator::Generator
        promise_type::initial_suspend
    Generator::getNextValue
getNext: start
getNext: before co_yield
        promise_type::yield_value
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
getNext: after co_yield
```

Сопрограмма без приостановки при выдаче значения

Не составляет труда так преобразовать генератор, чтобы он вырабатывал конечную последовательность значений.

6.6.2. Обобщение

Читателя могло удивить, что в приведённых ранее примерах потенциал генератора как универсального механизма использован далеко не полностью. Изменим программу так, чтобы генератор мог выдавать один за другим элементы произвольного контейнера из стандартной библиотеки.

Генератор, поочерёдно возвращающий элементы контейнера

```
1  // coroutineGetElements.cpp
2
3  #include <coroutine>
4  #include <memory>
```

```
5  #include <iostream>
6  #include <string>
7  #include <vector>
8
9  template<typename T>
10 struct Generator {
11     struct promise_type;
12     using handle_type = std::coroutine_handle<promise_type>;
13
14     Generator(handle_type h): coro(h) {}
15
16     handle_type coro;
17     std::shared_ptr<T> value;
18
19     ~Generator() {
20         if ( coro ) coro.destroy();
21     }
22
23     Generator(const Generator&) = delete;
24     Generator& operator = (const Generator&) = delete;
25     Generator(Generator&& oth): coro(oth.coro) {
26         oth.coro = nullptr;
27     }
28
29     Generator& operator = (Generator&& oth) {
30         coro = oth.coro;
31         oth.coro = nullptr;
32         return *this;
33     }
34
35     T getNextValue() {
36         coro.resume();
37         return coro.promise().current_value;
38     }
39
40     struct promise_type {
41         promise_type() {}
42
43         ~promise_type() {}
44
45         std::suspend_always initial_suspend() {
46             return {};
47         }
48
49         std::suspend_always final_suspend() noexcept {
50             return {};
51         }
52
53         auto get_return_object() {
54             return Generator{handle_type::from_promise(*this)};
55         }
56
57         std::suspend_always yield_value(const T value) {
```

```
58              current_value = value;
59              return {};
60          }
61
62          void return_void() {}
63
64          void unhandled_exception() {
65              std::exit(1);
66          }
67
68          T current_value;
69      };
70  };
71
72  template <typename Cont>
73  Generator<typename Cont::value_type> getNext(Cont cont) {
74      for (auto c: cont) co_yield c;
75  }
76
77  int main() {
78      std::string helloWorld = "Hello world";
79      auto gen = getNext(helloWorld);
80      for (int i = 0; i < helloWorld.size(); ++i) {
81          std::cout << gen.getNextValue() << " ";
82      }
83
84      std::cout << "\n\n";
85      auto gen2 = getNext(helloWorld);
86      for (int i = 0; i < 5 ; ++i) {
87          std::cout << gen2.getNextValue() << " ";
88      }
89
90      std::cout << "\n\n";
91      std::vector myVec{1, 2, 3, 4 ,5};
92      auto gen3 = getNext(myVec);
93      for (int i = 0; i < myVec.size() ; ++i) {
94          std::cout << gen3.getNextValue() << " ";
95      }
96
97      std::cout << '\n';
98  }
```

В этом примере генератор создаётся и используется трижды. В первых двух случаях (строки 79 и 85) инициализируются строкой, тогда как третий генератор (строка 92) – вектором целых чисел. Вывод программы неудивителен. Строка 81 печатает все символы строки по порядку, строка 87 – только первые пять символов, а строка 94 – элементы вектора целых чисел.

Эту программу можно посмотреть в действии, например в среде Compiler Explorer[1]. Результат её работы показан на рисунке.

[1] https://godbolt.org/z/j9znva.

```
H e l l o   w o r l d

H e l l o

1 2 3 4 5
```

Генератор, проходящий по элементам контейнера

Реализация класса `Generator` почти совпадает с исходным вариантом, представленным в начале раздела. Значительное различие имеется только в функции `getNext`.

Функция getNext

```
template <typename Cont>
Generator<typename Cont::value_type> getNext(Cont cont) {
    for (auto c: cont) co_yield c;
}
```

Теперь это шаблон функции, которая в качестве аргумента принимает контейнер и в цикле по диапазону проходит по всем его элементам. После каждой итерации выполнение этой функции приостанавливается. Тип возвращаемого значения этой функции может показаться читателю удивительным. Тип `Cont::value_type` – это член типа-параметра шаблона. Компилятору нужна подсказка о том, как следует понимать это составное имя. По умолчанию в сомнительных случаях компилятор предполагает, что имя относится к не типу (например, именует переменную или функцию). Поэтому в данном случае перед составным именем необходимо поставить ключевое слово `typename`.

6.7. Способы управления заданиями

Прежде чем менять поведение программы из раздела 5.8, попробуем сделать его более очевидным.

6.7.1. Функционирование контроллера ожидания

Возьмём уже изученную программу `startJob.cpp` и добавим вывод трассирующих сообщений.

Запуск задания по запросу (с трассировкой)

```
1  // startJobWithComments.cpp
2
3  #include <coroutine>
4  #include <iostream>
```

```
5
6  struct MySuspendAlways {
7      bool await_ready() const noexcept {
8          std::cout << "       MySuspendAlways::await_ready\n";
9          return false;
10     }
11
12     void await_suspend(std::coroutine_handle<>) const noexcept {
13         std::cout << "       MySuspendAlways::await_suspend\n";
14
15     }
16
17     void await_resume() const noexcept {
18         std::cout << "       MySuspendAlways::await_resume\n";
19     }
20 };
21
22 struct MySuspendNever {
23     bool await_ready() const noexcept {
24         std::cout << "       MySuspendNever::await_ready\n";
25         return true;
26     }
27
28     void await_suspend(std::coroutine_handle<>) const noexcept {
29         std::cout << "       MySuspendNever::await_suspend\n";
30
31     }
32
33     void await_resume() const noexcept {
34         std::cout << "       MySuspendNever::await_resume\n";
35     }
36 };
37
38 struct Job {
39     struct promise_type;
40     using handle_type = std::coroutine_handle<promise_type>;
41     handle_type coro;
42     Job(handle_type h): coro(h){}
43
44     ~Job() {
45         if ( coro ) coro.destroy();
46     }
47
48     void start() {
49         coro.resume();
50     }
51
52     struct promise_type {
53         auto get_return_object() {
54             return Job{handle_type::from_promise(*this)};
55         }
56
57         MySuspendAlways initial_suspend() {
```

```
58              std::cout << “    Job prepared\n”;
59              return {};
60          }
61
62          MySuspendAlways final_suspend() noexcept {
63              std::cout << “    Job finished\n”;
64              return {};
65          }
66
67          void return_void() {}
68          void unhandled_exception() {}
69      };
70  };
71
72  Job prepareJob() {
73      co_await MySuspendNever();
74  }
75
76  int main() {
77      std::cout <<  “Before job\n”;
78
79      auto job = prepareJob();
80      job.start();
81
82      std::cout <<  “After job\n”;
83  }
```

Прежде всего библиотечные контроллеры ожидания `std::suspend_always` и `std::suspend_never` заменены самодельными аналогами `MySuspendAlways` (строка 6) и `MySuspendNever` (строка 22). Они используются далее в строках 57, 62 и 73. Эти классы имитируют поведение соответствующих стандартных контроллеров ожидания, но, помимо этого, выводят сообщения. Из-за наличия операций вывода в поток `std::cout` их функции-члены `await_ready`, `await_suspend` и `await_resume` не получится объявить со спецификатором `constexpr`.

На следующем рисунке показан результат выполнения программы в среде Compiler Explorer[1].

```
Before job
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
After job
```

Запуск задания по запросу (с трассировкой)

[1] https://godbolt.org/z/T5rcE4.

Функция initial_suspend (строка 57) выполняется в начале сопрограммы, а функция final_suspend (строка 62) – в конце. Вызов функции prepare_job в строке 79 запускает создание объекта-сопрограммы, а вызов функции-члена start в строке 80 заставляет приостановившуюся после создания сопрограмму продолжить выполнение и, следовательно, завершиться. При этом вызываются последовательно функции-члены await_ready, await_suspend и await_resume класса MySuspendAlways. Если не пробуждать приостановленную сопрограмму, как, например, объект, возвращённый функцией final_suspend, функция await_resume вызвана не будет. В отличие от этого, контроллер MySuspendNever сразу сообщает о готовности своего результата, поскольку его функция await_ready возвращает значение true, и поэтому не приостанавливается.

Имея перед глазами подробное описание процесса выполнения программы, можно получить некоторое представление о жизненном цикле контроллеров ожидания. Пришла пора внести в него некоторые изменения.

6.7.2. Автоматическое возобновление работы

В рассмотренном выше коде запуск задания осуществляется в явном виде:

```
int main() {
    std::cout <<  "Before job\n";

    auto job = prepareJob();
    job.start();

    std::cout <<  "After job\n";
}
```

Явный вызов функции start объекта job необходим, поскольку функция await_ready контроллера ожидания MySuspendAlways всегда возвращает значение false. Предположим теперь, что функция await_ready может возвращать значения true или false случайным образом, а задание не запускается в явном виде. Напомним: когда функция await_ready возвращает значение true, следом за ней автоматически вызывается сразу функция await_resume, минуя функцию await_suspend.

Автоматическое возобновление работы

```
1   // startJobWithAutomaticResumption.cpp
2
3   #include <coroutine>
4   #include <functional>
5   #include <iostream>
6   #include <random>
7
8   std::random_device seed;
9   auto gen = std::bind_front(std::uniform_int_distribution<>(0, 1),
10                             std::default_random_engine(seed()));
11
12  struct MySuspendAlways {
```

```
13      bool await_ready() const noexcept {
14          std::cout << "        MySuspendAlways::await_ready\n";
15          return gen();
16      }
17
18      bool await_suspend(std::coroutine_handle<> handle) const noexcept {
19          std::cout << "        MySuspendAlways::await_suspend\n";
20          handle.resume();
21          return true;
22      }
23
24      void await_resume() const noexcept {
25          std::cout << "        MySuspendAlways::await_resume\n";
26      }
27  };
28
29  struct Job {
30      struct promise_type;
31      using handle_type = std::coroutine_handle<promise_type>;
32      handle_type coro;
33
34      Job(handle_type h): coro(h){}
35
36      ~Job() {
37          if ( coro ) coro.destroy();
38      }
39
40      struct promise_type {
41          auto get_return_object() {
42              return Job{handle_type::from_promise(*this)};
43          }
44
45          MySuspendAlways initial_suspend() {
46              std::cout << "    Job prepared" << '\n';
47              return {};
48          }
49
50          std::suspend_always final_suspend() noexcept {
51              std::cout << "    Job finished" << '\n';
52              return {};
53          }
54
55          void return_void() {}
56          void unhandled_exception() {}
57
58      };
59  };
```

```
60
61 Job performJob() {
62     co_await std::suspend_never();
63 }
64
65 int main() {
66     std::cout <<  “Before jobs” << ‘\n’;
67
68     performJob();
69     performJob();
70     performJob();
71     performJob();
72
73     std::cout <<  “After jobs\n”;
74 }
```

Прежде всего отметим, что сопрограмма теперь называется performJob и запускается автоматически. В строке 9 объявлен генератор случайных чисел 0 или 1. В основу его работы положен стандартный генератор по умолчанию, проинициализированный начальным значением, из системного источника. Благодаря функции std::bind_front[1] этот стандартный генератор можно подставить в объект класса std::uniform_int_distribution, отвечающий за равномерное распределение в заданном диапазоне, и получить вызываемый объект gen, который можно далее использовать как функцию.

В этом примере снова используется стандартный контроллер ожидания вместо самодельного MySuspendNever – однако тип MySuspendAlways остаётся и используется в качестве типа возвращаемого значения функции initial_suspend (строка 46). Функция await_ready, объявленная в строке 13, возвращает значение логического типа. Когда оно равно true, управление передаётся прямо в фунцию await_resume (строка 25), в противном случае сопрограмма приостанавливается и, следовательно, управление передаётся в функцию await_suspend (строка 18). В отличие от предыдущего примера, эта функция получает дескриптор сопрограммы и пробуждает её (строка 20). Помимо логического значения true, она могла бы также не возвращать ничего, т. е. иметь тип void.

Представленный на следующем рисунке результат выполнения программы подтверждает это описание. Всякий раз, когда функция await_ready возвращает значение true, вызывается функция await_resume, в противном случае вызывается функция await_suspend. За выполнением программы можно наблюдать на сайте Compiler Explorer[2].

---

[1] https://en.cppreference.com/w/cpp/utility/functional/bind_front.
[2] https://godbolt.org/z/8b1Y14.

```
Before jobs
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
After jobs
```

Автоматическое пробуждение сопрограммы

Попробуем усовершенствовать эту программу так, чтобы пробуждающаяся сопрограмма продолжала выполнение в отдельном потоке.

6.7.3. Автоматическое пробуждение сопрограммы в отдельном потоке

Следующая программа представляет собой модификацию предыдущей

Автоматическое пробуждение сопрограммы в отдельном потоке

```
1  // startJobWithResumptionOnThread.cpp
2
3  #include <coroutine>
4  #include <functional>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <vector>
9
10 std::random_device seed;
11 auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
12                            std::default_random_engine(seed()));
13
```

```
14 struct MyAwaitable {
15     std::jthread& outerThread;
16 
17     bool await_ready() const noexcept {
18         auto res = gen();
19         if (res) std::cout << " (executed)\n";
20         else std::cout << " (suspended)\n";
21         return res;
22     }
23 
24     void await_suspend(std::coroutine_handle<> h) {
25         outerThread = std::jthread([h] { h.resume(); });
26     }
27 
28     void await_resume() {}
29 };
30 
31 
32 struct Job{
33     static inline int JobCounter{1};
34     Job() {
35         ++JobCounter;
36     }
37 
38     struct promise_type {
39         int JobNumber{JobCounter};
40 
41         Job get_return_object() { return {}; }
42 
43         std::suspend_never initial_suspend() {
44             std::cout << "    Job " << JobNumber << " prepared on thread "
45                       << std::this_thread::get_id();
46             return {};
47         }
48 
49         std::suspend_never final_suspend() noexcept {
50             std::cout << "    Job " << JobNumber << " finished on thread "
51                       << std::this_thread::get_id() << '\n';
52             return {};
53         }
54 
55         void return_void() {}
56         void unhandled_exception() { }
57     };
58 };
59 
60 Job performJob(std::jthread& out) {
61     co_await MyAwaitable{out};
62 }
63 
64 int main() {
65     std::vector<std::jthread> threads(8);
```

```
66      for (auto& thr: threads) performJob(thr);
67  }
```

Основное отличие этой программы от предыдущей составляет новый контроллер ожидания `MyAwaitable`, используемый в сопрограмме `performJob` (строка 61). Объект-сопрограмма, получающийся в результате выполнения функции `performJob`, напротив, довольно прост. Его функции-члены `initial_suspend` (строка 43) и `final_suspend` (строка 49) возвращают предопределённый в стандарте контроллер ожидания `std::suspend_never`. Кроме того, обе функции возвращают целочисленный номер текущего задания и идентификатор потока, в котором оно сейчас выполняется. На следующем рисунке видно, какая сопрограмма запускается немедленно, а какая приостанавливается. Наличие идентификатора потока позволяет убедиться, что приостановленная сопрограмма продолжает выполнение в другом потоке. За выполнением программы можно наблюдать, например, в среде Wandbox[1].

```
Job 1 prepared on thread 140434982274944 (executed)
Job 1 finished on thread 140434982274944
Job 2 prepared on thread 140434982274944 (suspended)
Job 3 prepared on thread 140434982274944 (suspended)
Job 4 prepared on thread 140434982274944 (suspended)
Job 2 finished on thread 140434877310720
Job 5 prepared on thread 140434982274944 (executed)
Job 5 finished on thread 140434982274944
Job 6 prepared on thread 140434982274944 (suspended)
Job 7 prepared on thread 140434982274944 (suspended)
Job 3 finished on thread 140434868918016
Job 8 prepared on thread 140434982274944 (executed)
Job 8 finished on thread 140434982274944
Job 4 finished on thread 140434860525312
Job 6 finished on thread 140434852132608
Job 7 finished on thread 140434843739904
```

Автоматическое пробуждение сопрограммы в отдельном потоке

Разберём подробнее последовательность передачи управления в этой программе. В строке 65 создаётся восемь сконструированных по умолчанию объектов-потоков, затем каждый из них передаётся по ссылке в сопрограмму `performJob` (строка 60). Потом эта ссылка передаётся в конструктор контроллера ожидания `MyAwaitable` в строке 61. В зависимости от значения, которое возвращает функция `await_ready`, т. е. значения переменной `res` (строка 18), полученного от генератора случайных значений, сопрограмма продолжает своё выполнение (если это значение есть `true`) или приостанавливается (в противном случае). В случае приостановки вызывается функция `await_sus-`

1 https://wandbox.org/permlink/skHgWKF0SYAwp8Dm.

pend (строка 24). Благодаря присваиванию по ссылке в строке 25 выполнение сопрограммы продолжается во вновь созданном потоке. Время жизни потока, в котором выполняется сопрограмма, должно превышать время её выполнения. Именно поэтому потоки объявлены в области видимости главной функции.

6.8. Краткие итоги

- Сумму элементов вектора можно вычислять множеством различных способов. Это можно делать последовательно или параллельно, с максимальной или минимальной степенью совместного доступа к данным. Показатели производительности при этом разительно отличаются.
- Потокобезопасная инициализация объекта-одиночки представляет собой классический пример для применения различных средств потокобезопасного доступа к общим переменным. Есть ряд более или менее эффективных способов решить эту задачу.
- Разобрано, как, взяв за основу простую программу, шаг за шагом улучшать её, ослабляя упорядочивание операций над памятью. Показано, как проверять каждый шаг этой оптимизации с помощью инструментального средства CppMem. CppMem – это интерактивный инструмент, позволяющий исследовать поведение небольших фрагментов кода с точки зрения используемых в языке C++ моделей памяти.
- Стандарт C++ 20 содержит множество средств синхронизации потоков. Так, можно использовать переменные условия, атомарные флаги std::atomic_flag, атомарные переменные логического типа std::atomic<bool>, семафоры. Их быстродействие сравнивается на примере игры в пинг-понг.
- Благодаря новому ключевому слову co_return становится возможным реализовать жадный, ленивый фьючерс, а также фьючерс, работающий в отдельном потоке. Примеры программ выводят подробные сообщения, что облегчает понимание принципа их работы.
- Ключевое слово co_yield позволяет легко создавать бесконечные потоки данных. Разобран ряд примеров с модификациями и обобщениями потока данных, в частности конечные потоки и потоки-шаблоны.
- Разобраны примеры передачи управления при выполнении сопрограмм, основанные на операторе co_await, в том числе сопрограмма, пробуждающая сама себя в том же или в отдельном потоке.

7. Будущее языка С++

Эта глава посвящена вероятному будущему языка С++, стандарту С++ 23. Изложение здесь не может быть столь же строгим, как в других частях книги. Этому есть две причины. Во-первых, не все из описанных здесь средств войдут в окончательную редакцию стандарта С++ 23. Во-вторых, даже если то или иное средство в стандарт войдёт, его интерфейс наверняка претерпит заметные изменения. Это предостережение относится прежде всего к исполнителям (англ. *executors*). Эта книга постоянно дорабатывается, так что доступный в сети оригинал отражает текущее состояние вносимых в стандарт предложений.

Задача главы довольно проста: дать читателю представление о перспективных средствах параллельного программирования, которыми может вскоре пополниться язык.

Средства параллельного программирования, запланированные в стандарт С++ 23

7.1 Исполнители

Исполнители призваны стать основными блоками, из которых строится процесс выполнения программы; их роль можно сравнить с той, которую в языке С++ уже играют аллокаторы и контейнеры. Предполагается, что исполнителями будут пользоваться функции наподобие `std::async` (см. раздел 3.9.2), параллельные алгоритмы из стандартной библиотеки (глава 4), продолжения

фьючерсов (раздел 7.2.1.1.2), функция-член `run` блока заданий (раздел 7.4), а также функции `post`, `dispatch`, `defer` из предполагаемого будущего модуля для работы с сетью[1]. Да и вообще говоря, исполнение – одно из основополагающих понятий программирования. Однако в нынешнем стандарте нет единого способа управления способом выполнения функции. Рассмотрим пример, с которого начинается предложение к стандарту P0761[2].

Различные реализации параллельного цикла

```
void parallel_for(int facility, int n, function<void(int)> f) {
  if(facility == OPENMP) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i) {
      f(i);
    }
  }
  else if(facility == GPU) {
    parallel_for_gpu_kernel<<<n>>>(f);
  }
  else if(facility == THREAD_POOL) {
    global_thread_pool_variable.submit(n, f);
  }
}
```

Представленная здесь функция `parallel_for` обладает рядом недостатков.

- Простая, казалось бы, функция оказывается сложной в поддержке, причём сложность поддержки растёт с добавлением новых механизмов параллельного выполнения.
- Ветки условного оператора по-разному ведут себя с точки зрения синхронизации. Механизм OpenMP[3] может блокировать выполнение до тех пор, кока не закончат работу дочерние потоки, графический процессор обычно работает асинхронно, а пул потоков может как блокировать выполнение, так и не блокировать. Недостаточная синхронизация потоков может привести к гонке данных или к мёртвой блокировке. В лучшем случае программа может оказаться в состоянии гонки.
- Функция навязывает вызывающему контексту слишком жёсткие ограничения. Например, нет никакого способа передать в неё свой пул потоков взамен глобального.

7.1.1. Долгий путь исполнителя

В октябре 2018 года было написано несколько предложений касательно исполнителей, принцип их работы до сих пор требует уточнения по ряду аспектов. Эта глава основана главным образом на предложении о прин-

[1] https://en.cppreference.com/w/cpp/experimental.
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf.
[3] https://en.wikipedia.org/wiki/OpenMP.

ципе устройства исполнителей P0761[1] и на формальных описаниях P0433[2] и P1244[3]. У этих предложений есть неплохие шансы войти в стандарт C++ 23. В этой главе также рассмотрим относительно новое предложение P1055[4] о «скромных» исполнителях.

7.1.2. Что такое исполнитель

Прежде всего нужно ответить на вопрос, что собой представляет рассматриваемая сущность. Исполнитель – это набор правил, определяющих, *где*, *когда* и *как* запускать вызываемую единицу выполнения.

- Где. Вызываемая единица может выполняться на центральном процессоре, передаваться для выполнения на внешний или вспомогательный процессор – от этого зависит, каким образом можно получить результат вычисления.
- Когда. Вызываемую сущность можно выполнить немедленно или запланировать её выполнение на более позднее время.
- Как. Выполнение может происходить на центральном процессоре, графическом процессоре или посредством векторизации.

Если говорить более формально, исполнитель – это набор свойств, связанных с некоторой функцией выполнения (см. раздел 7.1.6).

7.1.2.1. Свойства исполнителя

Следующие свойства можно связать с исполнителем двумя способами: механизм `execution::require` требует непременного выполнения свойства, а механизм `execution::prefer` означает, что в процессе выполнения кода следует предпочесть данное свойство, если это возможно. Перечень свойств приведён ниже[5].

- Направленность: функция выполнения может работать по принципу «выстрели и забудь» (`execution::oneway`), вернуть фьючерс (`execution::twoway`) или вернуть продолжение (`execution::then`).
- Кардинальность: функция выполнения может создать одного агента выполнения (`execution::single`) или множество таких агентов (`execution::bulk`).
- Блокировка: выполнение функции может блокировать или не блокировать текущий поток. Имеется три стратегии блокирования: `execution::blocking::never` запрещает блокировку, `execution::blocking::possibly` разрешает, а `execution::blocking::always` требует её.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf.

[2] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html.

[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1244r0.html.

[4] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf.

[5] Исполнители, предлагаемые к включению в стандарт, уже присутствуют в коллекции библиотек Boost и проходят проверку практикой. См. https://www.boost.org/doc/libs/1_77_0/doc/html/boost_asio/reference.html. – *Прим. перев.*

- Способ продолжения: задание может выполняться в клиентском вызывающем потоке (`execution::continuation`) или нет (`execution::not_continuation`).
- Вероятность будущего задания: это свойство определяет, насколько вероятно появление задания в будущем (свойство `execution::outstanding_work::tracked` означает, что появление задания в будущем ожидается, а свойство `execution::outstanding_work::untracked` – что это маловероятно).
- Гарантии массового продвижения вперёд: какие имеются гарантии совокупного продвижения вперёд множества создаваемых агентов выполнения: `execution::bulk_sequenced_execution`, `execution::bulk_parallel_execution` и `execution::bulk_unsequenced_execution`.
- Гарантии размещения по потокам: должен ли каждый агент выполнения назначаться на отдельный поток (режим `execution::new_thread_execution_mapping`) или нет (режим `execution::thread_execution_mapping`).
- Аллокатор: к исполнителю может быть привязан объект, ответственный за управление памятью.

Программист может также добавлять собственные свойства.

Исполнители как строительные блоки

Поскольку исполнители предполагается сделать основными строительными блоками процесса выполнения программы, средства параллельного программирования в будущем стандарте языка C++ должны сильно зависеть от них. Это в первую очередь касается расширенных фьючерсов, сетевых расширений, описанных в документе N4734[1], параллельных алгоритмов стандартной библиотеки, о которых речь шла в главе 4, а также новых средств: защёлок, барьеров, сопрограмм, транзакционной памяти и блоков заданий.

7.1.3. Первые примеры

7.1.3.1. Использование исполнителя

Разберём ряд примеров, иллюстрирующих использование исполнителей.

7.1.3.1.1. Асинхронное обещание

Асинхронное выполнение обещания посредством исполнителя

```
// как-либо получить объект-исполнитель
my_executor_type my_executor = ...

// запустить асинхронное выполнение
auto future = std::async(my_executor, [] {
    std::cout << "Hello world, from a new execution agent!" << '\n';
});
```

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf.

7.1.3.1.2. Обход элементов контейнера

Обход элементов контейнера с использованием исполнителя

```
// как-либо получить объект-исполнитель
my_executor_type my_executor = ...

// запустить параллельный обход контейнера
std::for_each(std::execution::par.on(my_executor),
    data.begin(), data.end(), func);
```

7.1.3.1.3. Сетевое соединение с использованием системного исполнителя

Использование системного исполнителя для приёма соединения

```
// как-либо получить объект для приёма входящих соединений
tcp::acceptor my_acceptor = ...

// выполнить асинхронную операцию для приёма соединения
acceptor.async_accept(
    [](std::error_code ec, tcp::socket new_connection)
    {
        ...
    });
```

7.1.3.1.4. Сетевое соединение с использованием явно заданного исполнителя

Использование системного исполнителя для приёма соединения

```
// как-либо получить объект для приёма входящих соединений
tcp::acceptor my_acceptor = ...

// получить объект-исполнитель, связанный с пулом потоков
auto my_thread_pool_executor = ...

// выполнить асинхронную операцию для приёма соединения
acceptor.async_accept(
  std::experimental::net::bind_executor(my_thread_pool_executor,
    [](std::error_code ec, tcp::socket new_connection)
    {
        ...
    }
  ));
```

Функция `bind_executor` из экспериментального модуля для поддержки сети позволяет подставить в асинхронную функцию сетевого взаимодействия пользовательский объект-исполнитель. В данном примере обработчик соединения выполнит лямбда-функцию в пуле потоков.

7.1.3.2. Получение исполнителя

Есть различные способы получить объект-исполнитель.

7.1.3.2.1. Получение исполнителя из статического пула потоков

Получение исполнителя из статического пула потоков

```
// создать пул на 4 потока
static_thread_pool pool(4);

// получить объект-исполнитель, связанный с пулом потоков
auto exec = pool.executor();

// использовать исполнителя для запуска продолжительного задания
auto task1 = long_running_task(exec);
```

7.1.3.2.2. Получение исполнителя из параллельной политики выполнения

Получение исполнителя из параллельной политики выполнения

```
// получить объект-исполнитель, связанный с параллельной политикой
auto par_exec = std::execution::par.executor();

// использовать исполнителя для запуска продолжительного задания
auto task2 = long_running_task(par_exec);
```

7.1.3.2.3. Системный исполнитель по умолчанию

Исполнитель по умолчанию обычно предполагает создание нового потока для выполнения задания. Этот исполнитель используется в случае, если никакой другой исполнитель не задан явно.

7.1.3.2.4. Применение адаптера к существующему исполнителю

Применение адаптера к существующему исполнителю

```
// получить объект-исполнитель, связанный с пулом потоков
auto exec = pool.executor();

// обернуть исполнителя в адаптер, выводящий диагностические сообщения
logging_executor<decltype(exec)> logging_exec(exec);

// отсортировать контейнер под управлением исполнителя с адаптером
std::sort(std::execution::par.on(logging_exec),
    my_data.begin(), my_data.end());
```

Исполнитель `logging_executor` в этом примере представляет собой адаптер поверх исполнителя, связанного с пулом потоков.

7.1.4. Цели разработки исполнителей

Ниже перечислены цели, которыми руководствовались авторы документа P1055[1], формулируя понятие исполнителя.

1. **Дозируемость** – возможность управлять балансом между затратами на передачу вызываемого объекта и его размером.

---

[1] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf.

2. **Гетерогенновть** – возможность запускать вызываемый объект в разнородных средах (таких как графический процессор или внешний процессор) и получать назад результат вычислений.
3. **Упорядочиваемость** – возможность задавать порядок, в котором запускаются вызываемые объекты. Сюда относятся такие гарантии порядка выполнения, как стек (LIFO – last in, first out), очередь (FIFO – first in, first out), очередь с приоритетами, расписание или последовательное выполнение.
4. **Управляемость** – возможность назначить вызываемый объект на определённый вычислительный ресурс, отложить или отменить выполнение.
5. **Продолжаемость** – поддержка сигналов, позволяющих управлять асинхронным выполнением вызываемого объекта. Эти сигналы позволяют оповещать о готовности результата, о возникновении ошибки, а также о желании вызывающего клиента прервать выполнение задания. Также должны быть возможны явный запуск вызываемого объекта и отмена запуска запланированного задания.
6. **Многослойность** – иерархическое строение исполнителей должно поддерживать добавление свойств, не внося лишнюю сложность в простые сценарии использования.
7. **Удобство** как со стороны разработчика исполнителей, так и со стороны автора прикладного кода, пользующегося исполнителями, составляет особенно важную цель.
8. **Комбинируемость** – программисту должна быть предоставлена возможность расширять исполнителей свойствами, не предусмотренными в стандарте.
9. **Минималистичность** – в понятие исполнителя, реализованное в стандарте языка, не должны входить никакие детали, которые можно было бы добавить извне, реализовав в некоторой сторонней библиотеке.

7.1.5. Терминология

В документе-предложении P0761[1] определен ряд новых терминов, относящихся к выполнению вызываемого объекта.

- **Выполняющий ресурс** – экземпляр аппаратного или программного ресурса, способный выполнять вызываемый объект. Примерами выполняющего ресурса могут служить векторный модуль процессора или система выполнения, управляющая множеством потоков. Такие выполняющие ресурсы, как центральный или графический процессор, гетерогенны, так как их возможности и ограничения сильно различаются от экземпляра к экземпляру.
- **Контекст выполнения** – программный объект, представляющий определённый набор выполняющих ресурсов и агентов выполнения.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf.

Типичными примерами могут служить пул потоков, а также распределённая или гетерогенная вычислительная система.

- **Агент выполнения** – единица выполнения внутри определённого контекста выполнения, связанная с одним вызовом вызываемого объекта на выполняющем ресурсе. Примерами могут быть поток на центральном процессоре или единица выполнения на графическом процессоре.
- **Исполнитель** – программный объект, связанный с определённым контекстом выполнения. Он предоставляет одну или более функций для создания агента выполнения для заданного вызываемого объекта (например, функционального объекта или функции).

7.1.6. Функции выполнения

Как отмечено в предыдущем разделе, исполнитель предоставляет одну или более функций, создающих агента выполнения для некоторого вызываемого объекта. Исполнитель должен поддерживать по меньшей мере шесть следующих функций.

Функции выполнения, обязательные для любого исполнителя

Имя	Кардинальность	Направленность
`execute`	Единичная	Односторонняя
`twoway_execute`	Единичная	Двусторонняя
`then_execute`	Единичная	Продолжение
`bulk_execute`	Множественная	Односторонняя
`bulk_twoway_execute`	Множественная	Двусторонняя
`bulk_then_execute`	Множественная	Продолжение

Каждая функция выполнения обладает двумя характеристиками: кардинальностью и направленностью.

- Кардинальность означает, сколь много агентов выполнения может создаваться для выполнения одного вызываемого объекта:
  - единичная (single) предполагает создание единственного агента;
  - множественная (bulk) означает возможность создавать группу агентов выполнения.
- Направленность:
  - односторонняя (oneway) означает, что данные передаются в один конец, от вызывающего контекста в вызываемый объект; последний не возвращает результата;
  - двусторонняя (twoway) означает, что вызывающий контекст требует возврата значения от выполняемого объекта; для ожидания значения может использоваться фьючерс;
  - продолжение (then) означает, что в создаваемом агенте выполнения будет выполняться фьючерс, который начнёт своё выполнение после того, как получит результат выполнения некоторого другого фьючерса.

Все перечисленные функции принимают в качестве аргумента вызываемый объект. В последующих разделах дадим неформальное описание этих функций.

7.1.6.1. Единичная кардинальность

Единичная кардинальность весьма очевидна. Однонаправленная функция выполнения запускает задание по принципу «выстрели и забудь» и не возвращает значения. Она ведёт себя подобно фьючерсам из раздела 3.9.2.2, за исключением того, что не блокирует выполнение автоматически в деструкторе фьючерса. Двунаправленная функция выполнения возвращает фьючерс, через который впоследствии можно забрать результат вычислений. Это роднит её с объектом-обещанием `std::promise`, который несёт в себе дескриптор для доступа к соответствующему фьючерсу. Функция выполнения с направленностью `then` присоединяет продолжение к существующему фьючерсу `pred`. Она возвращает новый фьючерс, а соответствующий агент выполнения запускается только тогда, когда фьючерс `pred` сообщает о готовности своего результата.

7.1.6.2. Множественная кардинальность

Случай множественной кардинальности сложнее. Эти функции создают группу агентов выполнения, каждый из которых выполняет одну и ту же вызываемую сущность `f`, передаваемую в функцию выполнения первым параметром. Второй параметр функции выполнения – целое число, максимальное количество агентов выполнения, которым может быть поручена работа. При этом вызываемый объект `f` должен первым параметром принимать индекс текущего агента. Если функция выполнения двунаправленна, её следующий параметр – фабрика результатов, задача которой состоит в том, чтобы из результатов запуска `f` на отдельных агентах выполнения собрать результат вычисления в целом. Последний аргумент функции выполнения – фабрика общего параметра; она создаёт некоторый объект, который затем передаётся каждому агенту выполнения.

В случае функции `bulk_then_execute` вызываемый объект `f` принимает фьючерс-предшественник в качестве параметра. Вызываемый объект `f` принимает общий параметр и фьючерс-предшественник по ссылке, так как ни один агент выполнения не должен владеть этими объектами.

7.1.6.3. Проверка требований к исполнителю

Каким способом удостовериться, что исполнитель поддерживает ту или иную функцию выполнения? В частном случае, когда тип исполнителя известен заранее, известны и все поддерживаемые им функции, например:

Использование однонаправленного единичного исполнителя

```
void concrete_context(const my_oneway_single_executor& ex)
{
    auto task = ...;
```

```
    ex.execute(task);
}
```

В общем же случае для проверки функциональных возможностей исполнителя можно воспользоваться функцией `execution::require`:

Запрос двунаправленного единичного исполнителя

```
template<class Executor>
void generic_context(const Executor& ex)
{
    auto task = ...;

    // ensure .toway_execute() is available with execution::require()
    execution::require(ex, execution::single,
        execution::twoway).toway_execute(task);
}
```

7.1.7. Простой пример использования

Основываясь на документе-предложении P0443R3[1], можно написать программу, демонстрирующую различные сценарии использования исполнителя.

Пример использования исполнителей

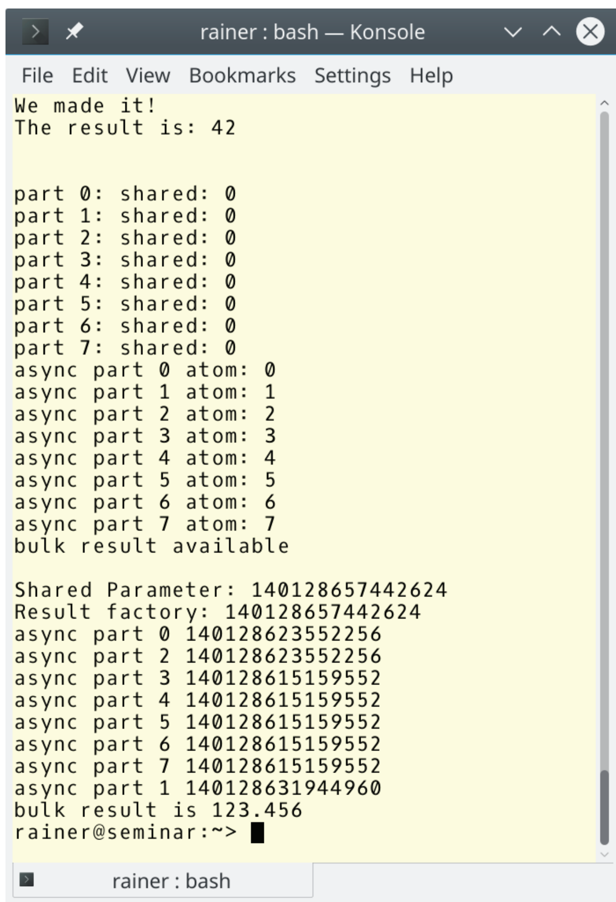
```
1  // executor.cpp
2
3  #include <atomic>
4  #include <experimental/thread_pool>
5  #include <iostream>
6  #include <utility>
7
8  namespace execution = std::experimental::execution;
9  using std::experimental::static_thread_pool;
10 using std::experimental::executors_v1::future;
11
12 int main() {
13     static_thread_pool pool{4};
14     auto ex = pool.executor();
15
16     // One way, single.
17     ex.execute([]{ std::cout << “We made it!” << std::endl; });
18
19     std::cout << std::endl;
20
21     // Two way, single.
22     future<int> f1 = ex.twoway_execute([]{ return 42; });
23     f1.wait();
24     std::cout << “The result is: “ << f1.get() << std::endl;
```

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0443r5.html.

```
25
26     std::cout << std::endl;
27
28     // One way, bulk.
29     ex.bulk_execute([](int n, int& sha){
30         std::cout << "part " << n << ": " << "shared: " << sha <<  "\n";
31         }, 8,
32         []{ return 0; }
33     );
34
35     std::cout << std::endl;
36
37     // Two way, bulk, void result.
38     future<void> f2 = ex.bulk_twoway_execute(
39         [](int n, std::atomic<short>& m){
40             std::cout << "async part " << n ;
41             std::cout << " atom: " << m++ << std::endl;
42         }, 8,
43         []{},
44         []{
45             std::atomic<short> atom(0);
46             return std::ref(atom);
47         }
48     );
49     f2.wait();
50     std::cout << "bulk result available" << std::endl;
51
52     std::cout << std::endl;
53
54     // Two way, bulk, non-void result.
55     future<double> f3 = ex.bulk_twoway_execute(
56         [](int n, double&, int &){
57             std::cout << "async part " << n << " ";
58             std::cout << std::this_thread::get_id() << std::endl;
59         }, 8,
60         []{
61             std::cout << "Result factory: "
62                       << std::this_thread::get_id() << std::endl;
63             return 123.456; },
64         []{
65             std::cout << "Shared Parameter: "
66                       << std::this_thread::get_id() << std::endl;
67             return 0; }
68     );
69     f3.wait();
70     std::cout << "bulk result is " << f3.get() << std::endl;
71 }
```

В этой программе используется исполнитель, связанный с пулом на четыре потока (строки 13 и 14). В строках 17 и 22 применены функции выполнения с единичной кардинальностью: для однонаправленного и двунаправленного (то есть возвращающего результат) заданий.

Остальные функции выполнения в этом примере (в строках 29, 38 и 55) имеют множественную кардинальность. Каждая из них создаёт по восемь агентов (см. строки 31, 42 и 59). В первом случае вызываемый объект выводит на печать порядковый номер агента и общее значение `sha`, которое создаётся фабрикой в строке 32. Второй случай, когда вызывается функция `bulk_twoway_execute`, интереснее. Хотя фабрика результатов не возвращает значения, общее состояние представляет собой атомарную переменную `atom`. Каждый агент увеличивает её значение на единицу (строка 41). Последний вызов функции выполнения (строки 55–68) возвращает значение 123,456 благодаря фабрике результатов. Довольно интересно посмотреть, сколько потоков участвует в выполнении лямбда-функции, а также в работе фабрики общего параметра и фабрики результатов. Выводимый программой текст свидетельствует о том, что фабрики общего параметра и результатов работают в одном и том же потоке, тогда как агенты могут запускаться в различных потоках.



Пример применения исполнителей

7.2. Расширенные фьючерсы

7.2.1. Техническая спецификация

7.2.1.1. Обновлённое понятие фьючерса

Название «расширенный фьючерс» объясняется просто. Во-первых, интерфейс класса `std::future` предлагается расширить; во-вторых, предлагается добавить в стандарт ряд новых функций для создания специфических разновидностей фьючерсов, которые можно было бы состыковывать между собой. Начнём с первого аспекта.

В технической спецификации предлагается расширить тип фьючерса тремя новыми функциями-членами:

- конструктор, который принимает в качестве аргумента фьючерс, завёрнутый во фьючерс (`future<future<T>>`), и извлекает его из обёртки, снимая один уровень вложенности[1];
- предикат `is_ready`, позволяющий узнать, готово ли общее состояние;
- функция `then`, которая присоединяет к фьючерсу продолжение[2].

Начнём же рассмотрение с того обстоятельства, что в технической спецификации предлагается различать валидность (valid) и готовность (ready) состояния фьючерса.

7.2.1.1.1. Валидность и готовность

Напомним, что под состоянием имеется в виду объект данных, в котором фьючерс формирует результат своей работы.

- Валидность означает: состояние, которым фьючерс владеет совместно с объектом-обещанием, имеется в наличии. Это условие не всегда выполняется, так как фьючерс может быть создан посредством конструктора по умолчанию, без объекта-обещания.
- Готовность означает, что фьючерс уже поместил результат своей работы в объект-состояние.

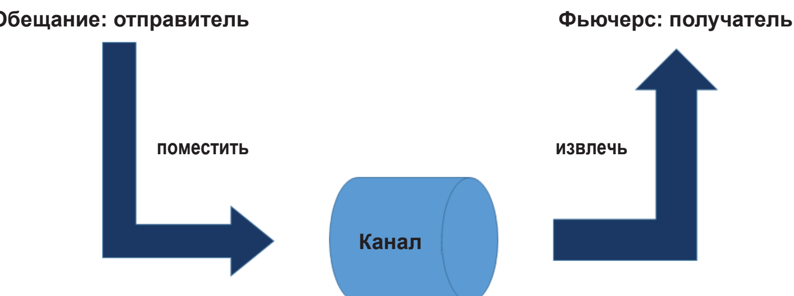
Таким образом, валидность есть непременное условие готовности: не может быть готов фьючерс, если он не валиден.

Вспомним удобную умозрительную модель обещания и фьючерса как двух концов канала, по которому передаются данные.

---

[1] Здесь можно заметить прямую связь с понятием монады, заимствованным из теории категорий и нашедшим применение в языках функционального программирования – особенно ярким примером может служить язык Haskell. Конструктор, снимающий один уровень вложенности, представляет собой аналог естественного преобразования μ из определения монады или функции `join` из языка Haslell. – *Прим. перев.*

[2] Это прямой аналог операции связывания `>>=` в языке Haskell. – *Прим. перев.*



Задания как каналы передачи данных

Теперь различие между валидностью и готовностью фьючерса выглядит естественным. Фьючерс валиден, если существует канал данных, связывающий его с обещанием. Фьючерс готов, если обещание уже выполнено, то есть если данные помещены на вход канала.

Разберёмся теперь с функцией-членом `then` и с продолжениями фьючерса.

7.2.1.1.2. Продолжение фьючерса

Функция-член `then` позволяет присоединить фьючерс к концу другого фьючерса. Кроме того, часто встречается ситуация, когда фьючерс упакован в другой фьючерс, то есть результатом работы фьючерса становится новый фьючерс. В этом случае особый конструктор помогает преобразовать вложенный фьючерс в одноуровневый, производящий только окончательный результат.

Предложение N3721

Перед тем как показать пример кода, нужно сказать несколько слов о предложении N3721[1]. Большая часть настоящего раздела основана на этом документе, озаглавленном «Улучшения типа `std::future<T>` и связанных с ним программных интерфейсов», в том числе и примеры кода. Удивительно, но авторы документа часто опускали в конце программы вызов функции-члена `get`, позволяющей получить из фьючерса результат его работы. В примерах ниже такой вызов добавлен, полученное из него значение сохранено в переменную. Также исправлены некоторые опечатки.

Продолжение фьючерса

```
1  #include <future>
2  using namespace std;
3
4  int main() {
5      future<int> f1 = async([]() { return 123; });
6      future<string> f2 = f1.then([](future<int> f) {
7          return to_string(f.get()); // этот вызов get() не блокирует
8      });
9
```

[1] https://isocpp.org/files/papers/N3721.pdf.

```
10     auto myResult= f2.get();
11 }
```

Между двумя вызовами функции-члена `get`, в строках 7 и 10, имеется тонкое различие. Как отмечено в комментарии к коду, первый вызов никогда не блокирует выполнение[1], тогда как вызов в строке 10 должен дождаться, пока не будет готов результат цепочки фьючерсов. Это справедливо как для сколь угодно длинных цепочек продолжений вида `f1.then(...).then(...).then(...).then(...)`, так и для композиции расширенных фьючерсов. Блокирующим оказывается только вызов функции `get` для получения окончательного результата.

7.2.1.2. Средства асинхронного выполнения

Не так много можно пока что сказать о предполагаемых расширениях функции `std::async` типа `std::promise` и о пакетах заданий `std::package_task`. Следует лишь отметить, что все они, согласно предложениям к будущему стандарту, должны работать с расширенными фьючерсами.

Композиция фьючерсов – более увлекательная тема. В новом стандарте могут появиться средства для сочленения асинхронных заданий между собой.

7.2.1.3. Создание новых фьючерсов

Ожидается, что в стандарт C++ 23 войдут четыре новые функции для создания особенных фьючерсов: `std::make_ready_future`, `std::make_execptional_future`, `std::when_all` и `std::when_any`. Рассмотрим сначала первые две.

7.2.1.3.1. Функции для создания фьючерсов с готовым результатом

Функции `std::make_ready_future` и `std::make_execptional_future` создают фьючерсы, результат которых известен наперёд. Первая создаёт фьючерс, возвращающий фиксированное значение, а вторая – фьючерс, бросающий исключение. Хотя на первый взгляд это может показаться удивительным, в таких функциях есть смысл. В стандарте C++ 11 для создания фьючерса с заранее известным значением требовался бы объект-обещание. Необходим он даже в том случае, когда общее состояние уже имеется в наличии. Рассмотрим пример применения функции.

Создание фьючерса с заранее известным результатом

```
future<int> compute(int x) {
    if (x < 0) return make_ready_future<int>(-1);
```

[1] Вызов функции `get` в строке 7 относится к фьючерсу `f`, переданному в продолжение в качестве аргумента. По определению, продолжение запускается только тогда, когда предшествующий фьючерс завершил работу и подготовил результат. Поэтому на момент, когда выполнение достигает строки 7, во фьючерсе гарантировано наличие результата, и функции `get` не требуется блокировать выполнение в ожидании результата. – *Прим. перев.*

```
    if (x == 0) return make_ready_future<int>(0);
    future<int> f1 = async([]() { return do_work(x); });
    return f1;
}
```

Здесь результат вычисляется посредством обещания только при `x > 0`.

Отметим мимоходом, что эти две функции напоминают функцию `return` из языка Haskell, которая оборачивает значение в монаду. Перейдём к функциям, которые служат для сочленения фьючерсов.

7.2.1.3.2. Композиции фьючерсов

У функций `std::when_all` и `std::when_any` много общего. Посмотрим внимательнее на их объявления.

Объявления функций `std::when_all` и `std::when_any`

```
template <class InputIt>
auto when_any(InputIt first, InputIt last)
  -> future<when_any_result<
      std::vector<typename std::iterator_traits<InputIt>::value_type>>>;

template <class... Futures>
auto when_any(Futures&&... futures)
  -> future<when_any_result<std::tuple<std::decay_t<Futures>...>>>;

template <class InputIt>
auto when_all(InputIt first, InputIt last)
  -> future<std::vector<typename std::iterator_traits<InputIt>::value_type>>;

template <class... Futures>
auto when_all(Futures&&... futures)
  -> future<std::tuple<std::decay_t<Futures>...>>;
```

Обе функции принимают в качестве аргументов либо пару итераторов по контейнеру фьючерсов, либо произвольное число фьючерсов. В перегрузке с итераторами все фьючерсы должны быть одного типа, тогда как в перегрузке с произвольным числом аргументов фьючерсы могут иметь различные типы – можно даже смешивать типы `std::future` и `std::shared_future`.

Тип возвращаемого значения этих функций различен для двух перегрузок. В любом случае возвращается фьючерс. В случае перегрузки с парой итераторов возвращается фьючерс с завёрнутым в него вектором фьючерсов. Перегрузка с произвольным числом аргументов возвращает фьючерс с кортежем фьючерсов.

При этом первая функция возвращает фьючерс, который становится готов тогда, когда готовы все фьючерсы, поступившие на вход функции, а вторая – когда готов хотя бы один из них. Следующие два примера иллюстрируют применение этих двух функций.

7.2.1.3.3. Пример применения функции std::when_all

Композиция фьючерсов функцией std::when_all

```
1  #include <future>
2
3  using namespace std;
4
5  int main() {
6      shared_future<int> shared_future1 = async(
7          [] { return intResult(125); });
8      future<string> future2 = async(
9          []() { return stringResult("hi"); });
10
11     future<tuple<shared_future<int>, future<string>>> all_f =
12         when_all(shared_future1, future2);
13
14     future<int> result = all_f.then(
15         [](future<tuple<shared_future<int>, future<string>>> f){
16             return doWork(f.get());
17         });
18
19     auto myResult = result.get();
20 }
```

Фьючерс all_f представляет собой композицию двух фьючерсов разных типов: shared_future1 и future2. Фьючерс result представляет собой продолжение – он начинает выполняться, когда готов фьючерс all_f, который, в свою очередь, становится готов, когда готовы два фьючерса-компонента. Последняя строка кода забирает из него результат вычислений.

7.2.1.3.4. Пример применения функции std::when_any

Композиция фьючерсов функцией std::when_any

```
1  #include <future>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      vector<future<int>> v{ .... };
8      auto future_any = when_any(v.begin(), v.end());
9
10     when_any_result<vector<future<int>>> result = future_any.get();
11
12     future<int>& ready_future = result.futures[result.index];
13
14     auto myResult = ready_future.get();
15 }
```

Результат работы фьючерса future_any помещается в переменную result. В объекте шаблонного типа when_any_result содержится целочисленный индекс того из входных фьючерсов, который первым сообщил о своей готов-

ности. Если не пользоваться типом when_any_result, пришлось бы опрашивать каждый фьючерс о его готовности, что было бы слишком утомительно.

Наконец, из контейнера фьючерсов в переменную ready_future извлекается тот, что готов, а из него – окончательный результат всего вычисления.

В документе P0701r1[1] отмечается, что ни фьючерсы, уже вошедшие в стандарт ранее, ни экспериментальные фьючерсы из технической спецификации[2] «не являются столь обобщёнными, выразительными и мощными, как им следовало бы быть». Кроме того, обновлённые фьючерсы должны быть унифицированы с исполнителями как основным инструментом управления выполнением всевозможных заданий.

7.2.2. Унифицированные фьючерсы

Рассмотрим недостатки, общие для фьючерсов – как стандартных, так и экспериментальных, предлагаемых в технической спецификации.

7.2.2.1. Недостатки фьючерсов

В упоминавшемся выше документе дано превосходное описание недочётов, присущих фьючерсам.

7.2.2.1.1. Привязка фьючерсов и обещаний к потокам

В стандарте C++ 11 определён единственный вид исполнителя – поток, т. е. тип std::thread. Следовательно, фьючерсы в их нынешнем виде неотделимы от потоков. Ситуация несколько меняется в стандарте C++ 17 с появлением параллельных алгоритмов. Ещё более существенных изменений следует ожидать с появлением в стандарте исполнителей, которые можно использовать для тонкой настройки способа выполнения фьючерса. Например, можно указать, что тот или иной фьючерс должен выполняться в отдельном потоке, через пул потоков или в вызывающем потоке.

7.2.2.1.2. Способ выполнения продолжений

Рассмотрим несложное продолжение – такое, как в следующем примере.

Фьючерс с продолжением

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f) {
    return to_string(f.get());
});
```

Попробуем ответить на вопрос: где именно должно выполняться продолжение? Сейчас возможны следующие варианты:

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html.
[2] http://en.cppreference.com/w/cpp/experimental/concurrency.

1) потребитель: выполнением продолжения занимается тот выполняющий агент, в котором работает потребитель, запросивший результат фьючерса `f2`;
2) производитель: продолжение может выполняться в том же агенте, который выполнял фьючерс `f1`;
3) автоматический выбор: если в момент присоединения продолжения фьючерс `f1` готов, продолжение выполняется на стороне потребителя, в противном случае – на стороне производителя;
4) отдельный поток: для выполнения продолжения может создаваться новый поток.

Первые два варианта обладают существенным недостатком: они блокируют выполнение. В первом случае потребитель блокирует свой поток до тех пор, пока не будет готов производитель. Во втором же случае производитель блокирует поток до тех пор, пока не закончит свою работу продолжение.

Ниже показаны примеры передачи исполнителей по цепочке продолжений, взятые из документа P0707r1[1].

Передача исполнителя по цепочке продолжений

```
auto i = std::async(thread_pool, f).then(g).then(h);
// f, g и h выполняются в пуле потоков

auto i = std::async(thread_pool, f).then(g, gpu).then(h);
// f выполняется в пуле потоков, g и h - на графическом процессоре

auto i = std::async(inline_executor, f).then(g).then(h);
// выражение h(g(f())) вызывается на агенте, выполняющем вызов
```

7.2.2.1.3. Передача фьючерса в продолжение слишком громоздка

В продолжение передаётся объект-фьючерс, а не выработанное им значение, что делает синтаксис довольно запутанным. Вот пример такого правильного, но излишне многословного кода.

Продолжение фьючерса

```
std::future<int> f1 = std::async([]() { return 123; });
std::future<std::string> f2 = f1.then([](std::future<int> f) {
    return std::to_string(f.get());
});
```

Предположим теперь, что результат предыдущего фьючерса может передаваться в продолжение по значению – или что функция `to_string` имеет перегрузку для типа `std::future<int>`. В результате получим более компактный код.

Продолжение фьючерса с передачей значения

```
std::future<int> f1 = std::async([]() { return 123; });
std::future<std::string> f2 = f1.then(std::to_string);
```

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html.

7.2.2.1.4. Композиции фьючерсов возвращают слишком громоздкие типы

Из примеров в разделе 7.2.1.3.2 видно, что типы, возвращаемые функциями `when_all` и `when_any`, слишком сложны в использовании.

7.2.2.1.5. Условную блокировку в деструкторе фьючерса нужно устранить

Фьючерсы, работающие по принципу «выстрели и забудь», выглядят привлекательно, но обладают большим недостатком. Фьючерс, созданный функцией `std::async`, ждёт в своём деструкторе, пока не выполнится связанное с ним обещание. То, что на первый взгляд кажется параллельным, выполняется последовательно. Согласно документу P0707r1, такое поведение чревато ошибками и недопустимо. Напомним, что аномалии, связанные с принципом «выстрели и забудь», рассматривались в разделе 3.9.2.2.

7.2.2.1.6. Нужна совместимость непосредственно данных и будущих значений

В стандарте C++ 11 нет удобного способа для создания фьючерсов – приходится начинать с объекта-обещания.

Создание фьючерса в нынешнем стандарте

```
std::promise<std::string> p;
std::future<std::string> fut = p.get_future();
p.set_value("hello");
```

Эта ситуация может измениться с принятием в стандарт предложений, ныне оформленных в технической спецификации.

Создание фьючерса средствами предполагаемого будущего стандарта

```
std::future<std::string> fut = make_ready_future("hello");
```

Если же в будущем появится возможность использовать будущие значения наряду с непосредственно данными, это заметно упростит работу программистов.

Совместное использование непосредственно данных и будущих значений

```
bool f(std::string, double, int);

std::future<std::string> a = /* ... */;
std::future<int> c = /* ... */;

std::future<bool> d1 = when_all(a, make_ready_future(3.14), c).then(f);
// f(a.get(), 3.14, c.get())

std::future<bool> d2 = when_all(a, 3.14, c).then(f);
// f(a.get(), 3.14, c.get())
```

Ни одна из желаемых синтаксических форм `d1` и `d2` не реализуема даже средствами технической спецификации. Следует надеяться, этот пробел будет закрыт в будущем.

7.2.2.2. Пять новых концептов

В предложении к стандарту 1054R0[1] определяются пять новых концептов, касающихся фьючерсов и обещаний.

- FutureContinuation – вызываемый объект, которому в качестве аргумента передаётся значение или исключение от фьючерса.
- SemiFuture – операция, которая принимает на вход исполнителя и возвращает объект, удовлетворяющий концепту ContinuableFuture (пример: `f = sf.via(exec)`).
- ContinuableFuture – конкретизация концепта SemiFuture; тип фьючерса, к экземплярам которого можно присоединить одно продолжение, удовлетворяющее концепту FutureContinuation, которое выполняется исполнителем, связанным с фьючерсом.
- SharedFuture – конкретизация концепта ContinuableFuture, позволяющая присоединять к экземпляру множество продолжений из концепта FutureContinuation.
- Promise – объект, связанный с фьючерсом и переводящий фьючерс в состояние готовности с вычисленным значением или исключением в качестве результата.

В документе приводится также объявление этих концептов.

Предлагаемые новые концепты для фьючерсов и обещаний

```
template <typename T>
struct FutureContinuation
{
    // По меньшей мере она из двух следующих перегрузок
    auto operator()(T value);
    auto operator()(exception_arg_t, exception_ptr exception);
};

template <typename T>
struct SemiFuture
{
    template <typename Executor>
    ContinuableFuture<Executor, T> via(Executor&& exec) &&;
};

template <typename Executor, typename T>
struct ContinuableFuture
{
    template <typename RExecutor>
    ContinuableFuture<RExecutor, T> via(RExecutor&& exec) &&;

    template <typename Continuation>
    ContinuableFuture<Executor, auto> then(Continuation&& c) &&;
};
```

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html.

```
template <typename Executor, typename T>
struct SharedFuture
{
    template <typename RExecutor>
    ContinuableFuture<RExecutor, auto> via(RExecutor&& exec);

    template <typename Continuation>
    SharedFuture<Executor, auto> then(Continuation&& c);
};

template <typename T>
struct Promise
{
    void set_value(T value) &&;

    template <typename Error>
    void set_exception(Error exception) &&;

    bool valid() const;
};
```

Из этих объявлений можно сделать следующие выводы:

- экземпляру концепта FutureContinuation можно передать либо значение какого-то типа, либо исключение. Этот экземпляр представляет собой вызываемый объект, который принимает на вход значение или исключение и делает их результатом работы фьючерса;
- все виды фьючерсов (SemiFuture, ContinuableFuture, SharedFuture) обладают функцией-членом via, которая принимает аргумент-исполнитель и возвращает экземпляр концепта ContinuableFuture. Функция via позволяет превращать один тип фьючерса в другой, подставляя различные исполнители;
- функция-член then есть только у концептов ContinuableFuture и SharedFuture. Эта функция принимает экземпляр концепта FutureContinuation и возвращает экземпляр концепта ContinuableFuture;
- концепт SharedFuture поддерживает семантику неисключительного владения фьючерсом, его экземпляры могут копироваться;
- концепт Promise позволяет установить значение или исключение в качестве результата работы фьючерса.

7.2.2.3. Направления дальнейшей работы

В предложении 1054R2 некоторые вопросы остаются открытыми:

- гарантии продвижения вперёд для фьючерсов и обещаний;
- требования синхронизации для использования фьючерсов и обещаний в непараллельных выполняющих агентах;
- интероперабельность фьючерсов и обещаний;
- раскрытие вложенных фьючерсов: как вида future<future<T>>, так и более сложных;
- функции when_all, when_any и when_n;
- семантика функции async.

7.3. Транзакционная память

Понятие транзакционной памяти основано на понятии транзакции, развитом в теории баз данных. Транзакционная память упрощает работу с потоками в двух аспектах. Во-первых, полностью устраняется опасность гонок данных и мёртвых блокировок; во-вторых, транзакции легко компоновать в более крупные транзакции.

Транзакция – это операция, обладающая следующими четырьмя свойствами: атомарность, согласованность, изолированность, устойчивость (англ. *atomicity*, *consistency*, *isolation*, *durability* – *ACID*). Эти свойства должны выполняться и для транзакционной памяти в языке C++, за исключением требования устойчивости, которое означает гарантию сохранения результата операции в долговременном хранилище, и потому не имеет смысла для оперативной памяти. Из четырёх свойств остаются три, которые нужно рассмотреть подробнее.

7.3.1. Требования ACI(D)

Что означают атомарность, согласованность и изолированность в применении к блоку операторов?

Атомарный блок операторов

```
atomic {
    statement1;
    statement2;
    statement3;
}
```

Атомарность: в любой момент времени потокам программы либо виден результат выполнения всех операторов этого блока, либо не виден результат ни одного из них.

Согласованность: система всегда находится в согласованном состоянии. Все выполняющиеся в ней транзакции вполне упорядочены между собой в линейную последовательность.

Изолированность: любая транзакция выполняется в полной изоляции от всех остальных транзакций.

Как добиться выполнения этих трёх свойств? Транзакция запоминает начальное состояние данных и затем выполняется без всякой синхронизации над временной копией этих данных. Если во время выполнения её операторов обнаруживается конфликт (т. е. если какая-то другая транзакция успела за это время изменить состояние данных), выполнение текущей транзакции прерывается, и её выполнение начинается заново. Если же к моменту завершения транзакции состояние данных остаётся неизменным, транзакция подтверждается – данные в общем доступе обновляются из локальной копии. Для обнаружения конфликтов обычно применяются ссылки на помеченные состояния (англ. *tagged state reference*).

Транзакцию можно представить себе как действие, которое подготавливается без гарантии осуществления и осуществляется при условии, что не изменились начальные условия, исходя из которых происходила подготовка. В противоположность мьютексам здесь имеет место оптимистический подход. Применяя транзакцию, надеются, что она завершится без конфликта, но готовы к повторной попытке в случае неудачи. В основу мьютексов положен пессимистический подход: перед входом в критическую секцию поток убеждается, что ни один другой поток не находится в ней, а войдя в неё, блокирует такую возможность для остальных потоков.

В будущих стандартах языка C++ могут появиться два средства поддержки транзакционной памяти: синхронизированные блоки и атомарные блоки[1].

7.3.2. Синхронизированные и атомарные блоки

До сих пор мы рассматривали лишь транзакции. Теперь речь пойдёт о синхронизированных и атомарных блоках. Строго говоря, синхронизированные блоки транзакциями не являются, так как внутри них может выполняться код, небезопасный с точки зрения транзакции. Примером транзакционно-небезопасного кода может служить вывод на консоль – действие, которое нельзя отменить. По этой причине синхронизированные блоки называют ещё ослабленными блоками.

7.3.2.1. Синхронизированные блоки

Синхронизированные блоки ведут себя так, будто находятся под единой глобальной блокировкой. Это значит, что все синхронизированные блоки в процессе выполнения программы образуют линейную последовательность. В частности, все изменения общих данных, сделанные некоторым синхронизированным блоком, видны в следующем синхронизированном блоке. Между синхронизированными блоками имеет место следующее отношение: подтверждение транзакции, выполненной одним блоком, синхронизировано с началом другой транзакции.

Синхронизированные блоки не могут зайти в тупик именно в силу глобальной линейной упорядоченности. Если классический мьютекс защищает критическую секцию программы, глобальная упорядоченность синхронизированного блока защищает программу в целом. По этой причине следующая программа обладает вполне определённым поведением.

Синхронизированный блок

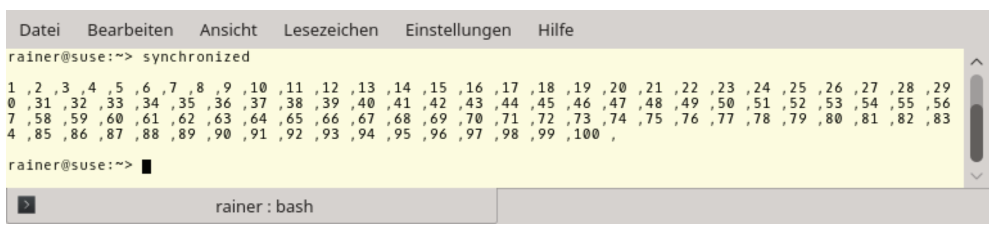
```
1  // synchronized.cpp
2
3  #include <iostream>
```

[1] Заслуживает упоминания реализация программной транзакционной памяти на языке C++, созданная Александром Граниным, см. исходный код https://github.com/graninas/cpp_stm_free, также доклад «Функциональный подход к Software Transactional Memory» https://youtu.be/VHZPcz8HwZs. – *Прим. перев.*

```
4  #include <vector>
5  #include <thread>
6
7  int i= 0;
8
9  void increment(){
10   synchronized{
11     std::cout << ++i << " ,";
12   }
13 }
14
15 int main(){
16   std::cout << std::endl;
17
18   std::vector<std::thread> vecSyn(10);
19   for(auto& thr: vecSyn)
20     thr = std::thread([]{ for(int n = 0; n < 10; ++n) increment(); });
21   for(auto& thr: vecSyn) thr.join();
22
23   std::cout << "\n\n";
24 }
```

Хотя переменная `i` объявлена в глобальной области видимости и функция `increment`, модифицирующая её, вызывается по десять раз из десяти параллельных потоков, программа обладает вполне определённым поведением. Операции над переменной `i`, как и операции вывода на консоль, выполняются в глобальном линейном порядке. Это и есть основное свойство синхронизированных блоков.

Выполнение программы даёт ожидаемый результат. Значения переменной `i` выводятся в возрастающей последовательности через запятую.



Наращивание переменной в синхронизированном блоке

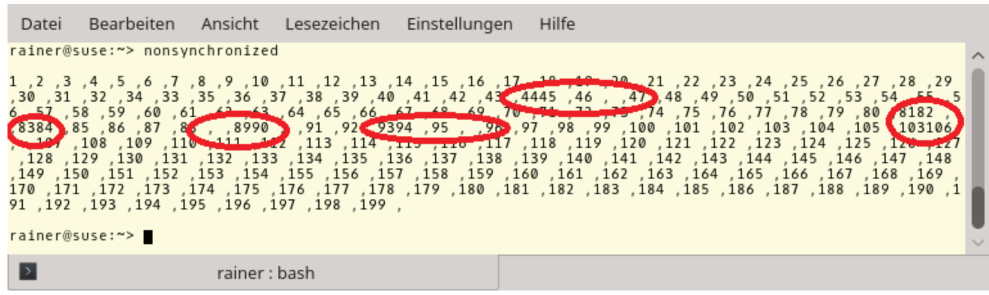
Как теперь обстоят дела с гонками данных? Возможны ли они в программе, где есть синхронизированные блоки? Небольшая модификация предыдущей программы позволяет убедиться, что возможны – если в синхронизированные блоки поместить не все опасные операции.

Гонка данных в программе с синхронизированными блоками

```
1  // nonsynchronized.cpp
2
3  #include <chrono>
```

```
4  #include <iostream>
5  #include <vector>
6  #include <thread>
7
8  using namespace std::chrono_literals;
9  using namespace std;
10
11 int i= 0;
12
13 void increment(){
14   synchronized{
15     cout << ++i << " ,";
16     this_thread::sleep_for(1ns);
17   }
18 }
19
20 int main(){
21   cout << endl;
22
23   vector<thread> vecSyn(10);
24   vector<thread> vecUnsyn(10);
25
26   for(auto& thr: vecSyn)
27     thr = thread([] {for(int n = 0; n < 10; ++n) increment();});
28   for(auto& thr: vecUnsyn)
29     thr = thread([] {for(int n = 0; n < 10; ++n) cout << ++i << " ,";});
30
31   for(auto& thr: vecSyn) thr.join();
32   for(auto& thr: vecUnsyn) thr.join();
33
34   cout << "\n\n";
35 }
```

Синхронизированный блок теперь притормаживает выполнение на одну наносекунду. В то же время делается попытка доступа к потоку вывода `std::cout` из потоков без синхронизированного блока. Таким образом, к глобальной переменной и потоку вывода обращаются 20 потоков, причём половина из них – без синхронизации. Результат работы программы показан на рисунке, проблема отчётливо видна.



Гонка данных в программе с синхронизированными блоками

Некоторые места в выводимом программой тексте обведены красным. Это места, где по меньшей мере два потока одновременно осуществляют вывод в поток `std::cout`. Стандарт C++ 11 гарантирует, что каждая отдельная операция вывода в поток происходит атомарным образом, поэтому здесь большой проблемы нет[1]. Хуже то, что переменную `i` могут изменять одновременно несколько потоков. Это гонка данных. Следовательно, поведение программы не определено. Если присмотреться к напечатанному программой тексту внимательнее, гонку данных легко увидеть в действии. После того как 20 потоков увеличивают переменную на единицу каждый по 10 раз, окончательное значение переменной должно было быть равно 200, но вместо этого оказывается равным 199. Это значит, что одно из промежуточных значений счётчика было перезаписано в ходе гонки. Таким образом, для надёжной защиты от гонки данных мало использовать синхронизированные блоки где-то в программе – нужно делать это везде, где происходит обращение к глобальному состоянию.

Линейный порядок выполнения характерен не только для синхронизированных, но и для атомарных блоков.

7.3.2.2. Атомарные блоки

Транзакции можно выполнять не только в синхронизированных, но и в атомарных блоках. Атомарные блоки могут выступать в трёх формах: `atomic_noexcept`, `atomic_commit` и `atomic_cancel`. Наименования вполне точно отражают стратегию обработки исключений в этих блоках:

- `atomic_noexcept` – если внутри блока возникает необработанное исключение, вызывается функция `std::abort`, и программа аварийно завершается;
- `atomic_cancel` – в случае исключения транзакция подтверждается;
- `atomic_commit` – в случае необработанного исключения в общем случае программа аварийно завершается. Однако если это исключение транзакционно-безопасно, транзакция прерывается, возвращается к исходному состоянию, а исключение пробрасывается наружу.

К транзакционно-безопасным относятся исключения типов `std::bad_alloc`[2], `std::bad_array_length`[3], `std::bad_array_new_length`[4], `std::bad_cast`[5], `std::bad_typeid`[6], `std::bad_exception`[7], `std::exception`[8] и всех типов, производных от них.

---

[1] За исключением некрасиво выведенного текста из-за того, что операции вывода числа и запятой из разных потоков часто оказываются перемешаны между собой. – *Прим. перев.*

[2] http://en.cppreference.com/w/cpp/memory/new/bad_alloc.

[3] https://www.cs.helsinki.fi/group/boi2016/doc/cppreference/reference/en.cppreference.com/w/cpp/memory/new/bad_array_length.html.

[4] http://en.cppreference.com/w/cpp/memory/new/bad_array_new_length.

[5] http://en.cppreference.com/w/cpp/types/bad_cast.

[6] http://en.cppreference.com/w/cpp/types/bad_typeid.

[7] http://en.cppreference.com/w/cpp/error/bad_exception.

[8] http://en.cppreference.com/w/cpp/error/exception.

7.3.3. Транзакционно-безопасный и транзакционно-небезопасный код

Функцию можно объявить транзакционно-безопасной или транзакционно-небезопасной с помощью ключевого слова `transaction_safe` и атрибута `transaction_unsafe` соответственно.

Транзакционно-безопасная и транзакционно-небезопасная функции

```
int transactionSafeFunction() transaction_safe;
[[transaction_unsafe]] int transactionUnsafeFunction();
```

Транзакционная безопасность является частью типа функции. Что она означает? Согласно документу N4265, транзакционно-безопасная функция должна удовлетворять следующим ограничениям:

- функция не должна иметь параметров со спецификатором `volatile` и не должна содержать переменных с таким спецификатором;
- функция не должна содержать транзакционно-небезопасных операторов;
- в теле функции не должны использоваться конструкторы и деструкторы класса, имеющего нестатические члены со спецификатором `volatile`.

Конечно, эта характеристика транзакционной безопасности не может считаться исчерпывающей, так как в ней использовано понятие транзакционно-небезопасного оператора. Подробности можно узнать в документе N4265.

7.4. Блоки заданий

Блоки заданий представляют собой средство для организации параллельного выполнения заданий в соответствии с широко известным шаблоном разветвления и слияния. Они уже включены в техническую спецификацию по расширенной поддержке параллельного программирования в языке C++ версии 2[1]. Поэтому вполне возможно, что скоро они будут включены в очередную версию стандарта языка.

Кто изобрёл поддержку блоков заданий в языке C++? Как компания Microsoft, разработавшая библиотеку параллельных шаблонов PPL[2], так и компания Intel с библиотекой строительных блоков для программирования потоков[3] участвовали в выработке предложения N4441[4]. Кроме того, компания Intel использовала свой опыт разработки библиотеки Cilk Plus[5].

---

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4742.html.

[2] https://en.wikipedia.org/wiki/Parallel_Patterns_Library.

[3] https://en.wikipedia.org/wiki/Threading_Building_Blocks.

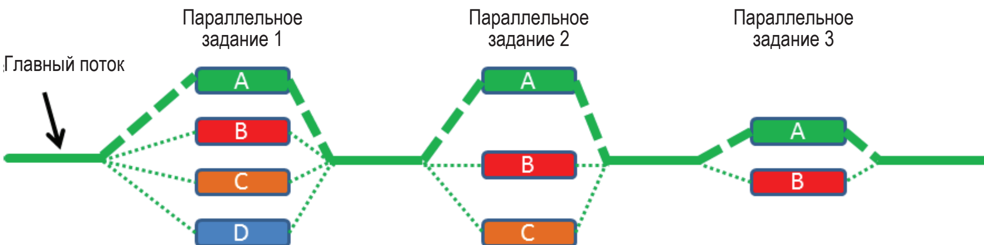
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf.

[5] https://en.wikipedia.org/wiki/Cilk.

Сущность шаблона разветвления и слияния объяснить довольно просто.

7.4.1. Разветвление и слияние

Шаблон разветвления и слияния проще всего объяснить с помощью графической схемы.



Шаблон разветвления и слияния

Как работает изображённая здесь система? Главный поток создаёт блок заданий с помощью функции `define_task_block` или `define_task_block_restore_thread`. Внутри блока могут создаваться новые задания или может происходить ожидание существующих заданий. В роли точки синхронизации выступает конец блока. Создание заданий в блоке – это фаза разветвления; синхронизация заданий в конце блока представляет собой фазу слияния. Это общее описание и впрямь выглядит очень простым. Рассмотрим теперь фрагмент программного кода.

Определение блока заданий

```
1  template <typename Func>
2  int traverse(node& n, Func && f){
3      int left = 0, right = 0;
4      define_task_block(
5           [&](task_block& tb){
6              if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
7              if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
8          }
9      );
10     return f(n) + left + right;
11 }
```

Шаблонная функция `traverse` вызывает функцию `f` для каждого узла дерева. Функция `define_task_block` создаёт блок заданий. Лямбда-функция, переданная в качестве аргумента, составляет первое задание блока. Она может создавать и добавлять в блок новые задания. Именно это происходит с левым и правым поддеревьями: их обход происходит в отдельных заданиях, находящихся под управлением этого блока. Окончание блока является также и точкой синхронизации – в этой точке блок ожидает завершения всех своих заданий.

Каркас HPX для высокопроизводительных параллельных вычислений

Приведённый выше пример взят из документации к каркасу HPX[1] (High-Performance ParallelX), который представляет собой систему широкого назначения для поддержки разнообразных параллельных и распределённых вычислений любого масштаба. В составе каркаса HPX уже реализованы многие из описанных в этой главе перспективных средств, предполагающихся к включению в будущие стандарты языка C++.

Для создания блока заданий предлагаются две функции. Рассмотрим подробнее различие между ними.

7.4.2. Две функции для создания блоков заданий

Тонкое различие межу функциями `define_task_block` и `define_task_block_restore_thread` состоит в том, что вторая из них, в отличие от первой, гарантирует, что потоком, выполняющимся после завершения блока заданий, станет тот же поток, который этот блок создал[2]. Пусть дан следующий код.

Две функции для создания блоков заданий

```
1  ...
2  define_task_block([&](auto& tb){
3    tb.run([&]{[] func(); });
4    define_task_block_restore_thread([&](auto& tb){
5      tb.run([&]([]{ func2(); });
6      define_task_block([&](auto& tb){
7        tb.run([&]{ func3(); }
8      });
9      ...
10     ...
11   });
12   ...
13   ...
14 });
15 ...
16 ...
```

Для создания вложенного блока заданий в строке 4 использована функция, сохраняющая поток. Поэтому гарантируется, что операторы в строках 12 и 13 будут выполняться в том же потоке, который выполнил строки 3 и 4. Напротив, для создания блока заданий третьего уровня вложенности в строке 6, а также для создания внешнего блока в строке 2 используется функция, не гарантирующая сохранение потока. Поэтому операторы в строках 15 и 16

---

[1] http://stellar.cct.lsu.edu/projects/hpx/.

[2] Подчеркнём данное обстоятельство, идущее в разрез с кажущимися самоочевидными принципами работы программ: некоторые функции управления блоками заданий могут возвращать управление не тому потоку, из которого они вызваны. Код, содержащий вызовы этих функций, может внезапно попасть в поток с другими значениями потоковых переменных и с иными захваченными мьютексами. – *Прим. перев.*

могут выполняться в ином потоке, нежели оператор в строке 1; подобным же образом могут отличаться потоки, выполняющие строку 5 и строки 9, 10.

7.4.3. Интерфейс

Интерфейс блока заданий минималистичен. Пользователь не может создавать в явном виде, разрушать, копировать или перемещать объекты класса `task_block`. Создание блока заданий возможно только с помощью двух рассмотренных выше функций. Созданный блок заданий передаётся помещённой в него функции как аргумент `tb` и может использоваться внутри неё для создания новых заданий (`tb.run`) и для ожидания завершения имеющихся заданий (`tb.wait`).

Интерфейс блока заданий на примере

```
1  define_task_block([&](auto& tb){
2    tb.run([&]{ process(x1, x2) });
3    if (x2 == x3) tb.wait();
4    process(x3, x4);
5  });
```

Что делает этот фрагмент кода? В строке 2 запускается новое задание. Ему для работы нужны переменные `x1` и `x2`. В строке 4 используются также переменные `x3` и `x4`. Если выполняется условие `x2 == x3`, переменные нужно защитить от одновременного доступа. Поэтому главное задание блокируется до тех пор, пока задание, запущенное в строке 2, не завершится.

Если в блоке заданий обнаруживается исключение, выполнение блока прерывается, а наружу выбрасывается исключение типа `task_cancelled_exception`.

7.4.4. Планировщик заданий

Планировщик управляет выполняющимися потоками. Это значит, что распределение заданий по потокам перестаёт быть делом программиста. Потоки становятся лишь деталью реализации.

Есть две стратегии выполнения вновь созданного задания (в следующем описании под родителем будем понимать поток, создавший задание, а под ребёнком – новое задание).

- Похищение ребёнка – планировщик берёт задание под свой контроль и отправляет его на выполнение в какой-либо поток по своему усмотрению.
- Похищение родителя – блок заданий сам принимается за выполнение нового задания. Тем временем планировщик забирает блок заданий и решает, какому потоку его отправить на выполнение.

В предложении N4441[1] поддерживаются обе стратегии.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf.

7.5. Библиотека для векторных вычислений

К включению в стандарт предлагается также ряд средств, поддерживающих параллельные вычисления на уровне данных, т. е. векторизированные вычисления. Принцип SID (англ. *Single Instruction – Multiple Data*) означает, что одна и та же операция может выполняться над многими элементами данных параллельно. Любая современная процессорная архитектура содержит расширения для векторизации вычислений, но нынешний стандарт языка C++ не содержит средств для управления ими. В следующей таблице дан обзор различных средств для векторных вычислений.

Векторные расширения

Архитектура	Расширение	Размер регистра, бит
ARM 32/64	NEON	128
Power / PowerPC	AltiVec	128
	VSX	128
x86 / AMD 64	MMX / 3DNow	64
	SSE	128
	AVX / AVX2 / AVX-512	128 / 256 / 512

Так, например, расширение SSE[1] позволяет за одну операцию складывать четыре 32-битных целых числа. Это ровно вчетверо быстрее, чем выполнять четыре 32-битных сложения последовательно, поскольку на современных процессорах векторная операция требует такого же времени, как и скалярная.

Автоматическая векторизация

Автоматическая векторизация означает, что задача оптимизации кода под особенности конкретной архитектуры возлагается на сам компьютер. Часто на пути генерации наиболее эффективного кода встают различные препятствия. Как отмечается в руководстве разработчику[2] на языке C++, выпущенному компанией Intel, главные причины затруднений таковы:

- снижение эффективности кеш-памяти из-за произвольного порядка обращений к памяти;
- зависимости по данным между итерациями цикла;
- сложность условий цикла, когда число повторений невозможно определить до входа в цикл.

Можно назвать и иные факторы, затрудняющие автоматическую векторизацию:

- зависимость по управлению между вызовами функций из тела цикла;
- вложенность циклов (наличие цикла в теле другого цикла);
- использование потоков с мьютексами и атомарными переменными.

---

[1] https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[2] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-using-automatic-vectorization.

Расширение стандарта C++ для поддержки векторных вычислений состоит из двух специальных типов и ряда операций над ними.

7.5.1. Векторные типы данных

Предлагаемые в стандарт типы данных для поддержки векторизированных вычислений – это типы `std::simd` и `std::simd_mask`.

```
template <class T, class Abi = simd_abi::compatible<T>>
class simd;

template <class T, class Abi = simd_abi::compatible<T>>
class simd_mask;
```

Тип-параметр `T` – это тип данных, над которыми предполагается векторное вычисление. Второй тип-параметр `Abi` используется для определения количества элементов, группируемых между собой для векторной операции, и способа их хранения.

7.5.2. Интерфейс векторизированных данных

Библиотека для векторных вычислений поддерживает вспомогательные типы для выравнивания данных, арифметические операции, разнообразные свойства данных.

7.5.2.1. Вспомогательные типы-признаки

Эти типы позволяют программисту управлять хранением и выравниванием данных.

7.5.2.1.1. Типы-признаки для управления способом хранения данных

Следующие типы определяют способ упаковки элементов данных в вектор, удобный для выполнения параллельных операций.

Типы-признаки способа хранения элементов

Имя	Описание
`scalar`	Хранение единственного элемента данных
`fixed_size`	Хранение заданного числа элементов
`compatible`	Подходящий доступный механизм хранения
`native`	Наиболее эффективный механизм хранения
`max_fixed_size`	Наибольшее аппаратно поддерживаемое число элементов

7.5.2.1.2. Типы-признаки для управления выравниванием данных

Эти типы определяют способ выравнивания элементов типа `T` при копировании данных в буфер или из буфера типа `U`.

Признаки выравнивания

Имя	Описание
`element_aligned`	Данные в буфере выровнены на границу элемента
`vector_aligned`	Данные выровнены по границе вектора
`overaligned`	Данные выровнены по заданному числу битов

7.5.2.2. Выражения над значениями векторного типа

Следующие шаблоны классов позволяют строить выражения над выбранными элементами.

Средства построения выражений

Имя	Описание
`const_where_expression`	Применение немодифицирующей операции
`where_expression`	Применение модифицирующей операции
`where`	Заменяется на `const_where_expression` или `where_expression`

7.5.2.3. Приведение типов

Операции приведения типов могут применяться к векторизированным данным поэлементно, разбивать векторизированное значение на ряд значений элементарного типа или, напротив, соединять несколько элементарных значений в одно векторизированное.

Операции приведения типов

Имя	Описание
`simd_cast`, `static_simd_cast`	Поэлементное статическое преобразование
`to_fixed_size`, `to_compatible`, `to_native`	Поэлементное преобразование двоичных представлений
`split`	Разбиение одного векторизированного значения на множество элементарных
`concat`	Сборка векторизированного значения из элементарных

7.5.2.4. Алгоритмы над векторизированными значениями

Следующие векторизированные алгоритмы принимают в качестве аргументов и возвращают векторизированные значения.

Векторизированные алгоритмы

Имя	Описание
`min`	Найти наименьший в каждой паре элементов
`max`	Найти наибольший в каждой паре элементов
`minmax`	Найти наименьший и наибольший
`clamp`	Привести значение к интервалу (см. ниже)

Под приведением к интервалу понимается следующая операция. Если даны значения `lo[i]` и `hi[i]` (границы интервала) и значение `v[i]`, то результат `sim[i]` определяется следующим образом:

- если `v[i] < lo[i]`, то `sim[i] = lo[i]`;
- иначе если `hi[i] < v[i]`, то `sim[i] = hi[i]`;
- иначе `sim[i] = v[i]`.

Векторизированный алгоритм проделывает эту операцию одновременно с каждым элементарным значением, упакованным в векторное значение.

7.5.2.5. Свёртка по операции

Эти алгоритмы производят свёртку всех элементарных значений, упакованных в одно векторное, по некоторой операции.

Алгоритмы свёртки векторизированных значений

Имя	Описание
`reduce`	Общий случай свёртки по произвольной операции
`hmin`	Нахождение минимального значения
`hmax`	Нахождение максимального значения

7.5.2.6. Свёртка с маской

Следующие алгоритмы применяют операцию исключительно к тем элементарным значениям, упакованным в векторизированное значение, для которых соответствующий элемент битовой маски установлен в значение `true`.

Свёртка с маской

Имя	Описание
`all_of`, `any_of`, `none_of`, `some_of`	Проверка, выполняется ли заданный предикат для всех (некоторых, ни одного, нескольких) элементов
`popcount`	Подсчёт числа элементов, для которых предикат выполняется
`find_first_set`, `find_last_set`	Нахождение индекса первого (последнего) элемента, для которого выполняется предикат

7.5.2.7. Классы свойств

Эти классы-шаблоны и соответствующие им переменные-шаблоны позволяют во время компиляции получить параметры типов, используемых для векторных расширений.

Имя	Описание
`is_abi_tag`	Проверка, является ли тип признаком способа хранения
`is_simd`	Является ли тип типом векторизированных данных
`is_simd_mask`	Является ли тип типом маски элементов
`is_simd_flag_type`	Является ли тип типом флага

Имя	Описание
`simd_size`	Число элементов заданного типа, помещающихся в заданном векторизированном типе
`memory_alignment`	Параметры выравнивания элементов данных
`abi_for_size`	Подходящий тип данных для упаковки заданного количества элементов заданного типа

7.6. Итоги

- Предсказывать будущее нелегко – изложенное в этой главе может не осуществиться.
- Исполнитель содержит в себе набор правил относительно того, где, как и когда должен выполниться вызываемый объект. Исполнители играют роль основных блоков, из которых строится процесс выполнения сложной программы. Они определяют, должны ли её части запускаться в отдельных потоках, в пуле потоков или даже последовательно, без распараллеливания.
- Объекты-обещания и фьючерсы, появившиеся в стандарте C++ 11 для поддержки заданий, обладают богатыми возможностями, но также и рядом серьёзных ограничений. Их трудно компоновать между собой в более крупные задания. Это ограничение предлагается снять в стандарте C++ 23 за счёт так называемых расширенных фьючерсов. Например, расширенный фьючерс можно настроить таким образом, чтобы он переходил в состояние готовности после того, как станет готов единственный фьючерс-предшественник, один из нескольких предшественников или все из множества предшественников.
- Понятие транзакционной памяти основано на понятии транзакции, заимствованном из теории баз данных. Транзакция над программной памятью – это действие, обладающее первыми тремя из четырёх свойств транзакции над базой данных: атомарностью, согласованностью и изолированностью (четвёртое свойство – надёжность – для оперативной памяти выполняется всегда). В будущем стандарте могут появиться два средства поддержки транзакционной памяти: синхронизированные блоки и атомарные блоки. Оба вида блоков ведут себя так, как если бы они находились под глобальной блокировкой.
- Блоки заданий позволяют пользоваться логикой разветвления и слияния заданий. В фазе разветвления создаются новые задания, тогда как фаза слияния позволяет синхронизировать момент их завершения.
- Библиотека параллельных векторизированных вычислений позволяет воспользоваться преимуществами векторных расширений процессора, т. е. за одну машинную операцию обрабатывать несколько значений элементарного типа, упакованных в один объект векторизированного типа.

8. Шаблоны и эмпирические правила

Задача этой главы – дать читателю представление о том, что такое шаблоны[1] проектирования и для чего они бывают полезны. Данная тема будет рассматриваться неформально и главным образом через призму языка C++. Более строгое изложение можно найти в литературе по ссылкам.

Ответим сначала на главный вопрос: что такое шаблон?

> *«Шаблон проектирования представляет собой трёхстороннее привило, выражающее отношение между определённым контекстом, проблемой и решением», как писал Кристофер Александер[2].*

Если говорить менее формально, шаблон – это устоявшееся и хорошо описанное решение типовой инженерной задачи в определённой области.

8.1. История понятия

Отцом шаблонов проектирования считается процитированный выше Кристофер Александер, чьи шаблоны дружественного к человеку градостроительства, проектирования зданий и дизайна интерьеров нашли продолжение в сфере разработки программ. В 1994 году вышла знаменитая книга «Приемы объектно-ориентированного проектирования. Паттерны проектирования» коллектива авторов, известного как «Банда четырёх» (Эрик Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес). В этой книге описаны 23 шаблона объектно-ориентированной разработки программ. Эти шаблоны разбиты на три категории: порождающие, структурные шаблоны и шаблоны поведения. Тем самым вводится единый словарь для индустрии программного обеспечения. Следующие шаблоны проектирования получили наибольшее распространение.

---

[1] В русскоязычной литературе наряду с термином «шаблон» используется также заимствованное из английского языка слово «паттерн». В частности, именно оно использовано в классическом переводе книги «Банды четырёх», о которой пойдёт речь ниже. – *Прим. перев.*

[2] https://ru.wikipedia.org/wiki/Александер,_Кристофер.

- Порождающие шаблоны:
  - фабричный метод;
  - объект-одиночка.
- Структурные шаблоны:
  - адаптер;
  - мост;
  - композит;
  - декоратор;
  - фасад;
  - прокси.
- Шаблоны поведения:
  - команда;
  - итератор;
  - наблюдатель;
  - стратегия;
  - шаблонный метод;
  - посетитель.

Годом позже Фрэнк Бушман, Регина Мёнье, Ханс Ронерт, Петер Зоммерфельд и Михаэль Шталь опубликовали свою книгу «Шиблонно-ориентированная архитектура программного обеспечения: система шаблонов» (также известную как POSA), оказавшую значительное влияние на всю отрасль[1]. Эта книга положила начало серии из пяти книг. Она вышла в 1995 году и содержала три категории шаблонов: архитектурные шаблоны, шаблоны проектирования и идиомы. Многие из них прочно вошли в обиход:

- архитектурные шаблоны:
  - многослойность;
  - конвейеры и фильтры;
  - брокер;
  - модель-визуализатор-контроллер;
- шаблоны проектирования:
  - главный-дублёр;
  - издатель-подписчик;
- идиомы:
  - указатель со счётчиком владельцев.

В чём различие между этими тремя категориями? В центре внимания архитектурного шаблона находится вся программная система как целое. Шаблоны этой группы более абстрактны, чем шаблоны проектирования, которые относятся к взаимодействию подсистем. Идиомы, в свою очередь, относятся к воплощению шаблонов в конкретном языке программирования и находятся на самом низком уровне абстракции.

Из книг серии POSA со второй по пятую каждая посвящена своей теме. Вторая книга носит заглавие «Шаблоны параллельных и сетевых объектов», третья – «Шаблоны управления ресурсами», четвёртая называется «Язык

---

[1] https://www.wiley.com/WileyCDA/Section/id-406899.html.

шаблонов для распределённых вычислений» и пятая – «О шаблонах и языке шаблонов». Главы 9 и 10 настоящей книги несут на себе значительный след второго тома серии POSA.

8.2. Неоценимая польза шаблонов

Появление шаблонов внесло неоценимый вклад в развитие индустрии программного обеспечения в целом. Конечно же, это относится и к параллельному программированию в частности. Повсеместное принятие шаблонов в практику разработки принесло главным образом три следующих преимущества: установление чёткой терминологии, улучшенную документированность разработки и возможность обучения на наилучших примерах.

Вклад шаблонов в развитие терминологии состоит в том, что разработчики получили возможность пользоваться единым недвусмысленным словарём для описания своих идей. Недопонимание между разработчиками по поводу устройства и принципов функционирования программы, как и необходимость в многословных объяснениях, уходит в прошлое. Если разработчик спрашивает совета о том, как семейство подобных по назначению алгоритмов реализовать так, чтобы можно было гибко переключаться между ними во время выполнения, ответ можно дать одной фразой: использовать шаблон «Стратегия». Если разработчик знает, что означает это название, он может сразу приступить к анализу преимуществ и накладных расходов от использования данного шаблона; в противном случае он может обратиться к литературе, чтобы понять полученный совет.

Документированность разработки улучшается двояким образом. Во-первых, если новый разработчик, знакомящийся с системой, узнаёт из текстуального или графического описания, что в ней присутствует сущность под названием «наблюдатель» (англ. *observer*), он уже может немало заключить об устройстве и функционировании системы. Скажем, в системе имеются источники событий (subject), наблюдатели могут регистрироваться на получение всех или некоторых событий от определённых источников, как и отменять свою регистрацию; источники, в свою очередь, при некоторых условиях рассылают оповещения (англ. *notify*) своим наблюдателям. Во-вторых, можно просто открыть исходный код и отыскать в нём ряд слов, таких как observer, subject, notify в нашем примере.

Шаблоны – это ещё и удобный способ передачи лучшего опыта. С помощью шаблонов начинающий программист учится сразу наилучшим решениям и освобождается от необходимости повторять ошибки, с которыми был сопряжен поиск этих решений. Шаблоны проектирования – это проверенные временем решения типовых задач, а значит, и инструмент борьбы со сложностью. Описание каждого шаблона содержит сведения о том, в каких ситуациях его стоит применять, каковы последствия его применения, а также в каких известных системах он используется.

8.3. Шаблоны или эмпирические правила

Читатель мог заметить, что в заглавии настоящей главы говорится не только о шаблонах, но и об эмпирических правилах разработки. Что общего между этими понятиями и что делает их различными? Следует признать, что вопрос о том, следует тот или иной типовой подход – например, неизменяемость общих данных или чистоту функций – отнести к шаблонам или к эмпирическим правилам, может вызвать интенсивный спор с самим собой. В конечном счёте что такое шаблоны, как не детально описанные эмпирические правила. Из многочисленных споров с собой автор настоящей книги извлёк ряд общих выводов.

- Невозможно с абсолютной чёткостью разграничить эти два понятия.
- Если некоторый подход к решению типовой проблемы детально и строго описан, его стоит отнести к шаблонам.
- Если подход к решению типовой проблемы по форме похож на полезный совет и не структурирован строго, лучше считать его эмпирическим правилом.
- То, что сегодня выглядит эмпирическим правилом, может завтра стать шаблоном.

8.4. Антишаблоны

Если шаблон представляет собой концентрированное выражение положительного опыта разработки программ, то антишаблон – это описание горького урока, или, словами Эндрю Кёнига[1], «описание плохого решения задачи, повлекшего за собой плохую ситуацию». Например, если внимательно изучить старую литературу по параллельному программированию, можно обнаружить «шаблон блокировки с двойной проверкой» (double-checked lock). В нескольких словах: этот шаблон призван оптимизировать потокобезопасную инициализацию общего состояния, в роли которого часто выступает объект-одиночка[2] (singleton). Подробный разбор блокировки с двойной проверкой помещён в этой книге в главу 6, посвящённую учебным примерам, не случайно: этим подчёркивается, что непродуманное использование двойной проверки может привести к неопределённому поведению. Проблемы, связанные с двойной проверкой, оказываются в конечном счёте теми же самыми, что и известные проблемы объекта-одиночки.

Программист, раздумывающий об использовании объекта-одиночки в своей программе, должен подумать о следующих связанных с ним трудностях:

---

[1] https://en.wikipedia.org/wiki/Andrew_Koenig_(programmer).

[2] https://ru.wikipedia.org/wiki/Одиночка_%28шаблон_проектирования%29.

- во-первых и в-главных: одиночка – это глобальное состояние. По этой причине объект-одиночка чаще всего используется напрямую, а не через интерфейс. Как следствие весь код, в котором используется одиночка, получает лишнюю скрытую зависимость;
- одиночка – это статический объект. Единожды созданный, он не может быть уничтожен. Время его жизни ограничено только временем выполнения программы;
- если статический член класса (в частности, класса одиночки) зависит от статического члена иного класса, определённого в другой единице трансляции, нет никаких гарантий относительно порядка, в котором они будут проинициализированы при запуске программы. Вероятность промахнуться с инициализацией составляет 50 %;
- шаблон «одиночка» нередко используют в ситуациях, когда на самом деле вполне можно было бы использовать экземпляр класса. Некоторые разработчики применяют этот шаблон только для того, чтобы доказать своё знание шаблонов.

8.5. Итоги

Шаблоны – это хорошо документированные эмпирические правила, заключающие в компактной форме практический опыт разработки и выражающие «отношение между определённым контекстом, проблемой и решением», как сформулировал автор идеи К. Александер.

9. Шаблоны синхронизации

Главная проблема, с которой приходится иметь дело в параллельном программировании, – это изменяемое состояние в общем доступе. Тони Ван Ээрд[1] в своём докладе «Неблокирующее программирование в примерах» на конференции CppCon в 2014 году выразил это так: «Забудьте то, чему вас учили в детском саду, перестаньте делиться (изменяемым состоянием. – *Прим. перев.*)».

Изменяемость

Общий доступ	Нет	Да
Нет	Порядок	Порядок
Да	Порядок	Гонка данных

Данные в параллельных программах

Необходимым условием возникновения гонки данных является изменяемое состояние в общем доступе. Если присутствует только общий доступ или только изменяемость состояния, гонка данных возникнуть не может. Два следующих раздела посвящены именно этому: как справиться с общим доступом и как справиться с изменяемым состоянием.

9.1. Управление общим доступом

Если состояние не находится в общем доступе у разных потоков, гонка данных возникнуть не может. Отсутствие общего доступа означает, что каждый поток работает с собственными локальными переменными. Этого можно добиться копированием значений, использованием потокового класса памяти,

[1] https://github.com/tvaneerd.

а также передачей результатов вычислений через фьючерс, по защищённому каналу. Шаблоны, описанные в этом разделе, довольно очевидны, однако для полноты изложения их нужно показать и снабдить краткими пояснениями.

9.1.1. Копирование значения

Если поток получает свои аргументы в виде скопированных значений, а не по ссылке на исходные значения, нет нужды синхронизировать доступ к этим данным. Ни гонка данных, ни неприятности со временем жизни объектов в этом случае невозможны.

9.1.1.1. Гонка данных при передаче по ссылке

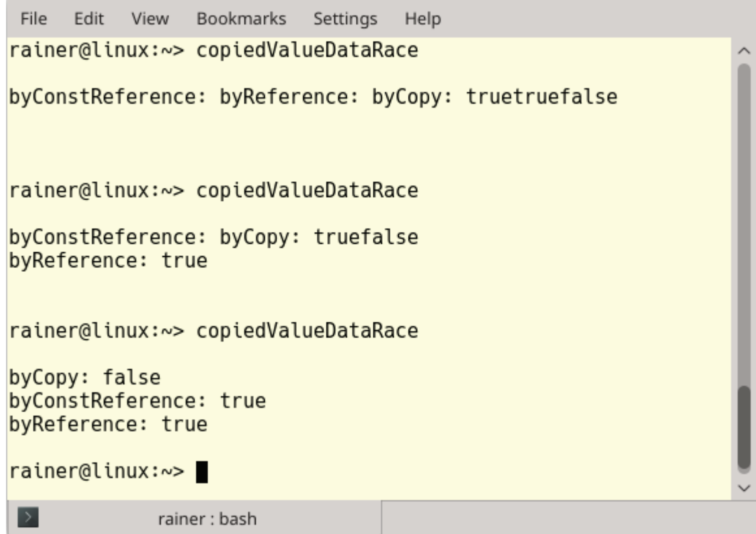
В следующей программе запускаются три потока. Один поток получает аргумент путём копирования значения, второй получает данные по ссылке, а третий – по константной ссылке.

Гонка данных при передаче по ссылке

```
1  // copiedValueDataRace.cpp
2
3  #include <functional>
4  #include <iostream>
5  #include <string>
6  #include <thread>
7
8  using namespace std::chrono_literals;
9
10 void byCopy(bool b){
11     std::this_thread::sleep_for(1ms);
12     std::cout << "byCopy: " << b << std::endl;
13 }
14
15 void byReference(bool& b){
16     std::this_thread::sleep_for(1ms);
17     std::cout << "byReference: " << b << std::endl;
18 }
19
20 void byConstReference(const bool& b){
21     std::this_thread::sleep_for(1ms);
22     std::cout << "byConstReference: " << b << std::endl;
23 }
24
25 int main(){
26     std::cout << std::boolalpha << std::endl;
27
28     bool shared{false};
29
30     std::thread t1(byCopy, shared);
31     std::thread t2(byReference, std::ref(shared));
```

```
32     std::thread t3(byConstReference, std::cref(shared));
33
34     shared = true;
35
36     t1.join();
37     t2.join();
38     t3.join();
39
40     std::cout << std::endl;
41 }
```

Каждый поток спит одну миллисекунду (строки 11, 15 и 20), перед тем как напечатать логическое значение. Различие между потоками состоит в том, что поток `t1` работает с локальной копией значения и потому не может приводить к гонке данных. Результат работы программы свидетельствует о том, что логические значения, с которыми работают потоки `t2` и `t3`, модифицируются без синхронизации.



Гонка данных при передаче по ссылке

Показанная выше программа основывается на предположении, очевидном для логического типа, но нетривиальном для более сложных типов. Передача аргументов по значению застрахована от гонки данных, если аргумент представляет собой так называемый объект-значение.

Объекты-значения

Объекты-значения – это такие объекты, сравнение которых на равенство основывается не на их индивидуальности, а только на состоянии. Объекты-значения должны быть неизменяемыми – чтобы два объекта, созданных равными, оставались равными между собой на протяжении всего времени жизни. Если объект-значение передаётся в поток, нет нужды синхронизировать доступ к нему. Следуя статье «ValueObject»

Мартина Фаулера[1], можно считать, что существуют две категории объектов: объекты-значения и объекты-ссылки.

9.1.1.1.1. Когда передача по ссылке оборачивается копированием

Читатель мог бы предположить, что поток t3 в предыдущем примере можно создавать и более простым способом, передавая непосредственно переменную shared. Программа в этом случае откомпилируется и заработает, однако то, что в ней выглядит передачей по ссылке, на самом деле работает как копирование значения. Причина в том, что к каждому аргументу, передаваемому в конструктор потока, применяется преобразователь типа std::decay[2]. Он меняет тип аргумента, превращая lvalue в rvalue, массив в указатель, а функцию – в указатель на функцию. В частности, тип ссылки он приводит к типу значения посредством преобразования std::remove_reference[3].

Проиллюстрируем сказанное следующей программой. В ней объявляется класс, для которого запрещена операция копирования.

Неявное копирование в поток, принимающий аргумент по ссылке

```
1  // perConstReference.cpp
2
3  #include <thread>
4
5  class NonCopyableClass{
6      public:
7
8      // the compiler generated default constructor
9      NonCopyableClass() = default;
10
11     // disallow copying
12     NonCopyableClass& operator = (const NonCopyableClass&) = delete;
13     NonCopyableClass (const NonCopyableClass&) = delete;
14 };
15
16 void perConstReference(const NonCopyableClass& nonCopy){}
17
18 int main(){
19     NonCopyableClass nonCopy;
20
21     perConstReference(nonCopy);
22
23     std::thread t(perConstReference, nonCopy);
24     t.join();
25 }
```

Объект nonCopy копировать нельзя. Вызов функции perConstReference с передачей этого объекта законен, так как функция принимает аргумент по константной ссылке. Однако использование той же самой функции с тем же

1 https://martinfowler.com/bliki/ValueObject.html.
2 https://en.cppreference.com/w/cpp/types/decay.
3 https://en.cppreference.com/w/cpp/types/remove_reference.

самым аргументом для создания потока заставляет компилятор GCC 6 выдать подробное сообщение об ошибке, простирающееся на триста с лишним строк, как показано на следующем рисунке.

```
File   Edit   View   Bookmarks   Settings   Help
rainer@linux:~> g++-6 perConstReference.cpp -o perConstReference -pthread 2>&1 | wc -l
366
rainer@linux:~> █
rainer : bash
```

Пространное сообщение об ошибке

При этом единственно важной в этом тексте оказывается строка, выделенная красным на следующем рисунке: «использование удалённой функции». Конструктор копирования для класса `NonCopyableClass` недоступен.

```
File   Edit   View   Bookmarks   Settings   Help
NonCopyableClass> >'
/usr/include/c++/6/type_traits:143:12:   required from 'struct std::__and_<std::is_nothrow_move_constructible<void (*)(co
nst NonCopyableClass&)>, std::is_nothrow_move_constructible<std::_Tuple_impl<1ul, NonCopyableClass> > >'
/usr/include/c++/6/tuple:218:7:   required from 'constexpr std::_Tuple_impl<_Idx, _Head, _Tail ...>::_Tuple_impl(std::_Tu
ple_impl<_Idx, _Head, _Tail ...>&&) [with long unsigned int _Idx = 0ul; _Head = void (*)(const NonCopyableClass&); _Tail
= {NonCopyableClass}]'
/usr/include/c++/6/functional:1428:41:   required from 'typename std::_Bind_simple_helper<_Func, _BoundArgs>::__type std:
:__bind_simple(_Callable&&, _Args&& ...) [with _Callable = void (&)(const NonCopyableClass&); _Args = {NonCopyableClass&}
; typename std::_Bind_simple_helper<_Func, _BoundArgs>::__type = std::_Bind_simple<void (*(NonCopyableClass))(const NonCo
pyableClass&)>]'
/usr/include/c++/6/thread:137:26:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (
&)(const NonCopyableClass&); _Args = {NonCopyableClass&}]'
perConstReference.cpp:25:45:   required from here
/usr/include/c++/6/tuple:69:35: error: use of deleted function 'NonCopyableClass::NonCopyableClass(const NonCopyableClass
&)'
  : _Head(std::forward<_UHead>(__h)) { }
                                   ^

perConstReference.cpp:13:5: note: declared here
     NonCopyableClass (const NonCopyableClass&) = delete;
     ^~~~~~~~~~~~~~~~
In file included from /usr/include/c++/6/thread:39:0,
                 from perConstReference.cpp:3:
/usr/include/c++/6/functional: In instantiation of 'std::_Bind_simple<_Callable(_Args ...)>::_Bind_simple(_Tp&&, _Up&& ..
.) [with _Tp = void (*)(const NonCopyableClass&); _Up = {NonCopyableClass&}; _Callable = void (*)(const NonCopyableClass&
); _Args = {NonCopyableClass}]':
/usr/include/c++/6/functional:1426:14:   required from 'typename std::_Bind_simple_helper<_Func, _BoundArgs>::__type std:
:__bind_simple(_Callable&&, _Args&& ...) [with _Callable = void (&)(const NonCopyableClass&); _Args = {NonCopyableClass&}
rainer : bash
```

Использование удалённой функции

9.1.1.2. Проблемы со временем жизни объектов, передаваемых по ссылке

Если поток принимает аргумент по ссылке, нужно быть предельно осторожным с отсоединением этого потока от потока-родителя. Следующая небольшая программа обладает неопределённым поведением.

Проблемы со временем жизни объектов, передаваемых по ссылке

```
1  // copiedValueLifetimeIssues.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <thread>
```

```
6
7  void executeTwoThreads(){
8      const std::string localString(“local string”);
9
10     std::thread t1([localString]{
11         std::cout << “Per Copy: “ << localString << std::endl;
12     });
13
14      std::thread t2([&localString]{
15         std::cout << “Per Reference: “ << localString << std::endl;
16     });
17
18     t1.detach();
19     t2.detach();
20 }
21
22 using namespace std::chrono_literals;
23
24 int main(){
25     std::cout << std::endl;
26
27     executeTwoThreads();
28
29     std::this_thread::sleep_for(1s);
30
31     std::cout << std::endl;
32 }
```

Функция `executeTwoThreads` запускает два потока. Оба потока отсоединяются после создания и вывоят на печать строковую переменную `localString`. Первый поток получает аргумент в виде копии значения, а второй – ссылку на локальную переменную функции `executeTwoThreads`. Для простоты в обоих случаях используется лямбда-функция.

Поскольку функция `executeTwoThreads` завершается, не дожидаясь завершения созданных ею потоков, поток `t2` работает с объектом, который может прекратить своё существование. Это приводит к неопределённому поведению. Примечательно, что компилятор GCC 6 с максимальным уровнем оптимизации `-O3` производит по-видимому работоспособный код, тогда как неоптимизированная версия терпит крах.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@linux:~> g++-6 -O3 copiedValueLifetimeIssues.cpp -o copiedValueLifetimeIssues -pthread
rainer@linux:~> copiedValueLifetimeIssues

Per Reference: local string
Per Copy: local string

rainer@linux:~> g++-6 copiedValueLifetimeIssues.cpp -o copiedValueLifetimeIssues -pthread
rainer@linux:~> copiedValueLifetimeIssues

Per Reference: Per Copy: local string
Segmentation fault (core dumped)
rainer@linux:~>
rainer : bash
```

Проблемы со временем жизни объектов, передаваемых по ссылке

9.1.1.3. Материал для дальнейшего изучения

Более подробную информацию по затронутым здесь темам можно найти в четвёртой книге из серии POSA[1].

9.1.2. Потоковая область хранения

Потоковая область хранения позволяет многочисленным потокам пользоваться своими копиями некоторых переменных так, как если бы они были глобальными. Спецификатор класса памяти `thread_local` делает переменную с глобальной областью видимости потоковой переменной. Поток может пользоваться ею без какой-либо синхронизации.

Рассмотрим типичный пример. Пусть нужно подсчитать сумму элементов вектора. Самый очевидный способ суммирования – цикл по диапазону.

```
// calculateWithLoop.cpp
...
unsigned long long sum = {};
for (auto n: randValues) sum += n;
```

Однако у компьютера, скажем, четыре процессорных ядра. Поэтому последовательную программу хочется преобразовать в параллельную.

```
// threadLocalSummation.cpp
...
thread_local unsigned long long tmpSum = 0;

void sumUp(
    std::atomic<unsigned long long>& sum,
    const std::vector<int>& val,
    unsigned long long beg, unsigned long long end)
{
    for (auto i = beg; i < end; ++i) {
        tmpSum += val[i];
    }

    sum.fetch_add(tmpSum, std::memory_order_relaxed);
}
...

std::atomic<unsigned long long> sum{};

std::thread t1(sumUp, std::ref(sum), std::ref(randValues),   0, fir);
std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
```

Цикл по диапазону помещён в функцию, каждый вызов этой функции в отдельном потоке делает четвёртую часть работы, накапливая частичную

[1] http://www.dre.vanderbilt.edu/~schmidt/POSA/POSA4/.

сумму в переменной `tmpSum`, находящейся в потоковой области. После завершения цикла накопленный частичный результат прибавляется к общему итогу `sum` посредством атомарной операции.

Различные способы суммирования элементов вектора рассматривались в разделе 6.1. Там можно найти полный текст программы с комментариями.

Использование алгоритмов из стандартной библиотеки

Программисту не нужно своими руками писать цикл, если задачу можно решить с помощью алгоритма из стандартной библиотеки. В данном примере суммирование всех элементов вектора или его части можно поручить алгоритму `std::accumulate`[1]. Начиная со стандарта C++ 17 можно также пользоваться алгоритмом `std::reduce`, который представляет собой вариацию предыдущего алгоритма с поддержкой параллельной стратегии выполнения.

9.1.2.1. Материал для дальнейшего изучения

Более подробные сведения можно найти в статье об оъектах-значениях[2] и во второй книге серии POSA[3].

9.1.3. Использование фьючерсов

Стандарт C++ 11 содержит три способа работы с фьючерсами: функцию `std::async`, класс `std::packageg_task` и работающие в паре типы `std::promise` и `std::future`. Термин «promise» (обещание) уходит корнями в 1970-е годы. Фьючерс можно представить себе как доступное только для чтения хранилище на одно значение, для заполнения которого служит доступный для записи объект – обещание. Главное с точки зрения синхронизации свойство пары «обещание–фьючерс» состоит в том, что эти два объекта связаны между собой потокобезопасным каналом.

При воплощении фьючерсов в каком-либо языке программирования нужно принять решения по следующим вопросам:

- значение фьючерса можно запрашивать неявно или явно – как это сделано в стандарте языка C++;
- фьючерс может начинать свою работу жадным или ленивым способом. В стандартной библиотеке языка C++ только функция `std::async` поддерживает ленивые вычисления посредством политик запуска.

Проиллюстрируем сказанное примером.

```
auto lazyOrEager = std::async([]{ return "LazyOrEager"; });
auto lazy = std::async(std::launch::deferred, []{ return "Lazy"; });
auto eager = std::async(std::launch::async, []{ return "Eager"; });
```

1 https://en.cppreference.com/w/cpp/algorithm/accumulate.
2 https://martinfowler.com/bliki/ValueObject.html.
3 https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/.

```
lazyOrEager.get();
lazy.get();
eager.get();
```

Если не задать в явном виде политику запуска, система сама вправе решать, каким способом выполнять асинхронное задание: жадным или ленивым. Использование политики `std::launch::async` означает создание нового потока и немедленный запуск задания. Политика `deferred`, напротив, откладывает выполнение задания до тех пор, пока его результат не будет запрошен функцией `get`. Такое обещание выполняется в потоке, запросившем результат.

Фьючерс может заблокировать выполнение, если значение обещания не готово. В стандартной библиотеке языка C++ блокироваться могут вызовы функций-членов `wait` и `get`. Кроме того, можно ожидать значения с заданным предельным временем ожидания с помощью функций `wait_for` и `wait_until`.

Существует ряд способов реализации фьючерсов: например, сопрограммы[1], генераторы[2], каналы[3].

9.1.3.1. Материал для дальнейшего изучения

Читатель, заинтересовавшийся данной темой, может изучить любые пособия, касающиеся фьючерсов и обещаний в языках программирования.

9.2. Управление изменяемым состоянием

Если программа не пытается одновременно писать и читать одни и те же данные, гонка данных возникнуть не может. Самый простой способ добиться такого состояния – пользоваться неизменяемыми значениями. Помимо этого эмпирического правила широко распространены ещё две стратегии. Во-первых, критическую секцию можно защитить блокировщиком: локальные блокировщики рассматриваются в разделе 9.2.1, а о параметризованном блокировщике речь пойдёт в разделе 9.2.2. В объектно-ориентированном программировании критическая секция обычно оформляется в виде объекта, обладающего определённым интерфейсом. Интерфейс называют потокобезопасным, если все его функции ставят блокировку на весь объект – см. раздел 9.2.3. Во-вторых, поток, вносящий изменения в данные, может послать сигнал остальным потокам, когда его работа закончена. Эту стратегию называют охраняемой приостановкой, о ней речь пойдёт в разделе 9.2.4.

---

1 https://en.wikipedia.org/wiki/Coroutine.
2 https://en.wikipedia.org/wiki/Generator_(computer_programming).
3 https://en.wikipedia.org/wiki/Channel_(programming).

9.2.1. Локальные блокировщики

В основе локальных блокировщиков лежит идиома RAII (захват ресурса при инициализации), применённая к мьютексу. Напомним, эта идиома состоит в том, чтобы привязать захват и освобождение ресурса к началу и концу времени жизни объекта соответственно. Это, в свою очередь, означает, что владение ресурсом привязано к области видимости объекта: по правилам языка C++ с выходом за область видимости объекта автоматически вызывается его деструктор, что приводит к освобождению ресурса.

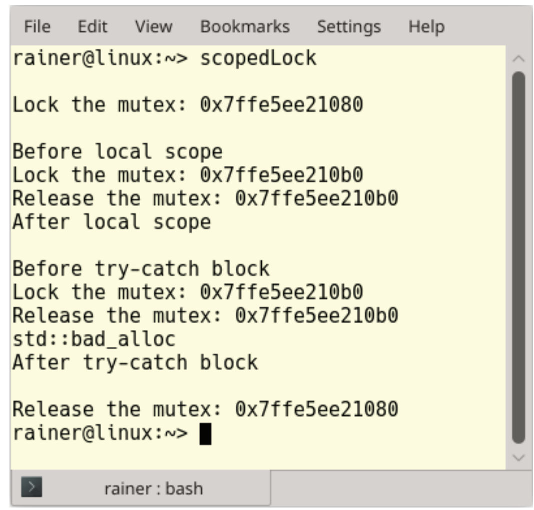
Показанный ниже класс `ScopedLock` представляет собой возможную реализацию локального блокировщика.

Локальный блокировщик

```
1  // scopedLock.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <new>
6  #include <string>
7  #include <utility>
8
9  class ScopedLock{
10   private:
11     std::mutex& mut;
12   public:
13     explicit ScopedLock(std::mutex& m): mut(m){
14         mut.lock();
15         std::cout <<  "Lock the mutex: " << &mut <<   std::endl;
16     }
17     ~ScopedLock(){
18         std::cout << "Release the mutex: " << &mut << std::endl;
19         mut.unlock();
20     }
21 };
22
23 int main(){
24   std::cout << std::endl;
25
26   std::mutex mutex1;
27   ScopedLock scopedLock1{mutex1};
28
29   std::cout << "\nBefore local scope" << std::endl;
30   {
31     std::mutex mutex2;
32     ScopedLock scopedLock2{mutex2};
33   }
34   std::cout << "After local scope" << std::endl;
35
36   std::cout << "\nBefore try-catch block" << std::endl;
37   try{
```

```
38        std::mutex mutex3;
39        ScopedLock scopedLock3{mutex3};
40        throw std::bad_alloc();
41    }
42    catch (std::bad_alloc& e){
43        std::cout << e.what();
44    }
45    std::cout << "\nAfter try-catch block" << std::endl;
46
47    std::cout << std::endl;
48 }
```

Конструктор класса `ScopedLock` получает мьютекс по ссылке. В конструкторе этот мьютекс запирается, а в деструкторе – отпирается. В соответствии с идиомой RAII деструкция объекта и, следовательно, освобождение мьютекса происходят автоматически.



Локальный блокировщик в действии

Область видимости объекта `scopedLock1` простирается до конца главной функции. Следовательно, мьютекс `mutex1` отпирается с завершением программы. Мьютексы `mutex2` и `mutex3` объявлены во вложенных областях видимости. Для мьютекса `mutex3` предусмотрен выход из области видимости также и по исключению. Интересно, что для объекта `mutex3` повторно используется та же область памяти, которую раньше занимал объект `mutex2`: программа печатает те же адреса.

Стандарт C++ 17 поддерживает блокировки четырёх видов. В библиотеке имеются шаблоны классов `std::lock_guard` и `std::scoped_lock` для простых сценариев использования, когда мьютексы захватываются один раз при создании блокировщика и освобождаются только с его деструкцией, и шаблоны классов `std::unique_lock` и `std::shared_lock` для более сложных случаев, пред-

полагающих отпирание и повторное запирание блокировщика на протяжении времени его жизни. Подробнее о них рассказано в разделе 3.3.2.

9.2.1.1. Материал для дальнейшего изучения

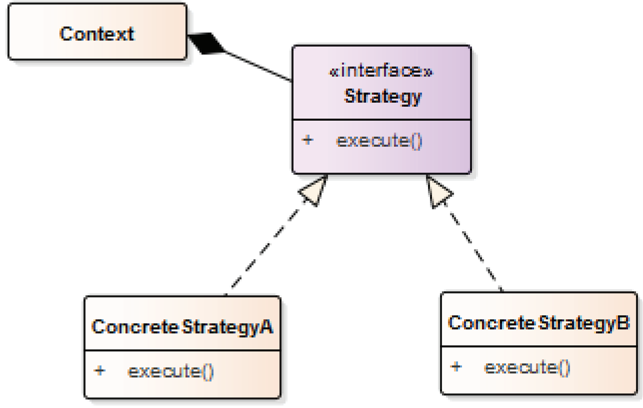
Более подробную информацию можно найти во второй книге из серии POSA[1].

9.2.2. Параметризованные блокировщики

Предположим, что нужно написать такой код (например, библиотеку), который должен использоваться в различных условиях, в том числе и в параллельных программах. Чтобы гарантировать корректную работу кода, критические секции в нём нужно защитить блокировками. Однако если библиотеку использовать в однопоточной среде, блокировки приведут к бессмысленной потере производительности, поскольку затратный механизм синхронизации, внедрённый в код библиотеки, оказывается ненужным. На выручку приходят параметризованные блокировщики.

Параметризованные блокировщики получаются, если шаблон проектирования «Стратегия» применить к обычным блокировщикам. Иными словами, различные стратегии блокирования критических секций нужно поместить в объекты с единым интерфейсом и подключать те или иные из них, в зависимости от потребностей. Для начала вспомним, что представляет собой шаблон «Стратегия».

9.2.2.1. Шаблон «Стратегия»



Шаблон «Стратегия»

Шаблон «Стратегия» был описан ещё в классической книге «Банды четырёх», также известен под названием «Политика». Идея состоит в том, что семей-

[1] https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/.

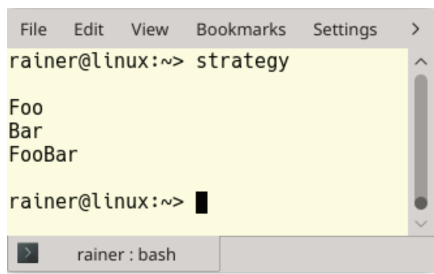
ство алгоритмов, по-разному решающих одну задачу, оформляется в виде семейства объектов, которые можно подключать к системе взаимозаменяемым образом. Следующая небольшая программа иллюстрирует основную идею шаблона.

Шаблон «Стратегия»

```
1  // strategy.cpp
2  
3  #include <iostream>
4  #include <memory>
5  
6  class Strategy {
7  public:
8      virtual void operator()() = 0;
9      virtual ~Strategy() = default;
10 };
11 
12 class Context {
13     std::shared_ptr<Strategy> _strat;
14 public:
15     explicit Context() : _strat(nullptr) {}
16     void setStrategy(std::shared_ptr<Strategy> strat) {
17         _strat = strat;
18     }
19     void strategy()  { if (_strat) (*_strat)(); }
20 };
21 
22 class Strategy1 : public Strategy {
23     void operator()() override {
24         std::cout << "Foo" << std::endl;
25     }
26 };
27 
28 class Strategy2 : public Strategy {
29     void operator()() override {
30         std::cout << "Bar" << std::endl;
31     }
32 };
33 
34 class Strategy3 : public Strategy {
35     void operator()() override {
36         std::cout << "FooBar" << std::endl;
37     }
38 };
39 
40 int main() {
41     std::cout << std::endl;
42 
43     Context con;
44 
45     con.setStrategy( std::shared_ptr<Strategy>(new Strategy1) );
```

```
46     con.strategy();
47
48     con.setStrategy( std::shared_ptr<Strategy>(new Strategy2) );
49     con.strategy();
50
51     con.setStrategy( std::shared_ptr<Strategy>(new Strategy3) );
52     con.strategy();
53
54     std::cout << std::endl;
55 }
```

Абстрактный класс `Strategy` определяет интерфейс, общий для всех конкретных стратегий. Всякая конкретная стратегия – например, классы `Strategy1`, `Strategy2` или `Strategy3` – должна поддерживать операцию функционального вызова. Класс `Context` играет роль клиента, пользующегося некоторой стратегией для своих вспомогательных задач. Конкретная стратегия устанавливается функцией `setStrategy` и применяется функцией `strategy`. Поскольку объект класса `Context` запускает стратегию через указатель на объект абстрактного класса `Strategy`, переопределение виртуальной функции-члена в них может быть скрыто в секции `private`. Объект `con` в главной функции программы поочерёдно пользуется разными стратегиями.



Шаблон «Стратегия» в действии

9.2.2.2. Реализация параметризованных блокировщиков

Есть два основных подхода к реализации блокировщиков, параметризованных стратегией: с полиморфизмом на этапе выполнения (в духе объектно-ориентированного подхода) и с полиморфизмом на этапе компиляции (через метапрограммирование на шаблонах). У каждого из этих способов есть свои преимущества и недостатки.

- Преимущества:
  - полиморфизм на этапе выполнения:
    - возможность конфигурировать стратегию во время выполнения;
    - простота и понятность для разработчиков, владеющих объектно-ориентированным подходом;
  - полиморфизм на этапе компиляции:
    - отсутствие накладных расходов на вызов виртуальной функции;
    - отсутствие иерархии классов.

- Недостатки:
  - полиморфизм на этапе выполнения:
    - потеря быстродействия из-за дополнительного обращения по указателю при вызове виртуальной функции;
    - потенциально – наличие многоуровневой иерархии классов;
  - полиморфизм на этапе компиляции:
    - чрезвычайная длина и неудобочитаемость сообщений компилятора в случае ошибки.

После этого краткого введения рассмотрим реализации параметризованного блокировщика, выполненные в соответствии с обоими указанными подходами. В качестве стратегий выберем отсутствие блокировки, исключительное блокирование и совместное блокирование. Для простоты будем пользоваться существующими в библиотеке мьютексами. Помимо того, параметризованные блокировщики также выполняют функции локальных блокировщиков.

9.2.2.2.1. Полиморфизм на этапе выполнения

В следующей программе реализованы три различные стратегии блокировки с единым интерфейсом.

Параметризованный блокировщик с полиморфизмом на этапе выполнения

```
1  // strategizedLockingRuntime.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <shared_mutex>
6
7  class Lock {
8  public:
9      virtual void lock() const = 0;
10     virtual void unlock() const = 0;
11 };
12
13 class StrategizedLocking {
14     Lock& lock;
15 public:
16     StrategizedLocking(Lock& l): lock(l){
17         lock.lock();
18     }
19     ~StrategizedLocking(){
20         lock.unlock();
21     }
22 };
23
24 struct NullObjectMutex{
25     void lock(){}
26     void unlock(){}
27 };
28
```

```
29 class NoLock : public Lock {
30     void lock() const override {
31         std::cout << "NoLock::lock: " << std::endl;
32         nullObjectMutex.lock();
33     }
34     void unlock() const override {
35         std::cout << "NoLock::unlock: " << std::endl;
36          nullObjectMutex.unlock();
37     }
38     mutable NullObjectMutex nullObjectMutex;
39 };
40
41 class ExclusiveLock : public Lock {
42     void lock() const override {
43         std::cout << "    ExclusiveLock::lock: " << std::endl;
44         mutex.lock();
45     }
46     void unlock() const override {
47         std::cout << "    ExclusiveLock::unlock: " << std::endl;
48         mutex.unlock();
49     }
50     mutable std::mutex mutex;
51 };
52
53 class SharedLock : public Lock {
54     void lock() const override {
55         std::cout << "        SharedLock::lock_shared: " << std::endl;
56         sharedMutex.lock_shared();
57     }
58     void unlock() const override {
59         std::cout << "        SharedLock::unlock_shared: " << std::endl;
60         sharedMutex.unlock_shared();
61     }
62     mutable std::shared_mutex sharedMutex;
63 };
64
65 int main() {
66     std::cout << std::endl;
67
68     NoLock noLock;
69     StrategizedLocking stratLock1{noLock};
70
71     {
72         ExclusiveLock exLock;
73         StrategizedLocking stratLock2{exLock};
74         {
75             SharedLock sharLock;
76             StrategizedLocking startLock3{sharLock};
77         }
78     }
79
80     std::cout << std::endl;
81 }
```

Класс StrategizedLocking управляет абстрактным мьютексом, отвечающим за тот или иной механизм блокировки (строка 14). Этот класс работает по принципу локального блокировщика, захватывая абстрактный мьютекс в конструкторе (строка 17) и освобождая в деструкторе (строка 20). Класс Lock, объявленный в строках 7–11, абстрактен, т. е. представляет собой только определение интерфейса, общего для всех подклассов. К последним относятся классы NoLock (строки 29–39), ExclusiveLock (строки 41–51) и SharedLock (строки 53–63). Класс SharedLock реализован как обёртка над стандартным классом std::shared_mutex и управляет им посредством функций lock_shared (строка 56) и unlock_shared (строка 60). Подобным же образом классы ExclusiveLock и NoLock реализованы поверх стандартного класса std::mutex и мьютекса-заглушки NullObjectMutex. Этот последний класс поддерживает операции запирания и отпирания, но эти функции не делают ничего. Во всех случаях объекты-мьютексы объявлены как изменчивые (mutable). Это значит, что в константных функциях-членах к ним можно применять модифицирующие операции, такие как lock и unlock.

Объект-заглушка

Класс NullObjectMutex представляет собой пример шаблона «Заглушка», который часто оказывается полезен на практике. Все его функции-члены имеют пустую реализацию, поэтому оптимизирующий компилятор может удалить обращения к ним из исполняемого кода программы.

9.2.2.2.2. Полиморфизм на этапе компиляции

Реализация на основе шаблонов оказывается похожей на рассмотренную выше объектно-ориентированную реализацию.

Параметризованный блокировщик с полиморфизмом на этапе компиляции

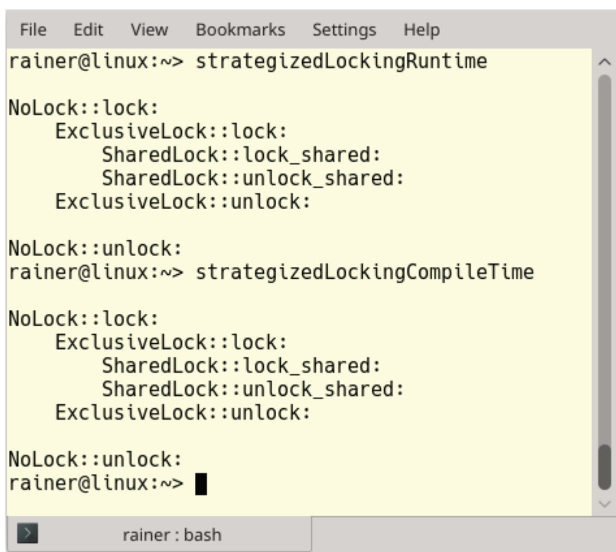
```
1  // StrategizedLockingCompileTime.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <shared_mutex>
6
7
8  template <typename Lock>
9  class StrategizedLocking {
10     Lock& lock;
11 public:
12     StrategizedLocking(Lock& l): lock(l){
13         lock.lock();
14     }
15     ~StrategizedLocking(){
16         lock.unlock();
17     }
18 };
19
20 struct NullObjectMutex {
```

```
21     void lock(){}
22     void unlock(){}
23 };
24 
25 class NoLock{
26 public:
27     void lock() const {
28         std::cout << "NoLock::lock: " << std::endl;
29         nullObjectMutex.lock();
30     }
31     void unlock() const {
32         std::cout << "NoLock::unlock: " << std::endl;
33          nullObjectMutex.lock();
34     }
35     mutable NullObjectMutex nullObjectMutex;
36 };
37 
38 class ExclusiveLock {
39 public:
40     void lock() const {
41         std::cout << "    ExclusiveLock::lock: " << std::endl;
42         mutex.lock();
43     }
44     void unlock() const {
45         std::cout << "    ExclusiveLock::unlock: " << std::endl;
46         mutex.unlock();
47     }
48     mutable std::mutex mutex;
49 };
50 
51 class SharedLock {
52 public:
53     void lock() const {
54         std::cout << "        SharedLock::lock_shared: " << std::endl;
55         sharedMutex.lock_shared();
56     }
57     void unlock() const {
58         std::cout << "        SharedLock::unlock_shared: " << std::endl;
59         sharedMutex.unlock_shared();
60     }
61     mutable std::shared_mutex sharedMutex;
62 };
63 
64 int main() {
65     std::cout << std::endl;
66 
67     NoLock noLock;
68     StrategizedLocking<NoLock> stratLock1{noLock};
69 
70     {
71         ExclusiveLock exLock;
72         StrategizedLocking<ExclusiveLock> stratLock2{exLock};
```

```
73         {
74             SharedLock sharLock;
75             StrategizedLocking<SharedLock> startLock3{sharLock};
76         }
77     }
78 
79     std::cout << std::endl;
80 }
```

В отличие от предыдущего примера, классы `NoLock` (строка 25), `ExclusiveLock` (строка 38) и `SharedLock` (строка 51) не имеют общего абстрактного базового класса. Следовательно, классу `SharedLock` можно в качестве параметра передать класс, не обладающий нужным интерфейсом, что приведёт к ошибке компиляции. Стандарт C++ 20 позволяет в явном виде формулировать требования к типу, передаваемому в шаблон в качестве параметра. Концепты в стандарте C++ 20 выполняют роль, сходную с той, которую в более ранних стандартах играют именованные требования[1], но также поддерживаются компилятором. В данном примере нужно объявить концепт, дав ему, скажем, имя `Lockable`, и использовать его вместо слова `typename` в строке 8. В этом случае все типы, подставляемые в шаблон в качестве аргументов, должны поддерживать данный концепт[2]. Если последнее требование не выполнено, компиляция завершается с осмысленным сообщением.

Программы из этого и предыдущего разделов выдают одинаковый текст, показанный на следующем рисунке.



Параметризованный блокировщик в действии

[1] https://en.cppreference.com/w/cpp/named_req.

[2] https://en.cppreference.com/w/cpp/language/constraints.

9.2.2.3. Материал для дальнейшего изучения

Читателю рекомендуется изучить книгу «Банды четырёх», где собраны сведения о наиболее важных шаблонах проектирования[1]. О шаблоне проектирования «Стратегия» много информации в общедоступных источниках[2]. Об объекте-заглушке как шаблоне проектирования также немало материалов[3]. Более подробную информацию о шаблонах параллельного программирования можно найти во второй книге из серии POSA[4].

9.2.3. Потокобезопасный интерфейс

Потокобезопасные интерфейсы лучше всего подходят в случаях, когда критическую секцию составляет сам по себе объект данных. Наивное решение, состоящее в том, чтобы каждую функцию-член защитить блокировкой, может вызвать снижение производительности в лучшем случае и мёртвую блокировку – в худшем. Следующий небольшой фрагмент псевдокода пояснит эту мысль.

```
struct Critical{
    void memberFunction1(){
        lock(mut);
        memberFunction2();
        ...
    }

    void memberFunction2(){
        lock(mut);
        ...
    }

    mutex mut;
};

Critical crit;
crit.memberFunction1();
```

Для простоты реализации здесь применяется локальный блокировщик в области видимости каждой функции. Вызов функции-члена `crit.memberFunction1` вызывает двукратное запирание мьютекса `mut`. Это приводит к двум проблемам:

- если объект `lock` представляет собой рекурсивный блокировщик, повторная блокировка в функции-члене `memberFunction2` излишня;
- если же блокировщик `lock` нерекурсивный, попытка повторно его заблокировать приводит к неопределённому поведению (на практике чаще всего к мёртвой блокировке).

---

1 https://ru.wikipedia.org/wiki/Design_Patterns.
2 https://ru.wikipedia.org/wiki/Стратегия_%28шаблон_проектирования%29.
3 https://en.wikipedia.org/wiki/Null_object_pattern.
4 https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/.

Потокобезопасный интерфейс позволяет справиться с обеими этими трудностями. Данный шаблон основан на нескольких простых идеях:

- все функции-члены, составляющие интерфейс объекта, т. е. объявленные в секции `public`, должны ставить блокировку на весь объект;
- все функции-члены, составляющие детали внутренней реализации, т. е. объявленные с уровнем доступа `protected` или `private`, не должны блокировать объект;
- интерфейсные функции-члены могут содержать вызовы внутренних функций-членов, но не должны вызывать другие интерфейсные функции.

Следующая программа иллюстрирует этот подход.

Потокобезопасный интерфейс

```
1  // threadSafeInterface.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6
7  class Critical{
8
9  public:
10     void interface1() const {
11         std::lock_guard<std::mutex> lockGuard(mut);
12         implementation1();
13     }
14
15     void interface2(){
16         std::lock_guard<std::mutex> lockGuard(mut);
17         implementation2();
18         implementation3();
19         implementation1();
20     }
21
22 private:
23     void implementation1() const {
24         std::cout << "implementation1: "
25                   << std::this_thread::get_id() << std::endl;
26     }
27     void implementation2(){
28         std::cout << "    implementation2: "
29                   << std::this_thread::get_id() << std::endl;
30     }
31     void implementation3(){
32         std::cout << "        implementation3: "
33                   << std::this_thread::get_id() << std::endl;
34     }
35
36     mutable std::mutex mut;
```

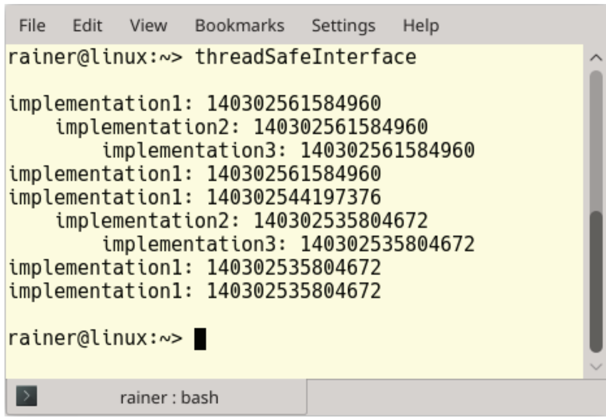
```
37 };
38
39 int main(){
40     std::cout << std::endl;
41
42     std::thread t1([]{
43         const Critical crit;
44         crit.interface1();
45     });
46
47     std::thread t2([]{
48         Critical crit;
49         crit.interface2();
50         crit.interface1();
51     });
52
53     Critical crit;
54     crit.interface1();
55     crit.interface2();
56
57     t1.join();
58     t2.join();
59
60     std::cout << std::endl;
61 }
```

Три потока, включая главный, обращаются к экземплярам класса `Critical`. Благодаря потокобезопасному интерфейсу все вызовы общедоступных функций-членов синхронизированы. Мьютекс `mut` объявлен в строке 36 с ключевым словом `mutable`, что даёт возможность изменять его в константной функции-члене `interface1`.

Потокобезопасные интерфейсы приносят троякую выгоду.

1. Повторный захват мьютекса из одного и того же потока становится невозможным. Напомним, что повторный захват нерекурсивного мьютекса – это неопределённое поведение, обычно выражающееся в мёртвой блокировке.
2. В программе используются минимально необходимые средства блокировки в минимально необходимом количестве – следовательно, выполнение программы требует минимальной синхронизации. Использование рекурсивного мьютекса `std::recursive_mutex` в каждой интерфейсной или закрытой функции-члене привело бы к гораздо большим накладным расходам на синхронизацию.
3. Использование класса `Critical` с точки зрения клиента выглядит простым, так как синхронизация оказывается лишь скрытой от внешнего мира деталью реализации.

Результат работы программы свидетельствует о перемежающемся выполнении трёх потоков.



Потокобезопасный интерфейс

Хотя идея потокобезопасного интерфейса кажется очень простой, есть две тонкости, которые нужно иметь в виду при реализации этого шаблона.

9.2.3.1. Тонкости потокобезопасных интерфейсов

При наличии у класса статических и виртуальных членов от программиста требуется особая внимательность.

9.2.3.1.1. Статические члены

Если класс содержит статическую переменную-член, не являющуюся константой, доступ к ней также нуждается в синхронизации[1]. Рассмотрим пример программы.

Потокобезопасный интерфейс со статической переменной-членом

```
1  class Critical{
2  public:
3      void interface1() const {
4          std::lock_guard<std::mutex> lockGuard(mut);
5          implementation1();
6      }
7
8      void interface2() {
9          std::lock_guard<std::mutex> lockGuard(mut);
10         implementation2();
11         implementation3();
12         implementation1();
```

[1] При этом если доступ к нестатическим данным нужно синхронизировать в масштабе экземпляра, то доступ к статическим данным должен быть синхронизирован в масштабах всего класса. – *Прим. перев.*

```
13     }
14
15 private:
16     void implementation1() const {
17         std::cout << "implementation1: "
18         << std::this_thread::get_id() << '\n';
19         ++called;
20     }
21
22     void implementation2(){
23         std::cout << " implementation2: "
24                   << std::this_thread::get_id() << '\n';
25         ++called;
26     }
27
28     void implementation3(){
29         std::cout << " implementation3: "
30                   << std::this_thread::get_id() << '\n';
31         ++called;
32     }
33
34     inline static int called{0};
35     inline static std::mutex mut;
36 };
```

В этом примере класс `Critical` обладает статической переменной `called` – единым для всех экземпляров счётчиком обращений к функциям реализации. Все объекты этого класса имеют общий доступ к этой переменной и, следовательно, нуждаются в синхронизации. Таким образом, критическая секция охватывает все экземпляры данного класса.

Встраиваемые статические члены-данные

Начиная со стандарта C++ 17 статические переменные можно объявлять встраиваемыми – с ключевым словом `inline`. Такие переменные можно определять и инициализировать внутри определения класса:

```
struct X
{
  inline static int n = 1;
};
```

9.2.3.1.2. Виртуальные функции-члены

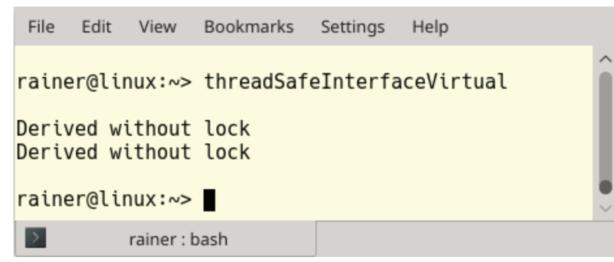
Если интерфейсная виртуальная функция переопределяется в порождённом классе, переопределённая функция должна блокировать объект, даже если в порождённом классе она становится закрытой. Проиллюстрируем сказанное примером.

Потокобезопасный интерфейс с виртуальной функцией-членом

```
1  // threadSafeInterfaceVirtual.cpp
2
```

```
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6
7  class Base {
8  public:
9      virtual void interface() {
10         std::lock_guard<std::mutex> lockGuard(mut);
11         std::cout << "Base with lock" << std::endl;
12     }
13 private:
14     std::mutex mut;
15 };
16
17 class Derived: public Base {
18     void interface() override {
19         std::cout << "Derived without lock" << std::endl;
20     }
21 };
22
23 int main(){
24     std::cout << std::endl;
25
26     Base* base1 = new Derived;
27     base1->interface();
28
29     Derived der;
30     Base& base2 = der;
31     base2.interface();
32
33     std::cout << std::endl;
34 }
```

В вызовах `base1->interface` и `base2.interface` объявленный тип объектов `base1` и `base2` – это класс `Base`. Следовательно, имя `interface` означает общедоступную функцию-член. Поскольку эта интерфейсная функция объявлена виртуальной, её вызов осуществляется исходя из фактического типа объекта на этапе выполнения, т. е. типа `Derived`. В итоге вызывается закрытая функция-член `interface` из класса `Derived`.



Потокобезопасный интерфейс с виртуальной функцией-членом

Есть два способа преодолеть данное затруднение[1]. Во-первых, интерфейсную функцию можно объявить невиртуальной[2]. Этот подход известен под названием NVI[3] (Non-Virtual Interface – невиртуальный интерфейс). Во-вторых, можно запретить переопределение виртуальной функции[4], объявив её с ключевым словом `final`.

9.2.3.2. Материал для дальнейшего изучения

Более подробную информацию можно найти во второй книге из серии POSA[5].

9.2.4. Охраняемая приостановка

Охраняемая приостановка в простейшем случае представляет собой комбинацию блокировки и предусловия, которое должно быть истинным перед выполнением операции. Если предусловие не выполнено, поток, пытающийся выполнить операцию, должен погрузиться в сон до тех пор, пока состояние данных не изменится. Чтобы избежать состояния гонки, обычно приводящего к гонке данных или мёртвой блокировке, поток должен ставить блокировку на время проверки условия.

Эта общая идея допускает ряд вариаций:

- поток может пассивно ожидать оповещения об изменении данных или активно опрашивать состояние на предмет изменений – иными словами, может работать по принципу вталкивания или втягивания;

---

[1] Есть смысл отчётливее пояснить, почему здесь говорится о затруднении. Базовый класс со скрытым от внешнего мира мьютексом и виртуальной функцией в интерфейсе может быть выпущен в составе общедоступной библиотеки. В этом случае автор класса не властен над тем, как программисты, использующие её, переопределяют виртуальные функции из интерфейса. Вполне возможно, что автор порождённого класса забудет поставить локальную блокировку на всё тело переопределённой функции. Тогда код, использующий интерфейс базового класса в предположении, что он потокобезопасен, может, сам того не зная, вызвать небезопасную реализацию из порождённого класса. – *Прим. перев.*

[2] В общем случае идиома невиртуального интерфейса состоит в том, что доступные в интерфейсе класса функции делаются невиртуальными, а полиморфизм достигается за счёт виртуальных непубличных функций. Реализация интерфейсной функции в базовом классе делегирует свою работу виртуальной функции, объявленной в том же базовом классе с уровнем доступа `private` или `protected`, которую порождённые классы могут переопределять. В применении к потокобезопасным интерфейсам эта идиома означает, что невиртуальная и единственная реализация интерфейсной функции в базовом классе захватывает блокировку, после чего вызывает доступную для переопределения виртуальную функцию. – *Прим. перев.*

[3] https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface.

[4] Этот подход может потребовать пояснений. На первый взгляд, объявить функцию виртуальной и сразу запретить её переопределение в порождённых классах ничем не отличается от объявления невиртуальной функции. Важное отличие, однако, проявляется в ситуации, когда интерфейс оформлен в виде чисто абстрактного базового класса, а реализация, включая блокировку объекта на время каждой интерфейсной операции, выполнена в одном или нескольких порождённых классах. В этом случае иерархия классов состоит ровно из двух уровней. – *Прим. перев.*

[5] https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/.

- ожидание может быть ограничено предельным временем или не ограничено;
- при изменении состояния данных оповещение может отсылаться одному или всем ожидающим потокам.

Все перечисленные варианты подробно рассматривались в предыдущих главах по отдельности. В настоящем разделе остаётся лишь свести эти элементы воедино. Читателю может понадобиться повторное обращение к соответствующему материалу из предыдущих глав.

9.2.4.1. Принцип вталкивания и принцип втягивания

Рассмотрение этого аспекта начнём с принципа вталкивания.

9.2.4.1.1. Принцип вталкивания

Чаще всего для подобных задач синхронизации потоков используют переменные условия или пары фьючерс–обещание. Переменная условия или обещание оповещают ожидающий поток. У объекта-обещания нет функции наподобие `notify` или `notify_all`. Вместо этого для отсылки оповещения обычно используется функция `set_value` без аргумента. Следующие два фрагмента кода иллюстрируют принцип работы оповещающего и ожидающего потоков.

Оповещение через переменную условия

```
void waitingForWork() {
    std::cout << "Worker: Waiting for work." << '\n';
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady() {
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    }
    std::cout << "Sender: Data is ready." << '\n';
    condVar.notify_one();
}
```

Оповещение через фьючерс и обещание

```
void waitingForWork(std::future<void>&& fut) {
    std::cout << "Worker: Waiting for work." << '\n';
    fut.wait();
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady(std::promise<void>&& prom) {
```

```
    std::cout << "Sender: Data is ready." << '\n';
    prom.set_value();
}
```

9.2.4.1.2. Принцип втягивания

Вместо того чтобы просто ждать, пока придёт оповещение о новом состоянии данных, поток может сам опрашивать данные. В отличие от вталкивания, в стандарте языка C++ нет прямых средств поддержки для принципа втягивания, однако его легко смоделировать, например, с помощью атомарных переменных. Проиллюстрируем это примером.

Защищённая приостановка по принципу втягивания

```
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork() {
    std::cout << "Waiting " << '\n';
    while (!dataReady.load()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
    mySharedWork[1] = 2;
    std::cout << "Work done " << '\n';
}

void setDataReady() {
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << '\n';
}
```

9.2.4.2. Ограниченное и неограниченное ожидания

У переменных условия и фьючерсов для ожидания предназначены три функции-члена: `wait`, `wait_for` и `wait_until`. Функции `wait_for` в качестве аргумента требуется промежуток времени (см. раздел 14.3), а функции `wait_until` – момент времени (раздел 14.2). Не будем говорить сейчас о функции `wait`, ожидающей неограниченно долго. Примеры раздела 9.2.4.1.1, посвящённого принципу вталкивания, работают именно с неограниченным ожиданием.

Поток-потребитель ожидает не более, чем до заданного момента времени, который определяется как сумма настоящего момента времени `steady_clock::now()` и продолжительности `dur`. Если обещание к этому времени готово, потребитель запрашивает его значение, в противном случае отображает свой идентификатор.

Ограниченное ожидание

```
void producer(promise<int>&& prom) {
    cout << "PRODUCING THE VALUE 2011\n\n";
    this_thread::sleep_for(seconds(5));
```

```
    prom.set_value(2011);
}

void consumer(
    shared_future<int> fut,
    steady_clock::duration dur)
{
    const auto start = steady_clock::now();
    future_status status= fut.wait_until(steady_clock::now() + dur);
    if ( status == future_status::ready ){
        lock_guard<mutex> lockCout(coutMutex);
        cout
            << this_thread::get_id()
            << " ready => Result: "
            << fut.get()
            << '\n';
    }
    else {
        lock_guard<mutex> lockCout(coutMutex);
        cout
            << this_thread::get_id()
            << " stopped waiting."
            << '\n';
    }
    const auto end = steady_clock::now();
    lock_guard<mutex> lockCout(coutMutex);
    cout
        << this_thread::get_id()
        << " waiting time: "
        << getDifference(start,end)
        << " ms"
        << '\n';
}
```

9.2.4.3. Оповещение одного или всех ожидающих потоков

Функция notify_one пробуждает один из ожидающих потоков, тогда как функция notify_all пробуждает все такие потоки. Функция notify_one не даёт никаких гарантий относительно того, какой именно из нескольких ожидающих потоков получит оповещение – остальные потоки продолжат ожидание. Такая ситуация не может возникнуть при использовании типа std::future, так как каждому фьючерсу соответствует одно и только одно обещание. Если же возникает необходимость на основе фьючерсов смоделировать отношение типа «один ко многим», следует воспользоваться типом std::shared_future, который поддерживает операцию копирования.

Следующая программа иллюстрирует связи между обещаниями и фьючерсами по типу «один к одному» и «один ко многим».

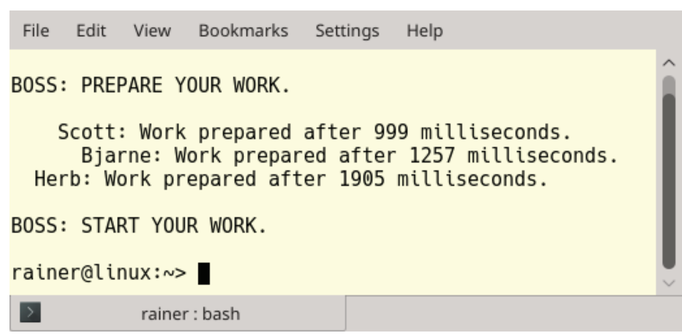
Система «начальник–работник»

```
1  // bossWorker.cpp
2
```

```
3  #include <future>
4  #include <chrono>
5  #include <iostream>
6  #include <random>
7  #include <string>
8  #include <thread>
9  #include <utility>
10
11 int getRandomTime(int start, int end) {
12
13   std::random_device seed;
14   std::mt19937 engine(seed());
15   std::uniform_int_distribution<int> dist(start,end);
16
17   return dist(engine);
18 };
19
20 class Worker {
21 public:
22   explicit Worker(const std::string& n):name(n) {};
23
24   void operator() (std::promise<void>&& preparedWork,
25                    std::shared_future<void> boss2Worker) {
26
27     // prepare the work and notfiy the boss
28     int prepareTime= getRandomTime(500, 2000);
29     std::this_thread::sleep_for(std::chrono::milliseconds(prepareTime));
30     preparedWork.set_value();
31     std::cout << name << ": " << "Work prepared after "
32               << prepareTime << " milliseconds." << std::endl;
33
34     // still waiting for the permission to start working
35     boss2Worker.wait();
36   }
37 private:
38   std::string name;
39 };
40
41 int main() {
42   std::cout << std::endl;
43
44   // define the std::promise => Instruction from the boss
45   std::promise<void> startWorkPromise;
46
47   // get the std::shared_future's from the std::promise
48   std::shared_future<void> startWorkFuture =
49       startWorkPromise.get_future();
50
51   std::promise<void> herbPrepared;
52   std::future<void> waitForHerb = herbPrepared.get_future();
53   Worker herb("  Herb");
54   std::thread herbWork(herb, std::move(herbPrepared), startWorkFuture);
```

```
55
56  std::promise<void> scottPrepared;
57  std::future<void> waitForScott = scottPrepared.get_future();
58  Worker scott("    Scott");
59  std::thread scottWork(
60      scott, std::move(scottPrepared), startWorkFuture);
61
62  std::promise<void> bjarnePrepared;
63  std::future<void> waitForBjarne = bjarnePrepared.get_future();
64  Worker bjarne("      Bjarne");
65  std::thread bjarneWork(
66      bjarne, std::move(bjarnePrepared), startWorkFuture);
67
68  std::cout << "BOSS: PREPARE YOUR WORK.\n " << std::endl;
69
70  // waiting for the worker
71  waitForHerb.wait(), waitForScott.wait(), waitForBjarne.wait();
72
73  // notify the workers that they should begin to work
74  std::cout << "\nBOSS: START YOUR WORK. \n" << std::endl;
75  startWorkPromise.set_value();
76
77  herbWork.join();
78  scottWork.join();
79  bjarneWork.join();
80 }
```

Основная идея этой программы состоит в том, что начальник (главный поток) управляет тремя работниками – потоками `herb`, `scott` и `bjarne`. В строке 71 главный поток ждёт, пока каждый из работников закончит подготовку к работе. Для этого каждый работник отмечает выполненным соответствующее обещание. Оповещение, приходящее от работника к начальнику, представляет собой отношение типа «один к одному», так как выполняется через объект типа `std::future`. В противоположность этому оповещение работников от начальника о том, что они могут приступать к основной работе (строка 75), имеет семантику «один ко многим», для него необходим тип `std::shared_future`. Результат запуска программы показан на рисунке.



Система «начальник–работник»

9.2.4.4. Материал для дальнейшего изучения

Затронутые в настоящем разделе вопросы подробно освещены в руководстве по параллельному программированию на языке Java[1]. Различие языков программирования не должно стать препятствием для понимания сути дела.

9.3. Краткие итоги

Необходимое условие для возникновения гонки данных – наличие общего доступа к изменяемому состоянию. Поэтому шаблоны синхронизации сводятся в основном к решению двух основных задач: управление общим доступом и управление изменяемым состоянием.

[1] http://gee.cs.oswego.edu/dl/cpj/.

10. Шаблоны параллельной архитектуры

Три шаблона, представленных в этой главе, можно назвать классическими. Все три очень хорошо объяснены во второй книге серии POSA, значение которой трудно переоценить[1]. Цель настоящей главы – познакомить читателя с шаблонами «Активный объект», «Объект-монитор» и «Полусинхронная архитектура». Как и в предыдущей главе, посвящённой шаблонам синхронизации, параллельные архитектурные шаблоны будут рассматриваться через призму языка C++. Прежде чем погрузиться в подробное изучение трёх шаблонов, составим о них самое общее представление.

Шаблон «Активный объект» разрывает жёсткую связь между вызовом и выполнением функции-члена объектом, который находится в собственном потоке управления. Параллельное выполнение поддерживается за счёт асинхронного механизма вызова функций-членов и планировщика, который обрабатывает поставленные в очередь запросы. Для предварительного знакомства с шаблоном можно воспользоваться статьёй из Википедии[2].

Шаблон «Объект-монитор» синхронизирует параллельное выполнение функций-членов так, чтобы в любой момент времени могла выполняться лишь одна из них. Этот шаблон также позволяет выстраивать вызовы функций-членов объекта в последовательности для достижения общей цели. Более подробные сведения можно найти во второй книге серии POSA.

Оба этих шаблона имеют дело с синхронизацией и планированием выполнения функций-членов. Различие между ними состоит в том, что активный объект выполняет функции-члены в другом потоке, а объект-монитор – в том же потоке, что и код, вызвавший функцию. В отличие от активного объекта и объекта-монитора, нацеленных главным образом на уровень подсистем и потому называемых шаблонами проектирования, следующий шаблон относится к строению системы как целого и потому считается архитектурным шаблоном.

---

[1] https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/.

[2] https://en.wikipedia.org/wiki/Active_object.

Шаблон «Полусинхронная архитектура» предполагает разделение параллельной системы на сервисы синхронной и асинхронной обработки, что упрощает программирование, не приводя к чрезмерной потере производительности. Для такого разделения требуются два промежуточных слоя взаимодействия: один для синхронной и один для асинхронной обработки. Более подробное описание содержится во второй книге серии POSA.

10.1. Активный объект

Шаблон проектирования «Активный объект» предполагает отделение выполнения функции-члена от её вызова для объектов, обладающих собственными потоками управления. Цель шаблона состоит в том, чтобы сделать работу системы параллельной за счёт асинхронных вызовов функций и планировщика, который управляет выполнением запросов.

Опишем данный шаблон более подробно. Когда клиент вызывает функцию-член объекта, на самом деле вызывается функция объекта-заместителя, который служит лишь интерфейсом активного объекта. Другой объект играет роль сервера и содержит реализацию всей функциональности активного объекта – он функционирует в отдельном потоке. Задача объекта-заместителя – превратить своих функций-членов в обращения к функциям объекта-сервера. Запросы к объекту-серверу ставятся в очередь, обработкой которой занимается планировщик. Планировщик постоянно выполняет цикл обработки событий, выбирает из очереди запросы по порядку и вызывает соответствующие функции объекта-сервера. Клиентский код получает результат вызова через посредство интерфейсного объекта-заместителя с помощью фьючерса.

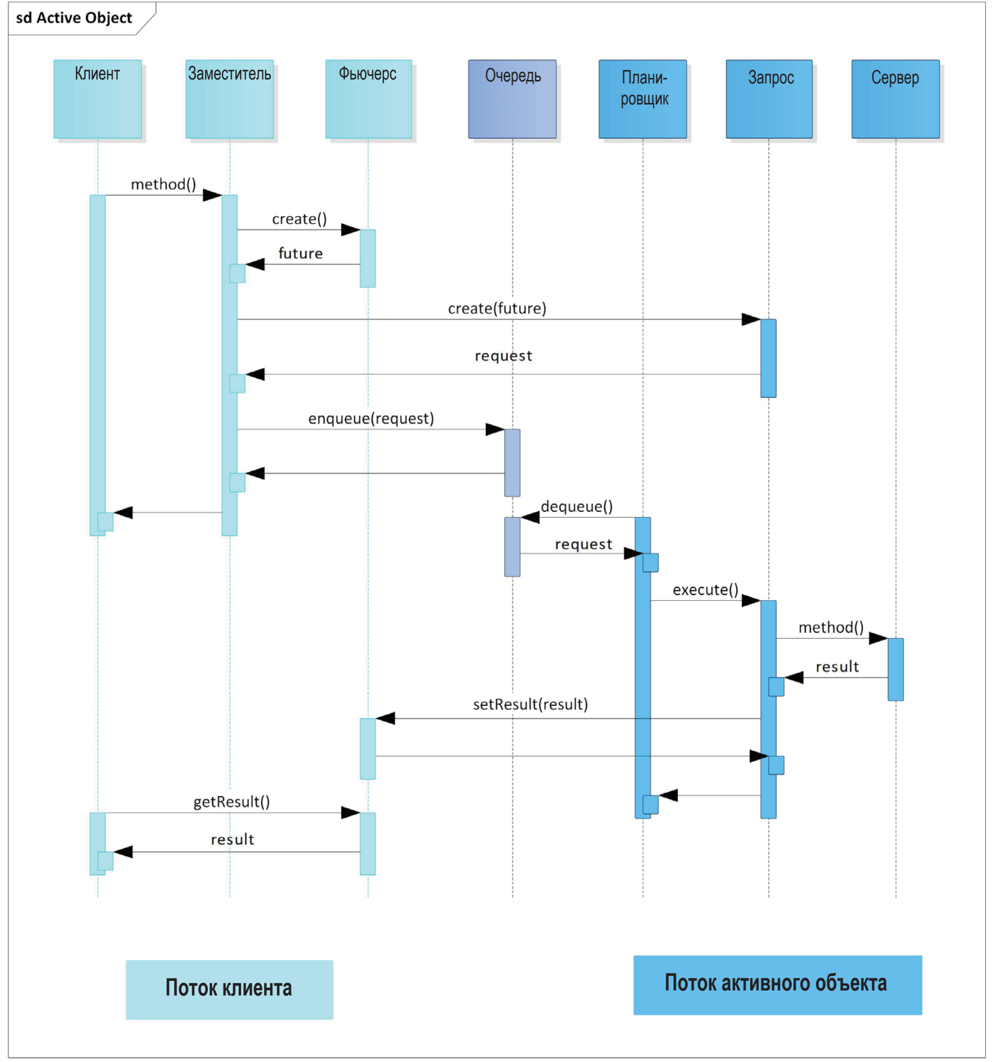
10.1.1. Компоненты шаблона

Шаблон «Активный объект» включает в себя шесть компонентов.

1. Объект-заместитель обеспечивает доступный для клиента интерфейс. Реализация интерфейсных функций объекта-заместителя состоит в создании запроса на вызов функции-члена и отправке его в очередь. Функции-члены этого объекта отрабатывают в клиентском потоке, который их вызывает.
2. Запрос на вызов функции – это объект данных, содержащий информацию о том, какую функцию и с какими аргументами нужно вызвать.
3. Очередь запросов – это линейная последовательность запросов на вызов, ожидающих своего выполнения. Она служит связующим звеном между клиентскими потоками, которые осуществляют вызовы, и потоком, в котором вызовы выполняются. Объект-заместитель помещает новые запросы в конец очереди, а планировщик изымает запросы из начала.
4. Планировщик работает в отдельном потоке. Он выбирает следующий по порядку запрос из очереди и вызывает соответствующую функцию-член объекта-сервера.

5. Объект-сервер содержит реализацию активного объекта. Его интерфейс совпадает с интерфейсом объекта-заместителя. Функции этого объекта вызываются в потоке планировщика.
6. Фьючерс, создаваемый объектом-заместителем. Он позволяет клиенту, обратившемуся к функции объекта-заместителя, получить результат вызова планировщиком соответствующей функции объекта-сервера. Клиент может как пассивно дождаться результата, так и активно опрашивать фьючерс на предмет готовности.

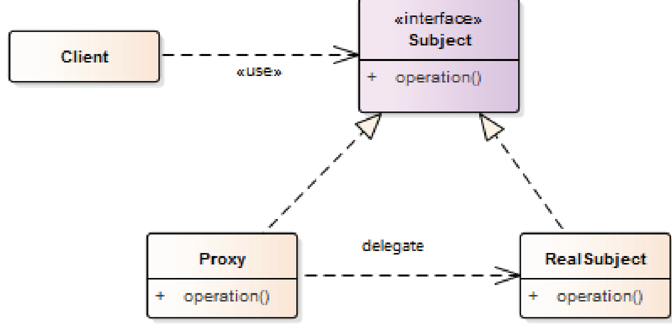
На следующем рисунке изображена последовательность взаимодействий между объектами, вместе составляющими активный объект.



Активный объект

Объект-заместитель

Шаблон проектирования «Заместитель» входит в число классических шаблонов, описанных в книге «Банды четырёх». В общем случае заместитель – это объект, который функционирует как представитель какого-либо иного объекта с тем же интерфейсом, делегируя ему свои функции. Типичными примерами могут служить объект-заместитель в технологии CORBA[1], заместитель, обеспечивающий безопасность объекта-сервера, виртуальный заместитель, создающий объект-сервер на лету, или умный указатель[2] наподобие типа `std::shared_ptr`. Всякий заместитель добавляет некоторую функциональность к объекту, который он замещает. Так, заместитель удалённого объекта инкапсулирует детали сетевого взаимодействия и создаёт у клиента иллюзию, будто объект-сервер функционирует на той же машине, что и клиент. Заместитель для обеспечения безопасности шифрует запросы к объекту-серверу и расшифровывает ответы от него. Виртуальный заместитель инкапсулирует создание тяжеловесного объекта-сервера в ленивом стиле. Наконец, умный указатель управляет временем жизни объекта-сервера.



Шаблон «Заместитель»

Основные характеристики шаблона таковы:

- объект-заместитель Proxy обладает тем же интерфейсом, что и объект-исполнитель RealSubject, содержит в себе ссылку на него и, возможно, управляет временем его жизни;
- интерфейс Subject – общий для двух объектов: заместителя Proxy и исполнителя RealSubject;
- объект RealSubject отвечает за реализацию всей полезной функциональности.

Больше подробностей о шаблоне «Заместитель» можно почерпнуть в статье Википедии[3].

10.1.2. Преимущества и недостатки активных объектов

Перед тем как показывать минимальную программную реализацию шаблона «Активный объект», перечислим его преимущества и недостатки. Начнём с преимуществ.

---

1 https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture.

2 https://en.cppreference.com/w/cpp/memory/shared_ptr.

3 https://en.wikipedia.org/wiki/Proxy_pattern.

- Синхронизация только в потоке планировщика активного объекта, но не в клиентских потоках.
- Чёткое разграничение клиента (пользовательского кода) и сервера (реализации функций объекта). При этом вся работа по синхронизации выполняется на стороне сервера.
- Повышение пропускной способности системы вследствие асинхронного способа обслуживания запросов. Продолжительные вызовы функций объекта-сервера не блокируют систему в целом.
- В планировщике можно реализовать различные стратегии выполнения ожидающих запросов. В зависимости от специфики задачи можно реализовать выполнение запросов не в том порядке, в котором они поступили[1].

Следует упомянуть также и о присущих данному шаблону недостатках.

- Если функции-члены активного объекта слишком просты и выполняются быстро, накладные расходы на прохождение вызова через объект-заместитель, очередь и планировщик могут оказаться чрезмерными.
- Наличие как собственного планировщика в составе активного объекта, так и планировщика потоков в операционной системе может сильно усложнить отладку активного объекта. Особенно сложной становится отладка, если задания выполняются сервером не в том порядке, в котором они заказаны клиентом.

10.1.3. Реализация

Ниже представлена упрощённая реализация шаблона «Активный объект». Так, в этом коде отсутствует запрос на вызов как отдельный объект данных, а планировщик всегда просто выбирает из очереди следующее задание.

Типы данных, возникающие при работе с асинхронными заданиями, часто имеют слишком громоздкие имена, например:

```
std::future<std::vector<std::future<std::pair<bool, int>>>>
```

Во избежание этого в программе активно используется ключевое слово `using`.

Активный объект

```
1   // activeObject.cpp
2
```

[1] Например, различным клиентам могут быть назначены те или иные приоритеты, чтобы запрос от клиента с более высоким приоритетом обслуживался раньше, чем более ранний запрос от низкоприоритетного клиента. Помимо этого, различные приоритеты могут быть назначены вызываемым функциям, чтобы запрос на критически важную функцию обслуживался раньше, чем более ранний запрос на чисто сервисную функцию. Также можно предусмотреть исключение из очереди просроченных запросов, клиент в этом случае может получить исключение. – *Прим. перев.*

```
3   #include <algorithm>
4   #include <deque>
5   #include <functional>
6   #include <future>
7   #include <iostream>
8   #include <memory>
9   #include <mutex>
10  #include <numeric>
11  #include <random>
12  #include <thread>
13  #include <utility>
14  #include <vector>
15
16  using std::async;
17  using std::boolalpha;
18  using std::cout;
19  using std::deque;
20  using std::distance;
21  using std::endl;
22  using std::for_each;
23  using std::find_if;
24  using std::future;
25  using std::lock_guard;
26  using std::make_move_iterator;
27  using std::make_pair;
28  using std::move;
29  using std::mt19937;
30  using std::mutex;
31  using std::packaged_task;
32  using std::pair;
33  using std::random_device;
34  using std::sort;
35  using std::thread;
36  using std::uniform_int_distribution;
37  using std::vector;
38
39  class IsPrime {
40  public:
41      pair<bool, int> operator()(int i) {
42          for (int j=2; j*j <= i; ++j) {
43              if (i % j == 0) return std::make_pair(false, i);
44          }
45          return std::make_pair(true, i);
46      }
47  };
48
49  class ActiveObject {
50  public:
51      future<pair<bool, int>> enqueueTask(int i) {
52          IsPrime isPrime;
53          packaged_task<pair<bool, int>(int)> newJob(isPrime);
```

```
54          auto isPrimeFuture = newJob.get_future();
55          auto pair = make_pair(move(newJob), i);
56          {
57              lock_guard<mutex> lockGuard(activationListMutex);
58              activationList.push_back(move(pair));
59          }
60          return isPrimeFuture;
61      }
62
63      void run() {
64          thread servant([this] {
65              while ( !isEmpty() ) {
66                  auto myTask = dequeueTask();
67                  myTask.first(myTask.second);
68              }
69          });
70          servant.join();
71      }
72
73  private:
74      pair<packaged_task<pair<bool, int>(int)>, int > dequeueTask() {
75          lock_guard<mutex> lockGuard(activationListMutex);
76          auto myTask= std::move(activationList.front());
77          activationList.pop_front();
78          return myTask;
79      }
80
81      bool isEmpty() {
82          lock_guard<mutex> lockGuard(activationListMutex);
83          auto empty = activationList.empty();
84          return empty;
85      }
86
87      using TaskType = packaged_task<pair<bool, int>(int)>;
88      deque<pair<TaskType, int>> activationList;
89      mutex activationListMutex;
90  };
91
92  vector<int> getRandNumbers(int number) {
93      random_device seed;
94      mt19937 engine(seed());
95      uniform_int_distribution<> dist(1000000, 1000000000);
96      vector<int> numbers;
97      for (long long i = 0 ; i < number; ++i)
98          numbers.push_back(dist(engine));
99      return numbers;
100 }
101
102 future<vector<future<pair<bool, int>>>> getFutures(
103     ActiveObject& activeObject,
104     int numberPrimes)
```

```
105 {
106     return async([&activeObject, numberPrimes]{
107         vector<future<pair<bool, int>>> futures;
108         auto randNumbers = getRandNumbers(numberPrimes);
109         for (auto numb: randNumbers){
110             futures.push_back(activeObject.enqueueTask(numb));
111         }
112         return futures;
113     });
114 }
115
116 int main() {
117     cout << boolalpha << endl;
118
119      ActiveObject activeObject;
120
121     // a few clients enqueue work concurrently
122     auto client1 = getFutures(activeObject, 1998);
123     auto client2 = getFutures(activeObject, 2003);
124     auto client3 = getFutures(activeObject, 2011);
125     auto client4 = getFutures(activeObject, 2014);
126     auto client5 = getFutures(activeObject, 2017);
127
128     // give me the futures
129     auto futures = client1.get();
130     auto futures2 = client2.get();
131     auto futures3 = client3.get();
132     auto futures4 = client4.get();
133     auto futures5 = client5.get();
134
135     // put all futures together
136     futures.insert(
137         futures.end(),
138         make_move_iterator(futures2.begin()),
139         make_move_iterator(futures2.end()));
140
141     futures.insert(
142         futures.end(),
143         make_move_iterator(futures3.begin()),
144         make_move_iterator(futures3.end()));
145
146     futures.insert(
147         futures.end(),
148         make_move_iterator(futures4.begin()),
150         make_move_iterator(futures4.end()));
151
152     futures.insert(
153         futures.end(),
154         make_move_iterator(futures5.begin()),
155         make_move_iterator(futures5.end()));
156
```

```
157     // run the promises
158     activeObject.run();
159
160     // get the results from the futures
161     vector<pair<bool, int>> futResults;
162     futResults.reserve(futures.size());
163     for (auto& fut: futures) futResults.push_back(fut.get());
164
165     sort(futResults.begin(), futResults.end());
166
167     // separate the primes from the non-primes
168     auto prIt = find_if(
169         futResults.begin(),
170         futResults.end(),
171          [](pair<bool, int> pa){ return pa.first == true; });
172
173     cout
174         << "Total primes: "
175         << distance(prIt, futResults.end())
176         << endl;
177     cout << "Primes:" << endl;
178     for_each(
179         prIt,
180         futResults.end(),
181         [](auto p){ cout << p.second << " ";} );
182
183     cout << "\n\n";
184
185     cout
186         << "Total non-primes: "
187         << distance(futResults.begin(), prIt)
188         << endl;
189     cout << "Non-primes:" << endl;
190     for_each(
191         futResults.begin(),
192         prIt,
193         [](auto p){ cout << p.second << " ";} );
194
195     cout << endl;
196 }
```

Принцип работы этой программы состоит в том, что клиенты, работая параллельно, добавляют в очередь свои задания, передавая в качестве аргумента целые числа. Задача сервера – определить, какие из этих чисел являются простыми. Очередь заданий здесь сделана частью активного объекта. Активный объект в отдельном потоке выбирает задания на выполнение, а клиенты получают результаты выполнения заданий.

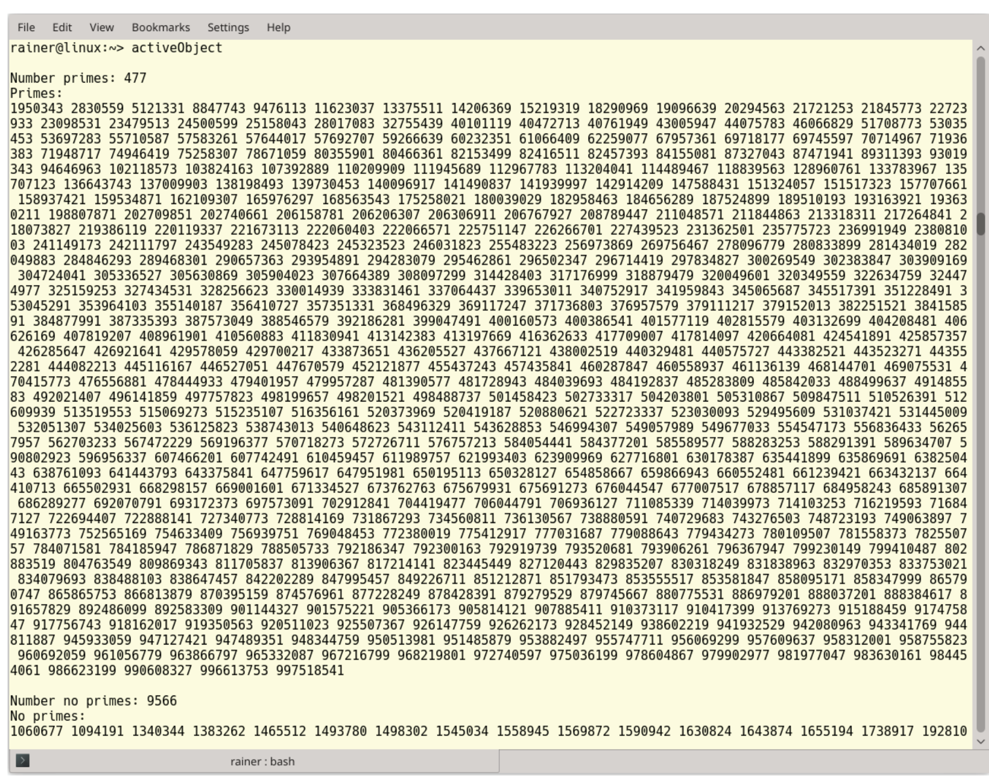
Рассмотрим подробнее внутреннее устройство программы. Пять клиентов (см. строки 122–126) загружают активный объект работой через функцию `getFutures`. Эта функция принимает ссылку на активный объект, которому бу-

дет отправлена работа на выполнение, и целое число numberPrimes – количество случайных чисел в диапазоне от 1 000 000 до 1 000 000 000, которые нужно сгенерировать. Задания, отправленные активному объекту, складываются в контейнер – вектор фьючерсов. Фьючерс содержит в себе пару, первый компонент которой – логическое значение, а второй – то же целое число, которое послужило аргументом при запуске задания, причём логическое значение показывает, является ли число простым. Рассмотрим внимательнее строку 110. Она добавляет новое задание в очередь. Конечно, все операции над очередью синхронизированы. Очередь представляет собой контейнер объектов-обещаний. Каждое обещание, когда приходит время его выполнять, вызывает функциональный объект IsPrime (строки 39–47) с соответствующим аргументом. Возвращаемое значение – пара, состоящая из целого числа (того же, с которым был вызван функциональный объект) и логического значения, показывающего, простое ли оно. Таким образом, задания для последующего выполнения сформированы, пора приступить к вычислениям. Клиенты возвращают свои контейнеры фьючерсов в строках 129–133. Слияние их в один контейнер (строки 136–154) упрощает дальнейшую обработку результатов. Вызов функции run активного объекта activeObject в строке 157 начинает выполнение заданий. Функция run запускает поток и выполняет задания из очереди, пока не окажутся выполненными все (строка 65). Функция-член isEmpty (строка 81) определяет, пуста ли очередь, а функция-член dequeTask выбирает из очереди следующее задание. Цикл в строке 162 извлекает из каждого фьючерса результат и помещает его в новый контейнер. Затем в строке 164 этот контейнер сортируется.

Оставшаяся часть программы обрабатывает результаты вычислений. Итератор ptIt указывает на первое простое число в контейнере. Количество простых чисел определяется как расстояние между итератором It и концом контейнера, а количество чисел, не являющихся простыми, – как расстояние между началом контейнера и итератором It. На рисунке представлено лишь начало списка непростых чисел[1].

---

[1] Представленная здесь реализация активного объекта имеет ряд недостатков и не в полной мере иллюстрирует этот шаблон проектирования. Основной смысл активного объекта состоит в том, что он работает в своём потоке параллельно с потоками-клиентами. Клиенты могут осуществлять вызовы, добавляя тем самым новые элементы в очередь, параллельно с работой активного объекта по обслуживанию ранее сделанных заявок. В показанном примере, напротив, активный объект ничего не делает всё то время, пока пять клиентов наполняют его очередь заданиями – тем самым теряя право называться *активным* объектом. Далее функция-член run, запускающая цикл обработки заявок, отрабатывает однократно и завершается, тогда как настоящий активный объект должен работать непрерывно, переходя в пассивное ожидание в случае пустой очереди. Наконец, вместо двух отдельных функций для проверки пустоты очереди и взятия из неё следующего элемента лучше было бы реализовать одну операцию, чтобы избежать двукратного захвата мьютекса, что, как известно, представляет собой дорогостоящую операцию. – *Прим. перев.*



Активный объект в действии

10.1.3.1. Материал для дальнейшего изучения

Более подробное изложение затронутых здесь вопросов можно найти во второй книге серии POSA; в статье Герба Саттера «Чем активные объекты лучше обычных потоков»[1]; кроме того, рекомендуется ознакомиться с тщательно сделанной реализацией активного объекта на языке C++[2].

10.2. Объект-монитор

Идея шаблона «Монитор» состоит в синхронизации параллельного выполнения функций-членов объекта так, чтобы не более одной из них могло выполняться в каждый момент времени. Кроме того, данный шаблон позволяет строить последовательность выполнения функций-членов объекта для решения общей задачи. Этот шаблон известен также под названием «Потокобезопасный пассивный объект».

---

1 Prefer Using Active Object instead of Naked Thread (Herb Sutter): http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095.

2 https://github.com/lightful/syscpp/.

10.2.1. Требования

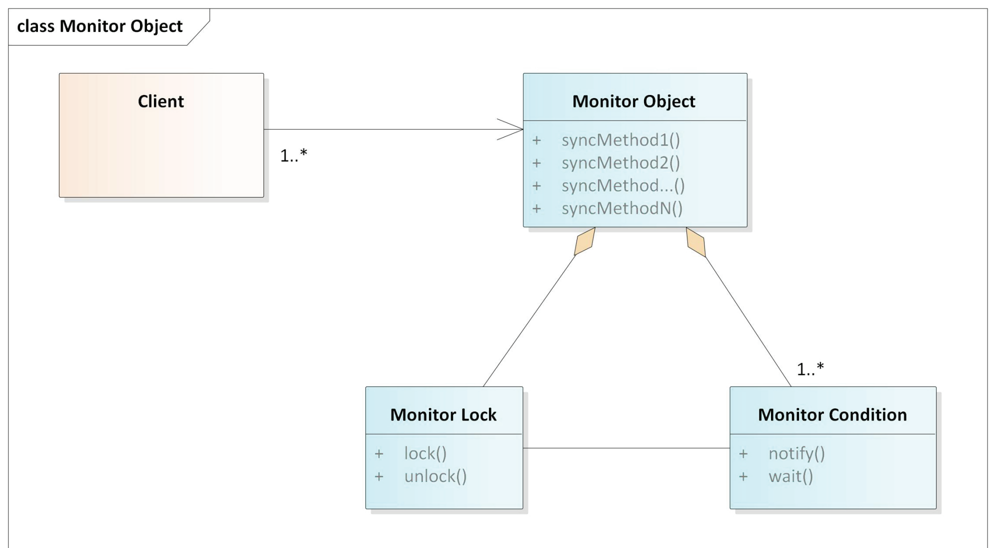
Если несколько потоков одновременно имеют доступ к одному объекту, должны соблюдаться следующие ограничения.

1. Во избежание гонки данных находящийся в общем доступе объект должен быть защищён от несинхронизированных операций записи и чтения.
2. Необходимые для этого механизмы синхронизации должны составлять деталь реализации объекта, а не часть его интерфейса.
3. Когда какой-либо поток заканчивает работу с общим объектом, ожидающие потоки должны получать оповещение о том, что могут приступить к работе с объектом. Этот механизм позволяет избежать мёртвых блокировок и улучшает общую производительность системы.
4. После того как функция-член объекта отработает, истинность инвариантов общего объекта должна сохраняться.

Шаблон «Монитор» представляет собой решение всех четырёх задач. Поток-клиент может получить доступ к синхронизированным функциям-членам объекта-монитора – причём благодаря наличию блокировки только одна функция может выполняться в любой момент времени. Каждый объект-монитор содержит переменную условия, через которую происходит оповещение ожидающих клиентов.

10.2.2. Компоненты

Объект-монитор состоит из четырёх компонентов, как показано на следующем рисунке.



Объект-монитор

1. Собственно объект-монитор, который поддерживает одну или несколько функций-членов. Каждый клиент может обращаться к объекту только посредством этих функций. Вызванная функция выполняется в потоке клиента.
2. Синхронизированные функции-члены объекта-монитора. Механизм синхронизации гарантирует, что только одна из них может выполняться в любой заданный момент времени. Для этого хорошо подходит шаблон «Потокобезопасный интерфейс», о котором шла речь в разделе 9.2.3, – он предписывает делать различие между интерфейсными функциями-членами и внутренними, составляющими детали реализации.
3. Блокировщик монитора. Каждый объект-монитор содержит один примитив блокировки, используемый для синхронизации интерфейсных функций.
4. Условия монитора, которые позволяют различным потокам согласовать между собой вызовы функций-членов монитора. Всякий раз, когда клиент заканчивает выполнение синхронизированной функции-члена, по условию пробуждается следующий клиент, ожидающий своего права вызвать функцию монитора.

Если блокировщик гарантирует исключительный доступ к монитору единственного клиента, то условие монитора сводит к минимуму время ожидания клиентов. Если блокировщик защищает монитор от гонки данных, то условия защищают от мёртвых блокировок.

10.2.3. Принцип действия монитора

Взаимодействие между различными составными частями шаблона происходит в несколько этапов.

- Когда клиент вызывает синхронизированную функцию-член монитора, в первую очередь запирается блокировщик объекта. Если клиенту удалось захватить блокировку, выполняется вызванная функция, после чего снимается блокировка. Если же захват блокировки не удался, клиент блокируется.
- Заблокированный клиент ждёт оповещения от условия монитора. Оповещение происходит в момент освобождения монитора предыдущим клиентом. Оно может отсылаться одному взятому наугад ожидающему клиенту или всем таким клиентам. Ожидание условия обычно экономно расходует машинные ресурсы, в отличие от цикла активного ожидания.
- Когда клиент получает оповещение и пробуждается, он запирает блокировщик монитора и выполняет функцию. По окончании этого блокировка снимается и высылается оповещение, пробуждающее следующего клиента.

10.2.3.1. Преимущества и недостатки мониторов

Мониторы обладают следующими преимуществами.

- Клиент не перегружен деталями синхронизации – они полностью скрыты в реализации монитора.

- Синхронизированные функции-члены по мере вызова автоматически выстраиваются в очередь. Механизм ожидания и оповещения, воплощённый в виде условия, работает как простой планировщик.

Данный шаблон не свободен также от ряда недостатков.

- Непросто бывает изменить механизм синхронизации, заложенный в функциях-членах монитора, поскольку полезная нагрузка функции и механизм синхронизации слишком жёстко связаны друг с другом.
- Если синхронизированная функция-член монитора вызывает, прямо или косвенно, функцию того же самого объекта-монитора, может произойти мёртвая блокировка.

10.2.3.2. Реализация монитора

В следующем примере показана реализация потокобезопасной очереди на основе шаблона «Монитор».

Объект-монитор

```
1  #include <condition_variable>
2  #include <functional>
3  #include <queue>
4  #include <iostream>
5  #include <mutex>
6  #include <random>
7  #include <thread>
8
9  template <typename T>
10 class Monitor {
11 public:
12     void lock() const {
13         monitMutex.lock();
14     }
15     void unlock() const {
16         monitMutex.unlock();
17     }
18
19     void notify_one() const noexcept {
20         monitCond.notify_one();
21     }
22     void wait() const {
23         std::unique_lock<std::recursive_mutex> monitLock(monitMutex);
24         monitCond.wait(monitLock);
25     }
26
27 private:
28     mutable std::recursive_mutex monitMutex;
29     mutable std::condition_variable_any monitCond;
30 };
31
32 template <typename T>
```

```
33 class ThreadSafeQueue: public Monitor<ThreadSafeQueue<T> > {
34 public:
35     void add(T val) {
36         derived.lock();
37         myQueue.push(val);
38         derived.unlock();
39         derived.notify_one();
40     }
41 
42     T get() {
43         derived.lock();
44         while (myQueue.empty()) derived.wait();
45         auto val = myQueue.front();
46         myQueue.pop();
47         derived.unlock();
48         return val;
49     }
50 
51 private:
52     std::queue<T> myQueue;
53     ThreadSafeQueue<T>& derived =
54         static_cast<ThreadSafeQueue<T>&>(*this);
55 };
56 
57 class Dice {
58 public:
59     int operator()() { return rand(); }
60 private:
61     std::function<int()> rand = std::bind(
62         std::uniform_int_distribution<>(1, 6),
63         std::default_random_engine());
64 };
65 
66 int main(){
67     std::cout << std::endl;
68 
69     constexpr auto NUM = 100;
70 
71     ThreadSafeQueue<int> safeQueue;
72     auto addLambda = [&safeQueue] (int val){ safeQueue.add(val); };
73     auto getLambda = [&safeQueue] {
74         std::cout
75             << safeQueue.get()
76             << " "
77             << std::this_thread::get_id()
78             << "; ";
79     };
80 
81     std::vector<std::thread> addThreads(NUM);
82     Dice dice;
83     for (auto& thr: addThreads) thr = std::thread(addLambda, dice() );
84 
```

```
85      std::vector<std::thread> getThreads(NUM);
86      for (auto& thr: getThreads) thr = std::thread(getLambda);
87
88      for (auto& thr: addThreads) thr.join();
89      for (auto& thr: getThreads) thr.join();
90
91      std::cout << "\n\n";
92  }
```

Основная идея этого примера состоит в том, что монитор реализован в виде класса и, следовательно, допускает многократное использование. В классе `Monitor` используются вспомогательные переменные-члены типа `std::recursive_mutex` для блокировки и типа `std::condition_variable_any` в качестве условия монитора. В отличие от знакомого по предыдущим примерам типа `std::condition_variable`, этот последний может использоваться совместно с рекурсивным мьютексом. Обе эти переменные объявлены с ключевым словом `mutable` и, следовательно, их можно модифицировать в константных функциях. Класс `Monitor` обладает минимально возможным интерфейсом, который должен поддерживать всякий объект-монитор.

Класс `ThreadSafeQueue`, объявленный в строках 32–55, представляет собой обёртку над стандартным типом очереди `std::queue` с потокобезопасным интерфейсом. Для этого класс `ThreadSafeQueue` порождён от класса `Monitor` и пользуется его функциями-членами в реализации своих синхронизированных функций `add` и `get`. Эти две функции используют предоставляемый монитором примитив блокировки, для того чтобы защитить состояние монитора, в частности контейнер `myQueue`. Функция `add` оповещает ожидающие потоки о добавлении в очередь нового элемента. Отсылка оповещения происходит потокобезопасным образом.

Порождение класса (в данном примере – класса `ThreadSafeQueue`) от шаблона класса, параметризованного этим же самым порождённым классом, – часто используемая идиома, характерная для языка C++, известная под названием CRTP[1]. Ключевую роль в этой идиоме играет переменная-член `derived` типа `ThreadSafeQueue<T>&`. Она представляет собой ссылку на текущий объект, приведённую к порождённому классу[2].

---

[1] Представленная в этой главе программа недостаточно показательна в качестве примера использования идиомы CRTP. В самом деле, поведение программы не изменится, если из шаблона класса `Monitor` сделать обычный класс, тогда отпадает необходимость в переменной-ссылке `derived`, появляется возможность обращаться к членам базового класса напрямую. Хороший материал для начального знакомства с идиомой CRTP на русском языке можно найти здесь: https://habr.com/ru/post/543098. – *Прим. перев.*

[2] Это утверждение крайне спорно. В идиоме CRTP объявление переменной-ссылки `derived` (или заменяющей её функции-члена) должно располагаться не в порождённом классе, как показано в данном примере, а в базовом – это даёт возможность из базового класса обращаться к членам порождённого класса, реализуя тем самым полиморфизм на этапе компиляции. Эта ключевая особенность, составляющая суть идиомы CRTP, в данном примере никак не используется. Объявление ссылки `derived` типа `ThreadSafeQueue<T>&` в порождённом классе никакой полезной нагрузки не несёт, так как указатель `this` в нём и без того имеет нужный тип. – *Прим. перев.*

Объект-монитор `safeQueue`, объявленный в строке 71, используется в двух лямбда-функциях: первая из них добавляет число в очередь, а другая извлекает его. `ThreadSafeQueue` представляет собой шаблон класса и может хранить значения произвольного типа. Начиная со строки 81 показано, как 100 потоков-клиентов, работая параллельно, добавляют по 100 случайных чисел из диапазона от 1 до 6 в очередь `safeQueue`, тем временем как другие 100 потоков выбирают из очереди каждый по 100 чисел. При этом программа выводит на печать числа и идентификаторы потоков.

File Edit View Bookmarks Settings Help

```
rainer@linux:~> monitorObject

1 140600200582912; 1 140600192190208; 5 140600183797504; 3 140600175404800; 4 140600167012096;
2 140600158619392; 1 140600150226688; 5 140600141833984; 5 140600133441280; 6 140600125048576;
3 140600116655872; 4 140600108263168; 5 140600099870464; 1 140600091477760; 4 140600083085056;
1 140600074692352; 5 140600066299648; 1 140600057906944; 3 140600049514240; 1 140600041121536;
3 140600032728832; 5 140600024336128; 4 140600015943424; 6 140600007550720; 6 140599999158016;
4 140599990765312; 1 140599982372608; 4 140599973979904; 3 140599965587200; 5 140599957194496;
6 140599940409088; 5 140599948801792; 2 140599932016384; 1 140599923623680; 5 140599915230976;
2 140599906838272; 4 140599898445568; 5 140599890052864; 6 140599881660160; 3 140599873267456;
2 140599864874752; 6 140599856482048; 5 140599848089344; 5 140599839696640; 4 140599831303936;
1 140599822911232; 4 140599814518528; 6 140599806125824; 2 140599797733120; 3 140599789340416;
5 140599780947712; 3 140599772555008; 2 140599764162304; 2 140599755769600; 3 140599747376896;
1 140599730591488; 3 140599722198784; 6 140599713806080; 6 140599705413376; 1 140599697020672;
6 140599688627968; 4 140599738984192; 4 140599680235264; 2 140599671842560; 6 140599663449856;
3 140599655057152; 2 140599646664448; 1 140599638271744; 6 140599629879040; 1 140599621486336;
4 140599613093632; 3 140599604700928; 2 140599596308224; 6 140599587915520; 4 140599579522816;
3 140599571130112; 6 140599562737408; 1 140599554344704; 5 140599545952000; 5 140599537559296;
5 140599529166592; 1 140599520773888; 1 140599512381184; 5 140599503988480; 6 140599495595776;
4 140599487203072; 5 140599478810368; 5 140599470417664; 6 140599462024960; 6 140599453632256;
2 140599445239552; 2 140599436846848; 3 140599428454144; 4 140599420061440; 4 140599411668736;
6 140599403276032; 3 140599394883328; 6 140599386490624; 2 140599378097920; 3 140599369705216;

rainer@linux:~>
```

rainer : bash

Объект-монитор в действии

CRTP: странно рекурсивный шаблон

Сокращение CRTP образовано от словосочетания «curiously recurring template pattern»[1] и означает часто используемую в языке C++ идиому, состоящую в том, что производный класс `Derived` порождается от класса-шаблона `Base`, в который класс `Derived` подставлен в качестве аргумента:

```
template<class T>
class Base{
  ....
};

class Derived : public Base<Derived>{
  ....
};
```

1 Первые два слова означают как «странно рекурсивный», так и «странно повторяющийся»; вторые два можно перевести как «шаблон работы с шаблонами» – английские слова «template» и «pattern» переводятся одинаково. – *Прим. перев.*

Ключ к пониманию идиомы CRTP состоит в том, что инстанцирование функций-членов выполняется ленивым способом. Это значит, что код функций-членов генерируется только тогда, когда это становится необходимо. У идиомы CRTP два основных применения.

- Статический полиморфизм – подход, позволяющий, подобно динамическому полиморфизму, через интерфейс базового класса вызывать реализацию функции в некотором порождённом классе, о котором базовый класс не знает. В отличие от динамического полиморфизма, однако, здесь выбор конкретной реализации осуществляется на этапе компиляции.
- Программирование примесей (англ. *mixin*). Это популярный подход, состоящий в разработке классов таким образом, чтобы пользователям легко было добавлять в них новый код[1]. В приведённом примере класс `ThreadSafeQueue` порождён от базового класса `Monitor` и получает все функции-члены этого класса.

В статье «Язык C++ ленив: CRTP»[2] эта идиома описана более подробно.

Шаблоны проектирования «Активный объект» и «Объект-монитор» похожи между собой, но отличаются рядом важных аспектов. Оба шаблона имеют целью синхронизацию доступа к совместно используемому объекту. Функции-члены активного объекта выполняются не в том потоке, который их вызвал, в отличие от объекта-монитора. Это означает, что активный объект обеспечивает более глубокую развязку между вызовом и выполнением функций-членов, и, следовательно, построенные на его основе системы проще поддерживать.

10.2.3.3. Материал для дальнейшего изучения

Более подробное изложение затронутых здесь вопросов можно найти во второй книге серии POSA.

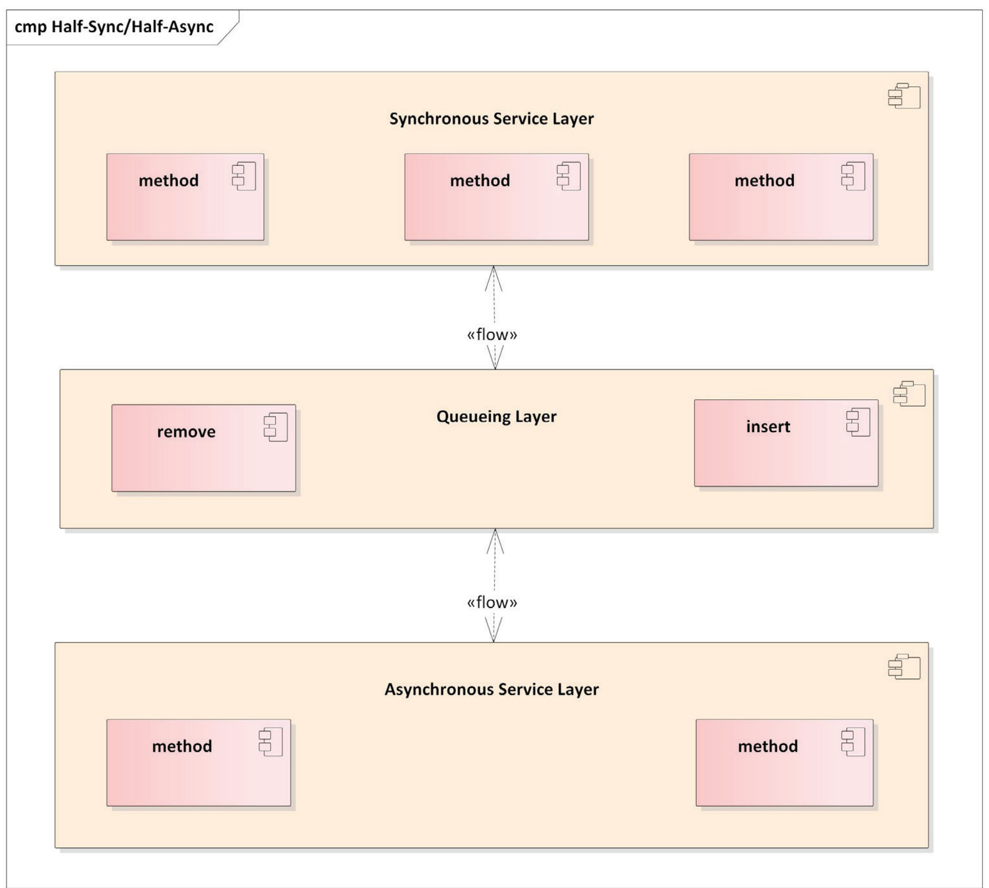
10.3. Полусинхронная архитектура

Этот архитектурный шаблон позволяет разделить части системы, занимающиеся синхронной и асинхронной обработками данных, без чрезмерного удара по её общей производительности. Этот шаблон предполагает появление двух дополнительных слоёв взаимодействия, один из которых служит для синхронной обработки, а другой – для асинхронной.

---

[1] Через порождённый класс, подставляемый в параметр шаблона базового класса, в последний впрыскивается код, определённый в порождённом классе. Тем самым поддерживается принцип открытозамкнутости (один из пятёрки фундаментальных принципов, известных под общим названием SOLID), состоящий в том, что класс должен быть закрыт для модификаций, но открыт для расширений. – *Прим. перев.*

[2] https://www.modernescpp.com/index.php/c-is-still-lazy.



Полусинхронная архитектура

Шаблон «Полусинхронная архитектура» часто применяется в циклах обработки событий серверных систем и в графических интерфейсах пользователя. Цикл обработки событий обычно состоит в том, чтобы принять событие от пользователя или клиента, поставить соответствующий запрос в очередь и затем обработать этот запрос синхронным образом в отдельном потоке. Асинхронный способ приёма запросов обеспечивает высокую производительность системы, а синхронная их обработка упрощает устройство обработчика. Асинхронная и синхронная части системы разведены по различным уровням её структуры, а очередь связывает их между собой. Асинхронный уровень включает в себя низкоуровневые механизмы наподобие прерываний, тогда как синхронный уровень отвечает за более высокоуровневые операции, такие как запросы к базе данных или манипуляции с файлами. Асинхронный и синхронный уровни могут общаться между собой через промежуточный уровень – диспетчер запросов.

10.3.1. Преимущества и недостатки

Преимущества полусинхронной обработки несомненны:

- чёткое разделение асинхронного и синхронного механизмов обработки. Низкоуровневые операции осуществляются на асинхронном, а высокоуровневые – на синхронном уровне;
- наличие промежуточного слоя ослабляет зацепление между синхронной и асинхронной подсистемами;
- чёткое разделение на уровни упрощает понимание, отладку, поддержку и расширение системы;
- блокировка в синхронном уровне не влияет на производительность асинхронного.

Данному архитектурному шаблону присущи также и некоторые недостатки:

- пересечение каждым запросом границ между уровнями влечёт определённые накладные расходы. Более того, это может требовать переключения контекста выполнения между режимами ядра и пользователя, поскольку асинхронные операции часто выполняются в пространстве ядра, а синхронные, как правило, в пространстве пользователя;
- строгое разделение синхронного и асинхронного уровней требует либо неизменяемости данных, либо их дорогостоящего копирования.

Полусинхронный принцип обработки часто применяется в системах демультиплексирования и диспетчеризации событий, подобных шаблонам «Реактор» и «Проактор».

10.3.2. Шаблон «Реактор»

Шаблон «Реактор» относится к событийно-управляемым архитектурам с демультиплексированием и диспетчеризацией запросов на множество параллельно работающих серверов. Этот шаблон известен также под названиями «Диспетчер» или «Оповещатель».

10.3.2.1. Требования

Предполагается, что система должна обрабатывать интенсивный поток запросов от клиентов. Каждый запрос обладает идентификатором типа, который позволяет назначить его определённому сервису-обработчику. При этом реактор должен удовлетворять следующим требованиям:

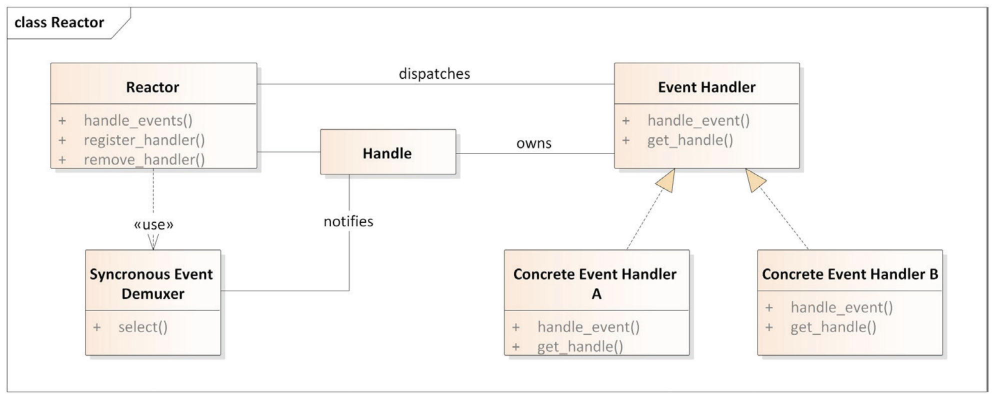
- не блокировать выполнение;
- обеспечивать максимальную пропускную способность, сводя к минимуму переключения контекстов, копирование данных и синхронизацию;

- быть легко расширяемым за счёт подключения новых или усовершенствованных сервисов-обработчиков;
- не использовать сложные механизмы синхронизации.

10.3.2.2. Решение

Для каждого поддерживаемого типа запроса нужно реализовать свой обработчик. Этот обработчик нужно зарегистрировать в реакторе. Реактор ждёт в синхронном режиме прихода очередного запроса от клиента и, когда событие приходит, использует демультиплексор событий, чтобы оповестить нужный сервис-обработчик.

10.3.2.3. Компоненты



Реактор

- Ресурсы:
  - соответствуют различным источникам входных данных и потребителям данных с выхода системы, например сетевым соединениям, открытым файлам или элементам графического интерфейса;
  - генерируют события, такие как входящее сообщение, чтение или запись, которые ставятся в соответствующую очередь.
- Демультиплексор синхронных событий:
  - в цикле ожидает очередное событие и отправляет его диспетчеру, когда его можно обработать на ресурсе без блокировки.
- Обработчик событий (абстрактный):
  - определяет интерфейс для обработки событий и, возможно, генерации новых событий;
  - определяет перечень функций системы.
- Обработчик событий (конкретный):
  - содержит ту или иную реализацию общего интерфейса обработчика событий, отвечая за обработку конкретного вида событий.

- Реактор:
  - имеет интерфейс для регистрации и удаления конкретных обработчиков событий;
  - использует демультиплексор синхронных событий, а также системные вызовы наподобие `select`[1], `epoll`[2], `WaitForMultipleObjects`[3] для ожидания событий;
  - ставит в соответствие событиям соответствующие конкретные обработчики;
  - управляет временем жизни цикла обработки событий.

Реактор (а не приложение) ожидает событий, чтобы затем демультиплексировать их и отправить на обработку походящему обработчику. Конкретные обработчики регистрируются в реакторе. Тем самым шаблон «Реактор» как бы переворачивает поток управления. Подобную инверсию часто называют «принципом Голливуда»[4].

Следующий фрагмент кода представляет собой цикл обработки событий в каркасе ACE (Adaptive Communication Environment)[5].

Цикл обработки событий в каркасе ACE

```
1  // CTRL c
2  SignalHandler *mutateTimer1= new SignalHandler( timerId1 );
3
4  // CTRL z
5  SignalHandler *mutateTimer2= new SignalHandler( timerId2 );
6
7  ACE_Reactor::instance()->register_handler( SIGINT, mutateTimer1);
8  ACE_Reactor::instance()->register_handler( SIGTSTP, mutateTimer2);
9
10 // “run” the timer.
11 Timer::instance ()->wait_for_event ();
```

Сначала определяются два обработчика сигналов, которые будут использованы для комбинаций клавиш **Ctrl**+**C** и **Ctrl**+**Z**, затем они регистрируются в реакторе. Последняя строка запускает цикл обработки событий.

10.3.2.4. Преимущества и недостатки

Шаблон «Реактор» имеет ряд сильных сторон:

- чёткое разделение на каркас и прикладную логику;
- модульная структура конкретных обработчиков событий;
- разделение интерфейса и реализации упрощает модификацию и расширение функций системы;

---

1 https://en.wikipedia.org/wiki/Select_(Unix).

2 https://en.wikipedia.org/wiki/Epoll.

3 https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects.

4 https://en.wikipedia.org/wiki/Inversion_of_control.

5 https://www.dre.vanderbilt.edu/~schmidt/ACE.html.

- общая структура системы хорошо подходит для параллельной работы.

К недостаткам шаблона можно отнести следующее.

- демультиплексор событий нуждается в системных вызовах;
- долго выполняющийся обработчик событий может заблокировать реактор;
- инверсия потока управления усложняет тестирование и отладку системы.

Шаблон полусинхронной обработки часто используется совместно с шаблоном «Реактор» для обслуживания запросов в отдельном потоке.

Шаблон «Проактор» представляет собой асинхронный вариант шаблона «Реактор». Если в шаблоне «Реактор» демультиплексирование события и отправка его конкретному обработчику выполняются синхронно, то в шаблоне «Проактор» это делается асинхронно.

10.3.3. Проактор

Шаблон «Проактор» позволяет событийно-управляемому приложению демультиплексировать и отправлять на обработку запросы, вызванные завершением асинхронных операций.

10.3.3.1. Требования

Производительность событийно-управляемых приложений, таких как серверы, часто удаётся повысить за счёт параллельной обработки нескольких асинхронных запросов. Чтобы добиться этого, приложение должно по возможности избегать синхронизации и переключений контекста. Кроме того, нужно обеспечить простоту подключения новых сервисов-обработчиков, а приложение в целом должно быть защищено от проблем, присущих параллельной и многопоточной обработке.

10.3.3.2. Решение

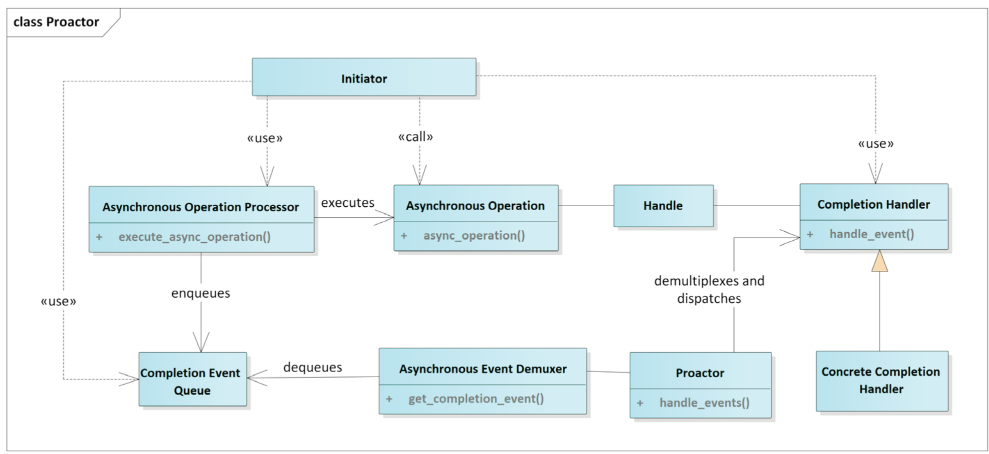
Приложение разбивается на две части: к одной относятся продолжительные операции, выполняющиеся асинхронно, а к другой – обработчики завершения, которым поступают результаты этих операций. Обработчик завершения работает подобно обработчику события в шаблоне «Реактор». Выполнение асинхронной операции часто возлагается на операционную систему. Подобно шаблону «Реактор», шаблон «Проактор» содержит цикл обработки событий.

Отличие же проактора от реактора состоит в следующем. В ответ на запрос клиента обработка запускается асинхронным образом и выполняется без блокирования вызывающего потока. Когда операция завершается, она помещает соответствующее событие в очередь. Проактор ожидает появления событий в очереди, используя демультиплексор асинхронных событий. Асинхронный демультиплексор удаляет событие завершения операции из

очереди, и проактор отправляет его на обработку подходящему обработчику завершения.

10.3.3.3. Компоненты

Шаблон «Проактор» состоит из девяти компонентов.



Проактор

- Ресурс:
  – служит обёрткой над объектом операционной системы – например, сетевым соединением, – способным генерировать сигнал о завершении операции.
- Асинхронная операция:
  – обычно продолжительная операция, выполняемая асинхронно. Например, это может быть операция чтения данных из сетевого соединения или отправка данных по сети.
- Исполнитель асинхронных операций:
  – запускает асинхронную операцию и ставит событие её завершения в очередь.
- Обработчик завершения:
  – общий интерфейс для обработки результатов асинхронной операции.
- Конкретный обработчик завершения:
  – реализует конкретный способ обработки результата той или иной асинхронной операции.
- Очередь событий завершения:
  – хранит события завершения до тех пор, пока демультиплексор не извлечёт их для дальнейшей обработки.

- Демультиплексор асинхронных событий:
  - блокируется, пока в очереди не появится событие завершения асинхронной операции;
  - выбирает событие из очереди и отправляет на обработку.
- Проактор:
  - вызывает демультиплексор асинхронных событий, чтобы получить из очереди очередное событие завершения операции;
  - распознаёт вид события и отправляет его соответствующему обработчику.
- Инициатор:
  - вызывает асинхронную операцию;
  - взаимодействует с исполнителем асинхронных операций.

10.3.3.4. Преимущества и недостатки

Преимущества шаблона «Проактор» таковы:

- асинхронные операции, не связанные со спецификой приложения, отделены от функциональности, составляющей суть данного конкретного приложения;
- интерфейс проактора можно использовать на различных операционных системах, подставляя лишь различные демультиплексоры асинхронных событий;
- приложениям не нужно запускать новые потоки, так как продолжительные асинхронные операции выполняются в потоке, который их вызывает;
- отсутствуют накладные расходы на переключение контекстов;
- поскольку приложение не запускает потоки, нет нужды в их синхронизации.

Недостатки у шаблона «Проактор» также имеются:

- чтобы извлечь выгоду из применения этого шаблона, операционная система должна обладать встроенной поддержкой асинхронных операций;
- вследствие отделения во времени и пространстве запуска операции от её завершения программу бывает весьма сложно отлаживать;
- вызовы асинхронных операций и хранение результатов их завершения требуют дополнительной памяти.

Библиотека асинхронного ввода-вывода Asio

Шаблон «Проактор» легко реализовать с помощью библиотеки Boost.Asio[1], которая может войти в стандарт C++ 23 в качестве библиотеки для работы с сетью. Разработанная Кристофером Колхоффом, библиотека Boost.Asio «представляет собой кроссплатформенный инструмент для работы с сетью и вводом-выводом на низком уровне и предлагает разработчикам целостную асинхронную модель, основанную на современных средствах языка C++».

[1] https://www.boost.org/doc/libs/1_69_0/doc/html/boost_asio.html.

10.3.4. Материал для дальнейшего изучения

Затронутые в этом разделе темы подробно изложены в документации к каркасу ACE (Adaptive Communication Environment), библиотеке Boost.Asio, а также во второй книге серии POSA.

10.4. Краткие итоги

Шаблоны «Активный объект» и «Монитор» помогают синхронизировать и выстроить в последовательность вызовы функций-членов объекта.

Шаблон «Полусинхронная архитектура» относится к уровню архитектуры и состоит в разделении частей системы, занимающихся синхронными и асинхронными операциями.

11. Эмпирические правила

В этой главе представлен несложный набор правил, позволяющих создавать хорошо структурированные и быстрые параллельные программы на современном языке C++. Многопоточные и параллельные вычисления лишь недавно появились в языке C++, и поэтому всё больше эмпирических правил открывается с каждым годом. Список правил, представленный в этой главе, нельзя считать исчерпывающим – он служит, скорее, отправной точкой для дальнейшего развития. Особенно это относится к параллельным алгоритмам стандартной библиотеки. На момент написания этой главы ещё слишком рано говорить о сформированных опытом правилах обращения с ними.

11.1. Общие правила

Начнём обзор с наиболее общих правил, справедливых и для атомарных переменных, и для потоков.

11.1.1. Рецензирование кода

Рецензирование кода должно быть непременной частью любого профессионального процесса разработки программ. Это становится особенно важным, когда дело идёт о параллельных вычислениях. Параллельному программированию внутренне присуща особая сложность, оно требует вдумчивости и опыта.

Чтобы сделать рецензирование более эффективным, свой код следует отсылать рецензентам как можно раньше для предварительного ознакомления. У рецензентов должно быть вдоволь времени для изучения кода до официального начала рецензирования. Стоит явно формулировать инварианты, которые должны сохраняться в ходе выполнения программы.

Если эти положения кажутся читателю неубедительными, рассмотрим один пример. Для этого нужно вспомнить гонку данных из программы `readerWriterLock.cpp`, приведённой в разделе 3.3.2.4.

Блокировка на чтение и запись

```
1  // readerWriterLock.cpp
2
```

```
3  #include <iostream>
4  #include <map>
5  #include <shared_mutex>
6  #include <string>
7  #include <thread>
8
9  std::map<std::string,int> teleBook{
10     {"Dijkstra", 1972}, {"Scott", 1976}, {"Ritchie", 1983}};
11
12 std::shared_timed_mutex teleBookMutex;
13
14 void addToTeleBook(const std::string& na, int tele){
15   std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
16   std::cout << "\nSTARTING UPDATE " << na;
17   std::this_thread::sleep_for(std::chrono::milliseconds(500));
18   teleBook[na]= tele;
19   std::cout << " ... ENDING UPDATE " << na << std::endl;
20 }
21
22 void printNumber(const std::string& na){
23   std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
24   std::cout << na << ": " << teleBook[na];
25 }
26
27 int main(){
28   std::cout << std::endl;
29
30   std::thread reader1([]{ printNumber("Scott"); });
31   std::thread reader2([]{ printNumber("Ritchie"); });
32   std::thread w1([]{ addToTeleBook("Scott",1968); });
33   std::thread reader3([]{ printNumber("Dijkstra"); });
34   std::thread reader4([]{ printNumber("Scott"); });
35   std::thread w2([]{ addToTeleBook("Bjarne",1965); });
36   std::thread reader5([]{ printNumber("Scott"); });
37   std::thread reader6([]{ printNumber("Ritchie"); });
38   std::thread reader7([]{ printNumber("Scott"); });
39   std::thread reader8([]{ printNumber("Bjarne"); });
40
41   reader1.join();
42   reader2.join();
43   reader3.join();
44   reader4.join();
45   reader5.join();
46   reader6.join();
47   reader7.join();
48   reader8.join();
49   w1.join();
50   w2.join();
51
52   std::cout << std::endl;
53
54   std::cout << "\nThe new telephone book" << std::endl;
55   for (auto teleIt: teleBook){
```

```
56      std::cout << teleIt.first << ": " << teleIt.second << std::endl;
57    }
58
59    std::cout << std::endl;
60  }
```

Проблема здесь в том, что операция обращения к элементу контейнера `teleBook[na]` в строке 24 может модифицировать контейнер[1]. Гонку данных можно спровоцировать, поставив читающий поток `reader8` перед остальными потоками-читателями. Автор часто использует эту программу в качестве упражнения на своих семинарах по языку C++. Задание состоит в том, чтобы обнаружить гонку данных. Лишь около 10 % участников укладываются в пять минут.

11.1.2. Сведение к минимуму совместного доступа к изменяемым данным

Совместный доступ к изменяемым данным нужно по возможности исключать сразу по двум причинам: для производительности и для безопасности. Безопасность здесь понимается главным образом как защита от гонки данных. Здесь сконцентрируем внимание на производительности – о корректности поговорим в следующем разделе.

В разделе 6.1 дан подробный анализ производительности различных средств параллельного программирования. Экспериментально измерено, сколь быстро происходит суммирование элементов вектора. Ниже показан главный участок кода, работающего в один поток.

Суммирование в один поток

```
constexpr long long size = 100000000;
std::cout << '\n';
std::vector<int> randValues;
randValues.reserve(size);
// random values
std::random_device seed;std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);

const unsigned long long sum = std::accumulate(
    randValues.begin(),
    randValues.end(),
    0);
```

Затем суммировать стали в четыре потока. Первое наивное решение состояло в том, чтобы накапливать сумму в единой переменной, общей всем потокам, с синхронизацией доступа.

[1] Если заданный ключ в ассоциативном контейнере отсутствует, операция индексирования добавляет запись с этим ключом и значением, сконструированным по умолчанию. – *Прим. перев.*

Многопоточное суммирование с блокировкой доступа к переменной

```
std::mutex myMutex;

void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        std::lock_guard<std::mutex> myLock(myMutex);
        sum += val[it];
    }
}
```

Затем эта программа была немного оптимизирована за счёт использования атомарной переменной.

Многопоточное суммирование с использованием атомарной переменной

```
void sumUp(
    std::atomic<unsigned long long>& sum,
    const std::vector<int>& val,
    unsigned long long beg,
    unsigned long long end)
{
    for (auto it = beg; it < end; ++it) {
        sum.fetch_add(val[it]);
    }
}
```

Существенного прироста производительности удалось добиться, суммируя без синхронизации элементы сегментов контейнера и затем складывая полученные результаты.

Раздельное суммирование и минимальная синхронизация блокировщиком

```
void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    std::lock_guard<std::mutex> lockGuard(myMutex);
    sum += tmpSum;
}
```

Показатели производительности впечатляют и дают ясное понимание: чем меньше в программе общего доступа к изменяемому состоянию, тем более эффективно используются ядра процессора.

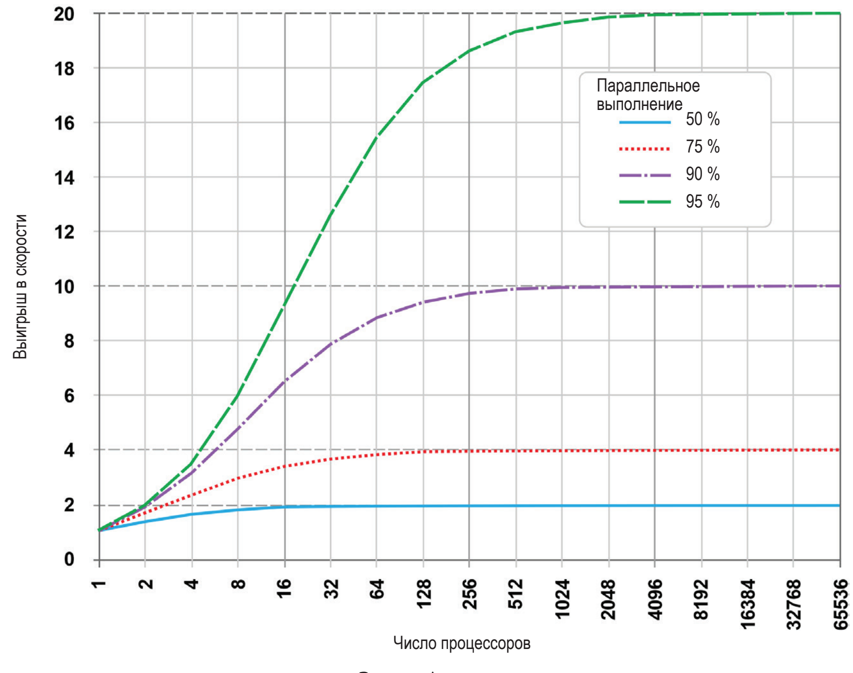
Способ	Время, с
Однопоточный	0,07
`std::lock_guard`	3,34
Атомарные переменные	1,34
Локальные переменные	0,03

11.1.3. Минимизация ожидания

Читатель мог слышать о законе Амдала[1]. Он устанавливает теоретический верхний предел выигрыша в скорости от распараллеливания алгоритма между несколькими процессорами или ядрами. Закон довольно прост. Если p – доля кода, которая может выполняться без синхронизации, то максимально возможный выигрыш скорости при неограниченном числе процессоров составляет $\frac{1}{1-p}$. Так, если 90 % кода могут выполняться параллельно, то на многопроцессорной системе можно получить не более, чем десятикратный выигрыш в скорости:

$$\frac{1}{1-p} = \frac{1}{1-0{,}9} = \frac{1}{0{,}1} = 10.$$

На это можно посмотреть и с другой стороны: если 10 % времени код должен выполняться последовательно из-за блокировки, общее время параллельного выполнения кода может сократиться лишь в 10 раз. Подчеркнём, что при этом предполагается наличие неограниченного ресурса процессоров. На следующем графике[2] закон Амдала показан наглядно.



Закон Амдала

[1] https://ru.wikipedia.org/wiki/Закон_Амдала.

[2] Рисунок из Википедии, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551.

Оптимальное число ядер, таким образом, сильно зависит от доли кода, который может выполняться параллельно без синхронизации. Например, если доля параллельного выполнения составляет 50 %, система вплотную приближается к теоретическому максимуму производительности при 16 процессорах. Дальнейшее наращивание их числа не может привести к сколько-нибудь заметному выигрышу производительности. Если же доля параллельного кода составляет 95 %, то к теоретическому максимуму производительности система приближается при 2048 процессорах.

11.1.4. Предпочтительное использование неизменяемых данных

Гонка данных – это ситуация, при которой по меньшей мере два потока получают доступ к одной и той же переменной, причём по крайней мере один из этих потоков пытается эту переменную модифицировать. Таким образом, необходимым условием гонки данных является наличие изменяемого состояния в совместном доступе. Следующий рисунок поясняет эту мысль.

Общий доступ \ Изменяемость	Нет	Да
Нет	Порядок	Порядок
Да	Порядок	Гонка данных

Изменяемость данных и общий доступ

Если данные, с которыми работают потоки, неизменяемы, гонка данных возникнуть не может. Нужно лишь гарантировать, что эти данные инициализируются потокобезопасным образом. Четыре способа потокобезопасной инициализации данных представлены в разделе 3.3.4:

- ранняя – до запуска потоков – инициализация данных;
- использование константных выражений;
- функция `std::call_once` с флагом `std::once_flag`;
- статическая переменная в локальной области видимости.

В языке C++ есть два способа объявлять неизменяемые данные: ключевые слова `const` и `constexpr`. Если спецификатор `const` позволяет во время выполнения сделать данные неизменяемыми в том или ином контексте[1], то спецификатор `constexpr` гарантирует, что данные вычисляются и подставляются на этапе компиляции и, следовательно, их использование потокобезопасно. На этапе компиляции могут инициализироваться даже данные пользовательских типов.

[1] Притом что в иных контекстах эти же данные в это же время могут трактоваться как изменяемые. – *Прим. перев.*

11.1.4.1. Пользовательские типы данных и константы этапа компиляции

Для того чтобы значения пользовательского типа могли вычисляться на этапе компиляции, этот тип должен удовлетворять некоторым условиям.

Так, `constexpr`-конструктор:

- должен вызываться с аргументами, которые сами являются константными выражениями;
- не может использовать обработку исключений[1];
- в стандарте C++ 11 должен быть объявлен как `default`, `delete` или иметь пустое тело.

Тип в целом:

- не должен иметь виртуальных базовых классов;
- каждый базовый класс и каждый нестатический член данных должен быть проинициализирован в списке инициализаторов конструктора или непосредственно в объявлении класса. Это означает, что все используемые для этого конструкторы базовых классов и данных-членов должны быть `constexpr`-конструкторами, а передаваемые им аргументы должны быть константными выражениями.

На сайте-справочнике cppreference.com приведён полный перечень правил для пользовательских типов, допускающих конструирование на этапе компиляции[2]. Чтобы подкрепить эту теорию практикой, разберём пример. В следующей программе объявляется класс `MyInt`. Этот класс отвечает всем перечисленным выше требованиям. Кроме того, его функция-член также имеет спецификатор `constexpr`.

Пользовательский тип с поддержкой неизменяемых данных

```
1  // userdefinedTypes.cpp
2
3  #include <iostream>
4  #include <ostream>
5
6  class MyInt{
7  public:
8      constexpr MyInt()= default;
9      constexpr MyInt(int fir, int sec): myVal1(fir), myVal2(sec){}
10     MyInt(int i){
11         myVal1= i - 2;
12         myVal2= i + 3;
13     }
14
15     constexpr int getSum() const { return myVal1 + myVal2; }
16
17     friend std::ostream& operator<< (
```

[1] Это ограничение отменено в стандарте C++ 20. – *Прим. перев.*

[2] https://en.cppreference.com/w/cpp/language/constexpr.

```
18          std::ostream &out, const MyInt& myInt)
19      {
20          out << "(" << myInt.myVal1 << "," << myInt.myVal2 << ")";
21          return out;
22       }
23
24  private:
25      int myVal1= 1998;
26      int myVal2= 2003;
27
28  };
29
30  int main(){
31      std::cout << std::endl;
32
33      constexpr MyInt myIntConst1;
34
35      constexpr int sec = 2014;
36      constexpr MyInt myIntConst2(2011, sec);
37      std::cout
38          << "myIntConst2.getSum(): "
39          << myIntConst2.getSum()
40          << std::endl;
41
42      int arr[myIntConst2.getSum()];
43      static_assert(
44          myIntConst2.getSum() == 4025,
45          "2011 + 2014 should be 4025" );
46
47      std::cout << std::endl;
48  }
```

Класс `MyInt` обладает двумя `constexpr`-конструкторами: это конструктор по умолчанию, объявленный в строке 8, и конструктор с двумя аргументами, объявленный в строке 9. Кроме этого, в классе объявлена одна[1] функция-член `getSum`, которая благодаря спецификатору `constexpr` также может вычисляться на этапе компиляции. Функция-член объявлена не только со спецификатором `constexpr`, но и со спецификатором `const`, потому что в стандарте C++ 14 (в отличие от стандарта C++ 11) `constexpr`-функция не является автоматически `const`-функцией[2].

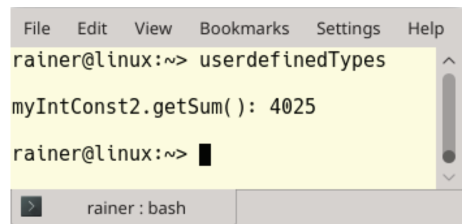
Проинициализировать переменные-члены `myVal1` и `myVal2` можно двумя способами. Во-первых, их значения можно установить с помощью списка

[1] Напомним, что дружественная (`friend`) функция не является членом класса. – *Прим. перев.*

[2] Это изменение в стандарте может требовать пояснений. Начиная со стандарта C++ 14 функция-член со спецификатором `constexpr` может модифицировать объект. В этом случае модифицированный объект будет вычислен на этапе компиляции. Предположительно затем из него будут извлечены некоторые данные другими `constexpr` функциями и подставлены в исполняемый код. – *Прим. перев.*

инициализаторов в конструкторе (строка 9). Во-вторых, начальные значения можно указать прямо в объявлении класса (строки 25 и 26). Если объект создаётся конструктором по умолчанию, используется второй способ, если же объект конструируется явно – первый способ имеет приоритет.

В строках 42 и 43 показано, что значение, возвращаемое `constexpr`-функцией, можно использовать в контексте, требующем константу этапа компиляции. Ниже показан результат работы программы.



Конструирование объекта на этапе компиляции

Следует подчеркнуть: к объекту данных, объявленному со спецификатором `constexpr`, на этапе выполнения можно применять только функции-члены со спецификатором `const`.

В языках функционального программирования наподобие Haskell вообще нет изменяемых объектов данных – поэтому на них особенно удобно писать параллельные программы.

11.1.5. Использование чистых функций

Язык Haskell называют чистым функциональным языком, поскольку в его основе лежат чистые функции. Чистой называется функция, которая при вызове с одинаковыми аргументами всегда возвращает одно и то же значение. У таких функций нет побочных эффектов, и, следовательно, они не могут менять состояние программы.

Чистые функции обладают важным преимуществом с точки зрения параллельного программирования. Их выполнение можно произвольно переупорядочивать или назначать отдельным потокам.

Функции в языке C++ по умолчанию не являются чистыми. Ниже показаны три чистые функции, каждая со своими особенностями.

```
int powFunc(int m, int n) {
    if (n == 0) return 1;
    return m * powFunc(m, n-1);
}
```

Здесь `powFunc` – это обычная функция, отрабатывающая на этапе выполнения программы.

```
template<int m, int n>
struct PowMeta {
    static int const value = m * PowMeta<m, n-1>::value;
};

template<int m>
struct PowMeta<m, 0> {
    static int const value = 1;
};
```

В этом фрагменте кода под именем `PowMeta` скрывается метафункция, вычисляемая на этапе компиляции.

```
constexpr int powConst(int m, int n) {
    int r = 1;
    for(int k = 1; k <= n; ++k) r *= m;
    return r;
}
```

Наконец, функция `powConst` благодаря спецификатору `constexpr` может вычисляться как на этапе выполнения, так и на этапе компиляции.

11.1.6. Отыскание правильных абстракций

Есть множество способов инициализации объекта-одиночки в многопоточной среде, некоторые из них разобраны в разделе 6.2. Так, можно опираться на такие средства стандартной библиотеки, как блокировщик `lock_guard` или триггер однократного вызова `std::call_once`, можно воспользоваться базовыми средствами языка, такими как статическая переменная, или же положить в основу решения атомарные переменные с семантикой захвата и освобождения. Семантика захвата и освобождения по ряду причин оказывается наиболее сложным решением. Она сложна в реализации и поддержке, также логику решения сложно объяснить коллегам. В противоположность этому широко известная реализация Мейерса гораздо проще в реализации и лучше по быстродействию.

Принцип выбора подходящей абстракции применим к самым разнообразным задачам. Так, вместо того чтобы своими руками реализовывать параллельный цикл для суммирования элементов контейнера, можно воспользоваться стандартным алгоритмом `std::reduce`. На вход ему нужно подать вызываемый объект с двумя аргументами и политику параллельного выполнения.

Чем тщательнее программист выбирает подходящие для задачи абстракции, тем меньше вероятность «выстрелить себе в ногу».

11.1.7. Использование статических анализаторов кода

В главе 6, посвящённой разбору учебных примеров, был показан инструмент CppMem (см. также главу 15). Это интерактивное средство позволяет исследовать поведение небольших фрагментов кода в соответствии с моделью памяти языка C++. Этот инструмент может принести программисту двоякую пользу. Во-первых, с его помощью можно проверить корректность своего кода. Во-вторых, он помогает углубить понимание модели памяти и, следовательно, вопросов многопоточного программирования.

11.1.8. Использование динамических анализаторов

Утилита ThreadSanitizer представляет собой детектор гонки данных для программ на языках C и C++. Эта утилита работает с компиляторами clang (начиная с версии 3.2) и GCC (с версии 4.8). Для её использования программу нужно собирать с ключом `-fsanitize=thread`. В следующей программе имеет место гонка данных.

Гонка данных при обращении к глобальной переменной

```
1  // dataRace.cpp
2
3  #include <thread>
4
5  int main(){
6
7    int globalVar{};
8
9    std::thread t1([&globalVar]{ ++globalVar; });
10   std::thread t2([&globalVar]{ ++globalVar; });
11
12   t1.join();
13   t2.join();
14
15 }
```

Потоки `t1` и `t2` обращаются к глобальной переменной одновременно, причём оба потока пытаются её модифицировать. Откомпилируем и запустим программу.

```
g++ -std=c++11 dataRace.cpp -fsanitize=thread -pthread -g -o dataRace
```

Запуск программы даёт результат, показанный на следующем рисунке.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@suse:~> dataRace
==================
WARNING: ThreadSanitizer: data race (pid=6764)
  Read of size 4 at 0x7fff031ca3bc by thread T2:
    #0 operator() /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f01)
    #1 _M_invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401b3d)
    #2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a51)
    #3 _M_run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x0000004019bc)
    #4 execute_native_thread_routine ../../../.././libstdc++-v3/src/c++11/thread.cc:83 (libstdc++.so.6+0x000000
0c11be)

  Previous write of size 4 at 0x7fff031ca3bc by thread T1:
    #0 operator() /home/rainer/dataRace.cpp:9 (dataRace+0x000000400eb9)
    #1 _M_invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401be7)
    #2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a8b)
    #3 _M_run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x000000401a06)
    #4 execute_native_thread_routine ../../../.././libstdc++-v3/src/c++11/thread.cc:83 (libstdc++.so.6+0x000000
0c11be)

  Location is stack of main thread.

  Thread T2 (tid=6767, running) created by main thread at:
    #0 pthread_create ../../../.././libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x00000002aaed)
    #1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++-v3/include/x86_64-pc
-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x0000000c14b4)
    #2 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_Stat
e> >, void (*)()) ../../../.././libstdc++-v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x0000000c14b4)
    #3 main /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f97)

  Thread T1 (tid=6766, finished) created by main thread at:
    #0 pthread_create ../../../.././libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x00000002aaed)
    #1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++-v3/include/x86_64-pc
-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x0000000c14b4)
    #2 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_Stat
e> >, void (*)()) ../../../.././libstdc++-v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x0000000c14b4)
    #3 main /home/rainer/dataRace.cpp:9 (dataRace+0x000000400f70)

SUMMARY: ThreadSanitizer: data race /home/rainer/dataRace.cpp:10 in operator()
==================
ThreadSanitizer: reported 1 warnings
rainer@suse:~>
rainer : bash
```

Обнаружение гонки данных утилитой ThreadSanitizer

Самая важная строка в этом тексте обведена красным. Гонка данных обнаружена в строке 10.

11.2. Работа с потоками

11.2.1. Общие вопросы многопоточного программирования

Напомним, что потоки – это основные блоки, из которых строятся параллельные программы.

11.2.1.1. Создание как можно меньшего числа потоков

Сколь дорого обходится создание потока? Весьма дорого! Именно с этим связано данное эмпирическое правило. Рассмотрим сначала типичные затраты памяти, а затем затраты на процедуру создания потока.

11.2.1.1.1. Затраты памяти

Объект типа `std::thread` представляет собой тонкую обёртку вокруг потока, находящегося под управлением операционной системы. Поэтому нужно выяснить, сколько памяти занимают потоки в ОС Windows и ОС семейства Posix[1]:

- для системы Windows, по заявлению её разработчиков[2], стек потока занимает 1 мегабайт;
- для систем семейства POSIX, согласно странице руководства `pthread_create`, стек потока имеет размер 2 мегабайта на архитектурах i386 и x86_64. Данные для различных архитектур приведены в следующей таблице.

Размер стека потока

Архитектура	Размер стека по умолчанию, Мб
i386	2
IA-64	32
PowerPC	4
S/390	2
Sparc-32	2
Sparc-64	4
x86_64	2

11.2.1.1.2. Затраты на создание потока

Автору не удалось найти в источниках данные о том, сколько времени занимает создание нового потока. Чтобы получить приблизительное представление об этом, понадобилось самостоятельно создать тестовую программу и измерить её производительность в системах Linux и Windows.

Для замера производительности использовался компилятор GCC 6.2.1 на настольной системе и cl.exe, входящий в состав среды Microsoft Visual Studio 2017, на переносном компьютере. В обоих случаях программа компилировалась с максимальным уровнем оптимизации: с флагом `O3` в системе Linux и с флагом `Ox` в системе Windows. Исходный текст этой программы показан ниже.

Программа для измерения скорости создания потоков

```
1  // threadCreationPerformance.cpp
2
```

[1] https://ru.wikipedia.org/wiki/POSIX_Threads.

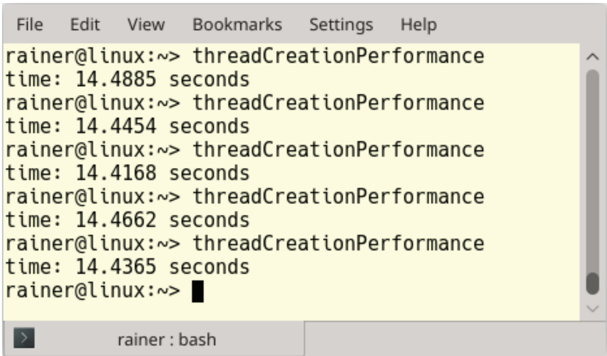
[2] https://docs.microsoft.com/en-us/windows/win32/procthread/thread-stack-size?redirectedfrom=MSDN.

```
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  static const long long numThreads= 1’000’000;
8
9  int main(){
10     auto start = std::chrono::system_clock::now();
11
12     for (volatile int i = 0; i < numThreads; ++i) std::thread([]{}).detach();
13
14     std::chrono::duration<double> dur=
15         std::chrono::system_clock::now() - start;
16     std::cout << “time: “ << dur.count() << “ seconds” << std::endl;
17 }
```

Эта программа создаёт один миллион потоков, в каждом из которых выполняется пустая лямбда-функция, и измеряет понадобившееся для этого суммарное время. В следующих подразделах представлены результаты её работы в двух операционных системах.

11.2.1.1.3. Система Linux

На следующем рисунке показан результат нескольких запусков тестовой программы.

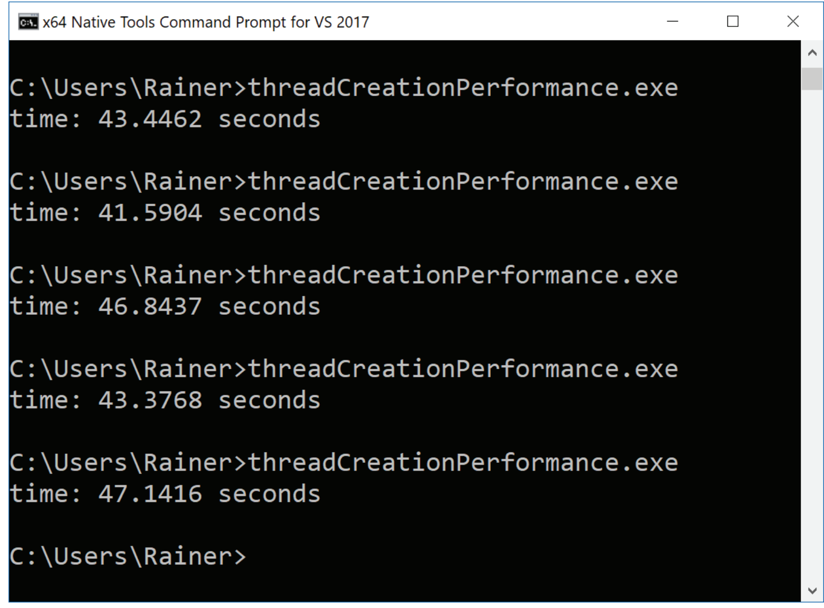


Затраты времени на создание потоков в системе Linux

Из этих данных следует, что в среднем на создание одного потока в системе Linux на данном компьютере тратится около 14,5 микросекунды.

11.2.1.1.4. Система Windows

Результат нескольких запусков программы показан на следующем рисунке.



Затраты времени на создание потоков в системе Windows

Таким образом, в среднем создание каждого потока в системе Windows на данном компьютере обошлось в 44 микросекунды.

Если посмотреть на полученные показатели производительности с противоположной стороны, получим, что за одну секунду на этом компьютере можно создать около 69 тысяч потоков в системе Linux и лишь 23 тысячи – в системе Windows.

11.2.1.2. Использование заданий вместо потоков

Рассмотрим следующую программу, в которой одно и то же вычисление выполняется двумя разными способами: в отдельном потоке и в асинхронном задании.

Сравнение асинхронных заданий с потоками

```
1   // asyncVersusThread.cpp
2
3   #include <future>
4   #include <thread>
5   #include <iostream>
6
7   int main(){
8     std::cout << std::endl;
9
10    int res;
```

```
11   std::thread t([&]{ res = 2000 + 11; });
12   t.join();
13   std::cout << "res: " << res << std::endl;
14
15   auto fut= std::async([]{ return 2000 + 11; });
16   std::cout << "fut.get(): " << fut.get() << std::endl;
17
18   std::cout << std::endl;
19 }
```

На примере этой программы можно увидеть несколько причин отдавать предпочтение заданиям, а не потокам. Так, в случае потоков необходим общий доступ к переменной и синхронизация такого доступа. Асинхронные задания, напротив, предоставляют удобный и безопасный канал для передачи результата наружу вычислений, будь то вычисленное значение, оповещение или исключение.

Если использовать расширенные фьючерсы, о которых шла речь в разделе 7.2, появляется возможность компоновать фьючерсы между собой, получая таким образом сложные схемы обработки данных. Для этого в первую очередь предназначено продолжние `then` и различные комбинации условий `when_all` и `when_any`.

11.2.1.3. Особая осторожность при отсоединении потока

Следующий небольшой фрагмент года должен приковать к себе внимание программиста:

```
std::string s{"C++11"}

std::thread t([&s]{ std::cout << s << '\n'; });
t.detach();
```

Поток `t` отсоединяется от кода, который его создаёт. При этом могут возникнуть два разных состояния гонки.

1. Время жизни потока `t` может превысить время выполнения блока, в котором он создаётся. В этом случае поток продолжит пользоваться ссылкой на объект `s`, уже прекративший своё существование.
2. Программа может начать своё завершение до того, как поток `t` выполнит свою работу. Это может привести к уничтожению глобального объекта `std::cout`, время жизни которого ограничено временем выполнения главного потока программы, в то время как поток `t` всё ещё пытается его использовать.

11.2.1.4. Предпочтительность потоков с автоматическим присоединением

Говорят, что поток `t` находится в присоединяемом состоянии, если вызываемый объект в нём присутствует и если не происходили вызовы `t.join()`

или `t.detach()`. Деструктор потока, находящегося в присоединяемом состоянии, аварийно завершает программу вызовом функции `std::terminate`. Чтобы не упустить необходимость присоединения потока, можно создать класс-обёртку над классом `std::thread`. Эта обёртка должна проверять, находится ли поток по-прежнему в присоединяемом состоянии, и, если это так, в своём деструкторе дожидаться его завершения с помощью функции `join`.

Программисту нет нужды создавать такую обёртку самостоятельно[1]. Можно воспользоваться готовым классом `scoped_thread` Энтони Вильямса или классом `gsl::joining_thread` из библиотеки guideline support library[2].

11.2.2. Управление доступом к данным

Все трудности многопоточного программирования начинаются там, где появляется совместный доступ к изменяемым данным.

11.2.2.1. Передача данных по значению

Рассмотрим следующий код:

```
std::string s{"C++11"}

std::thread t1([s]{ ... }); // действия над s
t1.join();

std::thread t2([&s]{ ... }); // действия над s
t2.join();

// действия над s
```

В поток `t1` строка передаётся путём копирования, поэтому поток-создатель и созданный им поток `t1` никак не связаны по данным. Ситуация оказывается иной для потока `t2`. Он получает ссылку на объект `s`. Это означает, что доступ к нему из главного потока и из потока `t2` необходимо синхронизировать. Это чревато ошибками и наносит урон производительности программы.

11.2.2.2. Использование умного указателя для совместного владения данными

Пусть имеется объект, к которому нужен доступ из нескольких потоков. Один из важнейших вопросов в этом случае – какой из потоков должен считаться владельцем объекта и, следовательно, отвечает за его уничтожение. Без ответа на него пришлось бы выбирать между утечкой памяти – в случае если занимаемую объектом память не освободит ни один поток – и неопределённым поведением – если потоки пытаются удалить объект более одного

---

1 Начиная со стандарта C++ 20 в стандартной библиотеке присутствует класс `std::jthread`, предназначенный для решения именно этой задачи. – *Прим. перев.*

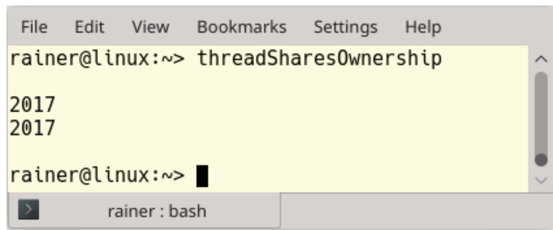
2 https://github.com/Microsoft/GSL.

раза. Неопределённое поведение чаще всего приводит к краху программы. Следующая программа иллюстрирует этот неразрешимый вопрос.

Неопределённое владение объектом

```
1  // threadSharesOwnership.cpp
2
3  #include <iostream>
4  #include <thread>
5
6  using namespace std::literals::chrono_literals;
7
8  struct MyInt {
9    int val{2017};
10   ~MyInt() {
11     std::cout << "Good Bye" << std::endl;
12   }
13 };
14
15 void showNumber(MyInt* myInt) {
16     std::cout << myInt->val << std::endl;
17 }
18
19 void threadCreator() {
20     MyInt* tmpInt= new MyInt;
21
22     std::thread t1(showNumber, tmpInt);
23     std::thread t2(showNumber, tmpInt);
24
25     t1.detach();
26     t2.detach();
27 }
28
29 int main() {
30     std::cout << std::endl;
31
32     threadCreator();
33     std::this_thread::sleep_for(1s);
34
35     std::cout << std::endl;
36 }
```

Этот пример намеренно сделан предельно простым. Главный поток засыпает на одну секунду, чтобы дать потокам `t1` и `t2` время завершиться. Конечно же, такой механизм синхронизации никак нельзя назвать приемлемым для разработки реальных программ, но для задач данного раздела он подходит. Ключевой вопрос теперь звучит так: какой поток должен удалить объект `tmpInt`? В этом примере возможных вариантов три: это может быть поток `t1`, `t2` или главный поток. Однако, поскольку невозможно предсказать продолжительность работы каждого потока, пришлось смириться с утечкой памяти. Поэтому деструктор объекта никогда не вызывается, в чём легко убедиться, запустив программу.



Неопределённое владение объектом

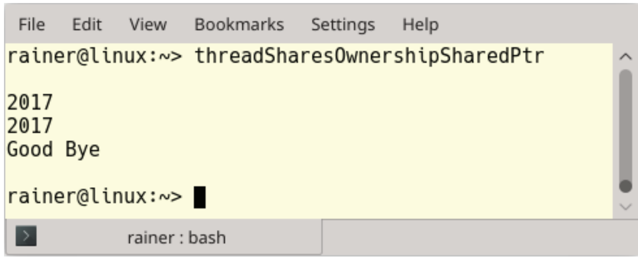
Управление временем жизни объекта становится довольно лёгкой задачей, если воспользоваться умным указателем `std::shared_ptr`. Ниже приведён текст реализации.

Управление временем жизни объекта через умный указатель

```
1  // threadSharesOwnershipSharedPtr.cpp
2
3  #include <iostream>
4  #include <memory>
5  #include <thread>
6
7  using namespace std::literals::chrono_literals;
8
9  struct MyInt{
10     int val{2017};
11     ~MyInt() {
12         std::cout << "Good Bye" << '\n';
13     }
14 };
15
16 void showNumber(std::shared_ptr<MyInt> myInt) {
17     std::cout << myInt->val << '\n';
18 }
19
20 void threadCreator() {
21     auto sharedPtr = std::make_shared<MyInt>();
22
23     std::thread t1(showNumber, sharedPtr);
24     std::thread t2(showNumber, sharedPtr);
25
26     t1.detach();
27     t2.detach();
28 }
29
30 int main(){
31     std::cout << '\n';
32
33     threadCreator();
34     std::this_thread::sleep_for(1s);
35
```

```
36     std::cout << '\n';
37 }
```

Эта программа отличается от предыдущей двумя важными деталями. Во-первых, вместо встроенного в язык типа указателя `MyInt*` в строке 21 теперь используется библиотечный тип-обёртка `std::shared_ptr`. Во-вторых, соответственно изменён тип функции `showNumber` (строка 16) – теперь она принимает аргумент типа умного указателя. Результат работы программы показан на рисунке. Из него видно, что деструктор для объекта вызывается – утечки памяти не происходит.



Управление временем жизни объекта через умный указатель

11.2.2.3. Сокращение времени блокировки

Пока какой-либо поток удерживает блокировку, никакой другой поток не может войти в критическую секцию, защищённую тем же мьютексом, и продолжить свою работу. Рассмотрим пример.

```
void setDataReadyBad(){
    std::lock_guard<std::mutex> lck(mutex_);
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << '\n';
    condVar.notify_one();
} // разблокировка

void setDataReadyGood(){
    mySharedWork = {1, 0, 3};
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    } // разблокировка

    std::cout << "Data prepared" << '\n';
    condVar.notify_one();
}
```

Функции `setDataReadyBad` и `setDataReadyGood` разными способами делают одно и то же – оповещают переменную условия о готовности некоторых данных. Переменная `dataReady` нужна для предотвращения ложных и по-

терянных пробеждений (см. раздел 3.5.2). Поскольку эта переменная имеет неатомарный тип, доступ к ней необходимо синхронизировать – для этого используется блокировщик lck. Для того чтобы удерживать блокировку как можно меньшее время, в функции setDataReadyGood сделан внутренний блок, при выходе из которого блокировка снимается, и остальные действия, не связанные с переменной dataReady, выполняются уже за пределами блокировки.

11.2.2.4. Обёртывание мьютекса в блокировщик

Не следует использовать мьютекс сам по себе, без обёртывания его в объект-блокировщик. Рассмотрим фрагмент кода.

```
std::mutex m;
m.lock();
// критическая секция
m.unlock();
```

Что-то неожиданное может произойти в критической секции[1] или программист забудет вставить в конце вызов функции unlock – итог один. Мьютекс останется запертым, и другие потоки будут заблокированы, что может привести к мёртвой блокировке всей системы.

Благодаря блокировщикам, которые автоматизируют управление мьютексом, риск попадания в мёртвую блокировку значительно уменьшается. В соответствии с идиомой RAII блокировщик захватывает мьютекс в конструкторе и освобождает в деструкторе. Тогда показанный выше фрагмент кода принимает следующий вид.

```
std::mutex m;
...
{
    std::lock_guard<std::mutex> lockGuard(m);
    // критическая секция
} // освободить мьютекс
```

Дополнительный блок, ограниченный фигурными скобками, гарантирует автоматическое уничтожение локального объекта и, следовательно, освобождение мьютекса.

11.2.2.5. Предпочтительный захват одного мьютекса

Логику программы следует продумывать таким образом, чтобы всякий раз требовался захват лишь одного мьютекса. Хотя, конечно, на практике бывают задачи, требующие одновременного захвата нескольких мьютексов, в этом случае заметно возрастает риск мёртвых блокировок, о чём пойдёт речь в одном из следующих разделов.

[1] Имеется в виду возможность выброса исключения – в этом случае функция unlock вызвана не будет. – *Прим. перев.*

11.2.2.6. Необходимость давать блокировщикам имена

Если объявить безымянный объект-блокировщик (например, типа `std::lock_guard`), как в следующем примере, он уничтожится немедленно после создания:

```
std::mutex m;
...
{
    std::lock_guard<std::mutex>{m};
    // критическая секция
}
```

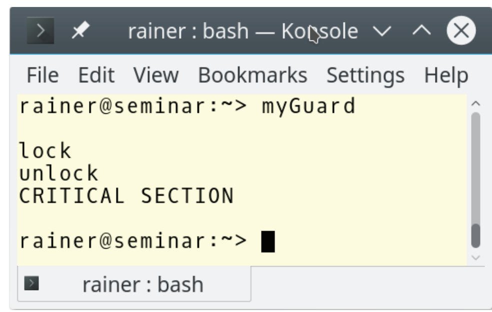
На первый взгляд, этот код выглядит вполне невинно, однако блокировщик уничтожается сразу после создания. Следовательно, критическая секция выполняется далее без всякой синхронизации. Напомним, что привязка времени жизни объекта к блоку кода, ограниченному фигурными скобками, составляет общепринятый приём программирования на языке C++, известный как идиома RAII. В частности, блокировщик должен захватывать мьютекс в конструкторе и освобождать в деструкторе. Следующий развёрнутый пример иллюстрирует странное поведение идиомы RAII, если объекту-обёртке не дать имени.

Ошибочная реализация идиомы RAII с безымянным блокировщиком

```
1  // myGuard.cpp
2  
3  #include <mutex>
4  #include <iostream>
5  
6  template <typename T>
7  class MyGuard {
8    T& myMutex;
9    public:
10     MyGuard(T& m):myMutex(m) {
11         myMutex.lock();
12         std::cout << "lock" << std::endl;
13     }
14 
15     ~MyGuard() {
16         myMutex.unlock();
17         std::cout << "unlock" << std::endl;
18     }
19 };
20 
21 int main() {
22   std::cout << std::endl;
23 
24   std::mutex m;
25   MyGuard<std::mutex> {m};                        // (1)
26   std::cout << "CRITICAL SECTION" << std::endl;   // (2)
```

```
27
28   std::cout << std::endl;
29 }                                                  // (3)
```

Конструктор и деструктор класса `MyGuard` вызывают функции `lock` и `unlock` мьютекса. Однако, поскольку объекту этого класса не дано имя, этот объект – временный, и его деструктор вызывается сразу после конструктора, в строке 25, а не при выходе из блока в строке 29. Следовательно, критическая секция в строке 26 выполняется без синхронизации. Запустив программу, можно убедиться, что разблокировка мьютекса происходит раньше, чем начинает выполняться критическая секция.



Ошибочная реализация идиомы RAII
с безымянным блокировщиком

11.2.2.7. Атомарный захват нескольких мьютексов

Если потоку необходимо захватить более одного мьютекса, программисту нужно соблюдать крайнюю осторожность, чтобы во всех местах они захватывались в одном и том же порядке. В противном случае неудачное стечение обстоятельств при перемежающемся выполнении потоков может спровоцировать мёртвую блокировку. Рассмотрим пример.

```
void deadLock(CriticalData& a, CriticalData& b){
    std::lock_guard<std::mutex> guard1(a.mut);
    // через некоторое время
    std::lock_guard<std::mutex> guard2(b.mut);
    // обработка объектов a и b
}
...
std::thread t1([&]{deadLock(c1, c2);});
std::thread t2([&]{deadLock(c2, c1);});
...
```

Каждому из потоков `t1` и `t2` нужны для работы два находящихся в общем доступе объекта типа `CriticalData`. При этом объект типа `CriticalData` содержит собственный мьютекс, который должен использоваться для синхронизации доступа к этому объекту. К сожалению, двум потокам, выполняющим

функцию deadLock, эти объекты передаются в различном порядке. Теперь программа находится в состоянии гонки. Если поток t1 успевает захватить первый мьютекс, но не второй, а поток t2 тем временем успевает захватить свой первый мьютекс, потоки до бесконечности блокируют друг друга, потому что первый мьютекс второго потока – это второй мьютекс первого потока, и наоборот.

К счастью, в стандартной библиотеке есть функция std::lock и класс std::unique_lock, который поддерживает отложенный захват. Показанный выше пример приобретает следующий вид:

```
void deadLock(CriticalData& a, CriticalData& b){
unique_lock<mutex> guard1(a.mut, defer_lock);
    // через некоторое время
    unique_lock<mutex> guard2(b.mut, defer_lock);
    std::lock(guard1, guard2);
    // обработка объектов a и b
}
...
std::thread t1([&]{deadLock(c1, c2);});
std::thread t2([&]{deadLock(c2, c1);});
...
```

В стандарте C++ 17 появляется новый вид блокировщика, тип std::scoped_lock, позволяющий блокировать произвольное число мьютексов атомарным образом. Теперь код становится ещё более очевидным.

```
void deadLock(CriticalData& a, CriticalData& b){
unique_lock<mutex> guard1(a.mut,defer_lock);
    std::scoped_lock(a.mut, b.mut);
    // обработка объектов a и b
}
...
std::thread t1([&]{deadLock(c1, c2);});
std::thread t2([&]{deadLock(c2, c1);});
...
```

11.2.2.8. Не вызывать неизвестный код под блокировкой

Вызов не вызывающей доверия функции unknownFunction из критической секции – лучший способ получить неопределённое поведение.

```
std::mutex m;
{
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable= unknownFunction();
}
```

Можно лишь строить предположения о том, что делает вызванная функция.

- Если она пытается захватить мьютекс m, получится неопределённое поведение (чаще всего на практике оно превратится в мёртвую блокировку).

- Если эта функция запускает новый поток, который, в свою очередь, пытается захватить мьютекс `m`, и ждёт его завершения – результатом становится мёртвая блокировка.
- Если функция захватывает другой мьютекс `m2`, возможна мёртвая блокировка, поскольку мьютексы `m` и `m2` захватываются неатомарным образом.
- Даже если вызываемая функция не пытается прямо или косвенно захватить мьютекс `m` и программа кажется безопасной, безопасность эта лишь кажущаяся. Другой программист может внести в функцию изменение или подключить новую версию библиотеки, в которой она определена. После этого можно делать ставки, какая из перечисленных здесь неприятностей случится первой.
- Функция может работать корректно, но медленно – в этом случае страдает производительность всей системы, поскольку вызов функции не позволяет освободить мьютекс.

Здесь можно воспользоваться локальной переменной, как показано ниже.

```
std::mutex m,
auto tempVar = unknownFunction();
{
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable = tempVar;
}
```

В самом деле, этот обходной манёвр устраняет все проблемы. Переменная `tmpVar` локальна, и поэтому не может стать жертвой гонки данных. Функцию `unknownFunction` теперь можно вызывать без какой-либо синхронизации. Наконец, время удержания блокировки сокращено до минимума: под блокировкой выполняется лишь присваивание значения из локальной переменной `tempVar` в общедоступную переменную `sharedVariable`.

11.2.3. Переменные условия

Идея синхронизации потоков посредством взаимных оповещений довольно проста, однако реализация этой идеи через переменные условия может оказаться весьма сложной. Основная причина состоит в том, что переменная условия не обладает состоянием. Для переменных условия характерны две проблемы (см. раздел 3.5.2):

- если переменная условия получает оповещение, оно может оказаться адресованным другой переменной условия – эта ситуация известна как ложное пробуждение;
- если переменная условия получает оповещение до того, как поток начинает ожидать оповещения через неё, это оповещение будет потеряно.

11.2.3.1. Обязательное использование предиката

Использование переменной условия без предиката способно привести к ложному пробуждению, потере пробуждения и к состоянию гонок. Рассмотрим пример.

Использование переменной условия без предиката

```
1  // conditionVariableLostWakeup.cpp
2
3  #include <condition_variable>
4  #include <mutex>
5  #include <thread>
6
7  std::mutex mutex_;
8  std::condition_variable condVar;
9
10 void waitingForWork(){
11     std::unique_lock<std::mutex> lck(mutex_);
12     condVar.wait(lck);
13     // обработка
14 }
15
16 void setDataReady(){
17     condVar.notify_one();
18 }
19
20 int main(){
21   std::thread t1(setDataReady);
22   std::thread t2(waitingForWork);
23
24   t1.join();
25   t2.join();
26 }
```

Если поток `t1` запускается раньше, чем поток `t2`, программа попадает в мёртвую блокировку. В самом деле, поток `t1` посылает своё оповещение до того, как поток `t2` становится готов его принять. Оповещение оказывается утерянным. Вероятность такого сценария довольно высока, так как поток `t1` создается первым и выполняет меньше действий.

Чтобы устранить эту проблему, достаточно лишь добавить логическую переменную `dataReady`. Она также защищает от ложного пробуждения, поскольку позволяет ожидающему потоку убедиться в том, что ожидаемое событие действительно произошло. Исправленный код показан ниже.

Использование переменной условия с предикатом

```
1  // conditioVarialbleLostWakeupSolved.cpp
2
3  #include <condition_variable>
4  #include <mutex>
5  #include <thread>
6
7  std::mutex mutex_;
8  std::condition_variable condVar;
9
10 bool dataReady{false};
11
```

```
12 void waitingForWork(){
13     std::unique_lock<std::mutex> lck(mutex_);
14     condVar.wait(lck, []{ return dataReady; });
15     // do the work
16 }
17
18 void setDataReady(){
19     {
20       std::lock_guard<std::mutex> lck(mutex_);
21       dataReady = true;
22     }
23     condVar.notify_one();
24 }
25
26 int main(){
27   std::thread t1(waitingForWork);
28   std::thread t2(setDataReady);
29
30   t1.join();
31   t2.join();
32 }
```

11.2.3.2. Замена переменных условия обещаниями и фьючерсами

Для однократных оповещений обещания и фьючерсы могут оказаться лучшим решением, чем переменные условия. Логику работы предыдущей программы можно реализовать следующим образом.

Оповещение через обещание и фьючерс

```
1  // notificationWithPromiseAndFuture.cpp
2
3  #include <future>
4  #include <utility>
5
6  void waitingForWork(std::future<void>&& fut) {
7      fut.wait();
8      // do the work
9  }
10
11 void setDataReady(std::promise<void>&& prom) {
12     prom.set_value();
13 }
14
15 int main() {
16   std::promise<void> sendReady;
17   auto fut = sendReady.get_future();
18
19   std::thread t1(waitingForWork, std::move(fut));
20   std::thread t2(setDataReady, std::move(sendReady));
```

```
21
22   t1.join();
23   t2.join();
24 }
```

Здесь количество текста, необходимое для того, чтобы выразить логику функционирования программы, сведено к абсолютному минимуму. Обещание `prom` путём вызова функции-члена `set_value` посылает оповещение фьючерсу `fut`, который ожидает на функции `wait`. В этой реализации не нужны ни мьютексы, ни блокировщики, так как в ней нет критических секций. Поскольку ложных и утерянных пробуждений здесь быть не может, не нужен также и предикат.

Однако если логика работы программы требует многократных оповещений, применить пару «обещание–фьючерс», увы, не удастся.

11.2.4. Обещания и фьючерсы

Обещания и фьючерсы часто представляют собой простую и удобную в использовании замену потокам и переменным условия.

11.2.4.1. Предпочтительность асинхронных заданий

Всегда, когда есть такая возможность, для создания асинхронного задания следует пользоваться функцией `std::async`. Например:

```
auto fut = std::async([]{ return 2000 + 11; });
// some time passes
std::cout << "fut.get(): " << fut.get() << '\n';
```

Вызов функции `std::async` как бы говорит системе: «Выполни это задание», и при этом не имеет значения, будет оно выполнено немедленно или позднее, будет оно выполняться в отдельном потоке, в пуле потоков или в том же потоке, который запрашивает это задание, и даже – выполнится оно на центральном процессоре или на графическом. Клиентский код заинтересован лишь в том, чтобы однажды в будущем забрать результат выполнения этого задания с помощью функции `get`.

С концептуальной точки зрения управление потоками становится для асинхронного задания лишь деталью реализации. Программист только указывает, *что* должно быть сделано, а не *как* это должно делаться.

11.3. Модель памяти

В основе многопоточного программирования лежит *хорошо определённая* модель памяти. Понимание принципов работы памяти помогает лучше понять сложности многопоточного программирования и пути их преодоления.

11.3.1. Недопустимость volatile-переменных для синхронизации

В отличие от языков C# и Java, в языке C++ семантика спецификатора `volatile` никак не связана с многопоточностью. В языках C# и Java спецификатор `volatile` похож на тип `std::atomic` из стандартной библиотеки языка C++ и означает, что значение переменной может меняться независимо различными потоками. В языке C++ этот спецификатор означает лишь, что значение переменной может измениться независимо от хода выполнения программ. Следовательно, операции над этой переменной запрещается оптимизировать, например хранить её в буфере.

11.3.1.1. Совет избегать неблокирующего программирования

Этот совет может показаться странным в книге, посвящённой параллельному программированию, особенно после того, как целая глава была посвящена моделям памяти и атомарным операциям. Причина проста. Неблокирующее программирование отличается крайней сложностью и высокой вероятностью допустить ошибку, оно требует от программиста чрезвычайно высокой квалификации. В частности, при реализации неблокирующих структур данных нужно опасаться так называемой проблемы ABA, о которой речь пойдёт в разделе 13.1. Таким образом, без крайней необходимости и надлежащих умений эту область лучше не трогать.

11.3.2. Использование шаблонов неблокирующего программирования

Если в программе обнаружено узкое место, устранить которое можно именно за счёт исключения блокировок, стоит применять общепринятые шаблоны неблокирующего программирования.

Использование атомарной переменной логического типа в качестве флага или атомарной целочисленной переменной в качестве счётчика вполне безопасно и не требует предосторожностей.

Для реализации сценария «производитель–потребитель» следует пользоваться потокобезопасными и неблокирующими контейнерами. Если контейнер потокобезопасен, значения можно помещать и извлекать, не заботясь о синхронизации. Тем самым программист перекладывает сложности неблокирующего программирования на библиотеку.

11.3.3. Использование гарантий, предоставляемых языком

Зачастую программист может вообще не тратить усилий на реализацию своих абстракций, если сам язык предоставляет необходимые гарантии. Напри-

мер, в разделе 6.2 были подробно рассмотрены различные способы потокобезопасной инициализации объекта, находящегося в свободном доступе. Среди них упоминались константные выражения, статические переменные в области видимости, а также функция `std::call_once` вместе с флагом `std::once_flag`. Реализация, основанная на статической переменной, оказалась и самой эффективной, и самой лаконичной. Таким образом, сам язык предоставляет готовое решение, избавляя программиста от необходимости изобретать своё.

Программируя на языке C++, можно реализацию своих абстракций строить на атомарных переменных, с усложнённой логикой, основанной на семантике захвата и освобождения. Не нужно этого делать без крайней необходимости. К таким средствам стоит прибегать, только если обнаружено узкое место, измерена производительность критического пути и доказано, что решение, реализованное своими руками, превосходит по быстродействию встроенных средств языка.

11.3.4. Не нужно изобретать велосипед

Создание потокобезопасных структур данных – довольно трудоёмкое занятие. Ещё труднее программировать неблокирующие структуры данных. Поэтому рекомендуется пользоваться существующими библиотеками: например, Boost.Lockfree[1] и CDS[2].

11.3.4.1. Библиотека Boost.Lockfree

Эта библиотека содержит три структуры данных.

- очередь с множественными поставщиками и множественными потребителями;
- стек с множественными поставщиками и множественными потребителями;
- свободная от ожиданий очередь с единственным поставщиком и единственным потребителем (также известная как кольцевой буфер).

11.3.4.2. Библиотека CDS

Название библиотеки CDS означает Concurrent Data Structures – параллельные структуры данных. В неё входит ряд контейнеров, как владеющих своими элементами, так и не владеющих – в этом случае данные, необходимые для управления временем жизни, хранятся в самих элементах. Напомним, что контейнеры из стандартной библиотеки языка C++ владеют элементами, автоматически управляют их временем жизни и не хранят в элементах свои служебные данные. Среди контейнеров, входящих в библиотеку CDS, имеются:

---

1 http://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html.
2 http://libcds.sourceforge.net/.

- стек (неблокирующий);
- очереди и очереди с приоритетами (неблокирующие);
- упорядоченные списки;
- упорядоченные множества и ассоциативные массивы (как неблокирующие, так и с блокировками);
- неупорядоченные множества и ассоциативные массивы (как неблокирующие, так и с блокировками).

11.4. Краткие итоги

- Параллельному программированию внутренне присуща высокая сложность, особенно это касается управления потоками и модели памяти.
- Общее правило параллельного программирования состоит в том, чтобы как можно больше данных делать неизменяемыми, а изменяемые данные, где только возможно, делать локальными. Оба пути устраняют возможность гонки данных.
- Всегда, когда это возможно, следует пользоваться асинхронными заданиями вместо явного управления потоками и переменных условия. Задания могут возвращать результаты вычислений, а также посылать оповещения.
- Неблокирующее программирование в общем случае чрезвычайно сложно, им могут заниматься лишь программисты высочайшей квалификации. Из этого правила есть ряд исключений, например атомарные счётчики.

Структуры данных

12. Структуры данных с блокировками

Если несколько потоков имеют совместный доступ к структуре данных, причём изменяемой, её необходимо защищать от одновременного доступа со стороны нескольких потоков. Такая защита может располагаться либо вне структуры данных, либо внутри её. Защита извне означает, что ответственность за предотвращение одновременного доступа к данным возлагается на клиентский код. Именно такой, внешний, подход использовался главным образом в предыдущих главах этой книги. Защита изнутри означает, что структура данных сама отвечает за собственную защиту. Если структура данных содержит механизмы, делающие невозможной гонку данных, её называют потокобезопасной. Именно о таких структурах данных с внутренней защитой пойдёт речь в этой главе.

Автор считает необходимым особо подчеркнуть: невозможно написать книгу о параллельном программировании без опоры на работы других авторов. Это справедливо и для главы о параллельных структурах данных. На настоящую книгу существенное влияние оказали книги «Искусство многопроцессорного программирования» Мориса Херлихи и Нира Шавита[1] и «Практика многопоточного программирования» Энтони Уильямса[2].

Прежде всего каковы общие принципы, которыми следует руководствоваться при разработке параллельных структур данных?

12.1. Общие соображения

Реализация потокобезопасных структур данных – особенная область программирования. Прежде чем погружаться в подробный разбор её специфических трудностей, дадим общую картину в виде списка вопросов, на которые нужно найти ответы при проектировании.

---

[1] https://www.oreilly.com/library/view/the-art-of/9780123705914/.

[2] https://www.manning.com/books/c-plus-plus-concurrency-in-action (*Уильямс Э.* Практика многопоточного программирования. СПб.: Питер, 2020. 640 с).

- **Стратегия блокировки**. Должна структура данных блокироваться крупными или мелкими частями? Блокировка структуры целиком проще в реализации, но усиливает конкуренцию потоков.
- **Гранулярность интерфейса**. Чем обширнее интерфейс потокобезопасной структуры данных, тем сложнее становится рассуждать о её использовании из нескольких потоков.
- **Типовые сценарии использования**. Если, например, разрабатываемая структура данных будет преимущественно использоваться для чтения, не стоит оптимизировать её для записи.
- **Избегание прорех**. Нельзя делать детали реализации доступными клиентскому коду.
- **Конкуренция потоков**. Насколько вероятны одновременные запросы к структуре данных из нескольких потоков?
- **Масштабируемость**. Как меняется быстродействие разрабатываемой структуры данных с ростом её размера или с ростом числа параллельных клиентов?
- **Инварианты**. Какие свойства и соотношения должны выполняться в структуре данных на всём протяжении её жизни?
- **Исключения**. Как структура данных должна вести себя с исключениями?

Конечно же, ответы на эти вопросы зависят друг от друга. Например, использование крупнозернистой стратегии блокировки упрощает рассуждения о гранулированном интерфейсе и об инвариантах. С другой стороны, это усиливает конкуренцию потоков и ухудшает масштабируемость.

12.1.1. Стратегии блокировки

Какую стратегию блокировки должны поддерживать структуры данных: крупнозернистую или мелкозернистую? Прежде всего уточним, что имеется в виду под этими названиями. Крупнозернистая блокировка означает, что блокируется вся структура данных целиком, так что в любой момент времени её может использовать лишь один поток. Шаблон проектирования «Потокобезопасный интерфейс», о котором шла речь в разделе 9.2.3, представляет собой типичный метод реализации крупнозернистой блокировки. Напомним принципы, лежащие в основе потокобезопасных интерфейсов.

- Все интерфейсные (т. е. с уровнем доступа `public`) функции-члены должны блокировать объект.
- Функции-члены, относящиеся к деталям реализации (с уровнем доступа `private` или `protected`), не должны захватывать блокировку.
- Интерфейсные функции-члены могут содержать вызовы лишь скрытых (`private` или `protected`), но не других интерфейсных функций-членов.

Шаблон «Потокобезопасный интерфейс» обладает двумя привлекательными свойствами: все общедоступные функции-члены гарантированно потокобезопасны и гарантированно свободны от мёртвых блокировок. Потокобезопасность обеспечивается тем, что каждая общедоступная функция-

член захватывает блокировку всего объекта. Отсутствие мёртвых блокировок следует из того, что каждая общедоступная функция, захватив блокировку, не может вызвать другую общедоступную функцию этого класса. Сказанное хорошо иллюстрирует следующий код.

Потокобезопасный интерфейс

```
1  // threadSafeInterface.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6
7  class Critical {
8
9  public:
10     void interface1() const {
11         std::lock_guard<std::mutex> lockGuard(mut);
12         implementation1();
13     }
14
15     void interface2() {
16         std::lock_guard<std::mutex> lockGuard(mut);
17         implementation2();
18         implementation3();
19         implementation1();
20     }
21
22 private:
23     void implementation1() const {
24         std::cout << "implementation1: "
25                   << std::this_thread::get_id() << std::endl;
26     }
27     void implementation2(){
28         std::cout << "    implementation2: "
29                   << std::this_thread::get_id() << std::endl;
30     }
31     void implementation3(){
32         std::cout << "        implementation3: "
33                   << std::this_thread::get_id() << std::endl;
34     }
35
36     mutable std::mutex mut;
37 };
38
39 int main(){
40     std::cout << std::endl;
41
42     std::thread t1([]{
43         const Critical crit;
44         crit.interface1();
```

```
45      });
46
47      std::thread t2([]{
48          Critical crit;
49          crit.interface2();
50          crit.interface1();
51      });
52
53      Critical crit;
54      crit.interface1();
55      crit.interface2();
56
57      t1.join();
58      t2.join();
59
60      std::cout << std::endl;
61  }
```

Потокобезопасные интерфейсы выглядят многообещающей идеей, но обладают и очевидными недостатками. Структура данных, построенная по принципу потокобезопасного интерфейса, представляет собой узкое место, поскольку в любой момент времени использовать её может лишь один поток. Это означает, что если в системе предполагается много параллельных потоков, работающих с одной структурой данных, стоит задуматься о более мелкозернистой стратегии блокировки. Например, вместо того чтобы защищать единой блокировкой весь односвязный список, можно блокировать доступ к отдельно взятым его элементам.

12.1.2. Гранулярность интерфейса

Предположим, ставится цель реализовать класс `ThreadSafeQueue` – блокирующую обёртку над стандартным контейнером `std::deque`. Следующий фрагмент кода должен дать общее представление об этом классе.

```
class ThreadSafeQueue{
    ...
public:
    bool empty() const;
    int pop();
    ...
private:
    std::deque<int> data;
    ...
};
```

Для простоты здесь показаны лишь две функции-члена. Функция `empty` возвращает логическое значение, которое показывает, пуст ли контейнер, а функция `pop` извлекает из контейнера верхний элемент и возвращает его. Гранулярность этого интерфейса выбрана неправильно! Почему? Рассмот-

рим случай, когда два потока одновременно пытаются обратиться к одному контейнеру ThreadSafeQueue[1].

```
ThreadSafeQueue threadSafeQueue;

std::shared_ptr<int> getHead() {
    if (!threadSafeQueue.empty()) {
        auto head = threadSafeQueue.pop();
    return head;
    }
    return std::shared_ptr<int>();
}
...
std::thread t1([&]{ auto res = getHead();
        ...
        });
std::thread t2([&]{ auto res = getHead();
        ...
        });
```

Этот код ведёт к состоянию гонки, результатом которого может стать неопределённое поведение. Между проверкой наличия элемента с помощью вызова функции empty и извлечением первого элемента очереди с помощью функции pop проходит некоторое время. В частности, операции, выполняемые двумя потоками, могут перемежаться следующим образом:

Поток t1	**Поток t2**
!threadSafeQueue.empty() == true	
	!threadSafeQueue.empty() == true
auto head = threadSafeQueue.pop();	
	auto head = threadSafeQueue.pop();

Если в очереди threadSafeQueue находится только один элемент, вызов функции pop из потока t2 попытается извлечь элемент из уже пустого контейнера. Хотя каждая функция-член, взятая по отдельности, потокобезопасна, их совместное применение обладает неопределённым поведением. Интерфейс класса возлагает ответственность за синхронизацию вызовов на клиента. Это далеко от идеала.

Изменение гранулярности функций-членов класса позволяет элегантно решить проблему. Довольно лишь объединить функции-члены empty и pop в одну.

---

[1] Обратим внимание, что функция pop возвращает не само значение, а умный указатель на него. Это сделано для того, чтобы корректно обрабатывать случай пустого контейнера – тогда функция pop должна вернуть пустой указатель. Следует отметить, что умный указатель – не самый удачный выбор для обработки отсутствующего значения. В стандартной библиотеке языка C++ присутствует шаблон класса std::optional, предназначенный именно для этой цели. – *Прим. перев.*

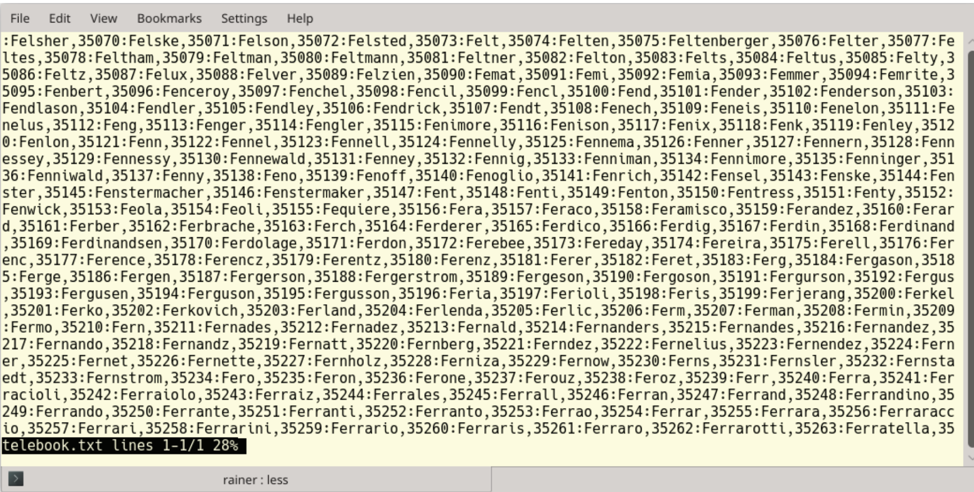
```
class ThreadSafeQueue{
    ...
public:
    std::shared_ptr<int> tryPop() {
        std::lock_guard<std::mutex> queLock(queMute);
        if (!data.empty()){
            auto head = data.pop();
            return head;
        }
        return std::shared_ptr<int>();
    }
    ...
private:
    std::deque<int> data;
    mutable std::lock_mutex queMutex;
    ...
};
```

12.1.3. Типовые сценарии использования

Самый частый сценарий использования большинства структур данных – это доступ для чтения. Блокировка в режиме писателя и читателя, о которой шла речь в разделе 3.3.2.4, позволяет оптимизировать структуру данных для чтения. Если для доступа к данным используется мьютекс типа `std::shared_timed_mutex`, помещённый в блокировщик `std::shared_lock`, имеет место режим совместного доступа, при котором несколько потоков могут читать общие данные; если же мьютекс поместить в блокировщик `std::lock_guard` или `std::unique_lock`, получается исключительный режим, при котором лишь один поток имеет доступ к данным и может их безопасно модифицировать.

Телефонная книга – хороший пример структуры данных, которая используется для чтения значительно чаще, чем для записи, и, следовательно, является идеальным кандидатом для применения блокировки описанного типа. Начнём с исключительной блокировки на каждую операцию, чтобы измерить начальные показатели производительности. Для эксперимента использовалась книга на примерно 89 тысяч записей. Десять потоков читают из неё каждый по 89 тысяч записей в произвольном порядке, а один поток дописывает цифру 1 к каждому десятому номеру телефона. При этом все потоки, конечно же, работают параллельно.

На следующем рисунке показан фрагмент телефонной книги. Можно видеть, что пары имя–номер разделены между собой двоеточием, а в каждой паре номер отделён от имени запятой.



Исходные данные телефонной книги

Следующая программа читает данные из этого файла в структуру данных в оперативной памяти – ассоциативный массив.

Телефонная книга с исключительной блокировкой

```
1   // exclusiveLockingTelebook.cpp
2
3   #include <chrono>
4   #include <fstream>
5   #include <future>
6   #include <iostream>
7   #include <mutex>
8   #include <random>
9   #include <regex>
10  #include <shared_mutex>
11  #include <sstream>
12  #include <string>
13  #include <unordered_map>
14  #include <vector>
15
16  using map = std::unordered_map<std::string, int>;
17
18  class TeleBook{
19      mutable std::mutex teleBookMutex;
20      mutable map teleBook;
21      const std::string teleBookFile;
22
23  public:
24      TeleBook(const std::string& teleBookFile_):
25      teleBookFile(teleBookFile_) {
26          auto fileStream = openFile(teleBookFile);
27          auto fileContent = readFile(std::move(fileStream));
```

```
28          teleBook = createTeleBook(fileContent);
29          std::cout
30              << "teleBook.size(): "
31              <<  teleBook.size()
32              << std::endl;
33      }
34
35      map get() const {
36          std::lock_guard<std::mutex> lockTele(teleBookMutex);
37          return teleBook;
38      }
39
40      int getNumber(const std::string& name) const {
41          std::lock_guard<std::mutex> lockTele(teleBookMutex);
42          return teleBook[name];
43      }
44
45      void setNewNumber(const std::string& name) {
46          std::lock_guard<std::mutex> lockTele(teleBookMutex);
47          teleBook[name]++;
48      }
49
50  private:
51      std::ifstream openFile(const std::string& myFile) {
52          std::ifstream file(myFile, std::ios::in);
53          if ( !file ) {
54              std::cerr << "Can't open file "+ myFile + "!" << std::endl;
55              exit(EXIT_FAILURE);
56          }
57          return file;
58      }
59
60      std::string readFile(std::ifstream file) {
61          std::stringstream buffer;
62          buffer << file.rdbuf();
63          return buffer.str();
64      }
65
66      map createTeleBook(const std::string& fileCont) {
67          map teleBook;
68
69          std::regex regColon(":");
70          std::sregex_token_iterator fileContIt(
71              fileCont.begin(), fileCont.end(), regColon, -1);
72          const std::sregex_token_iterator fileContEndIt;
73
74          std::string entry;
75          std::string key;
76          int value;
77          while (fileContIt != fileContEndIt) {
78              entry = *fileContIt++;
79              auto comma = entry.find(",");
```

```
80              key = entry.substr(0, comma);
81              value = std::stoi(entry.substr(
82                  comma + 1, entry.length() -1));
83              teleBook[key] = value;
84          }
85          return teleBook;
86      }
87  };
88
89  std::vector<std::string> getRandomNames(const map& teleBook) {
90      std::vector<std::string> allNames;
91      for (const auto& pair: teleBook) allNames.push_back(pair.first);
92
93      std::random_device randDev;
94      std::mt19937 generator(randDev());
95
96      std::shuffle(allNames.begin(), allNames.end(), generator);
97
98      return allNames;
99  }
100
101 void getNumbers(
102     const std::vector<std::string>& randomNames, TeleBook& teleBook)
103 {
104     for (const auto& name: randomNames) teleBook.getNumber(name);
105 }
106
107 int main() {
108     std::cout << std::endl;
109
110     // get the filename
111     const std::string myFileName = "tele.txt";
112     TeleBook teleBook(myFileName);
113
114     std::vector<std::string> allNames =
115         getRandomNames(teleBook.get());
116     std::vector<std::string> tenthOfAllNames(
117         allNames.begin(),
118         allNames.begin() + allNames.size() / 10);
119
120     auto start = std::chrono::steady_clock::now();
121
122     auto futReader0 = std::async(
123         std::launch::async,
124         [&]{ getNumbers(allNames, teleBook); });
125     auto futReader1 = std::async(
126         std::launch::async,
127         [&]{ getNumbers(allNames, teleBook); });
128     auto futReader2 = std::async(
129         std::launch::async,
130         [&]{ getNumbers(allNames, teleBook); });
131     auto futReader3 = std::async(
```

```
132         std::launch::async,
133         [&]{ getNumbers(allNames, teleBook); });
134     auto futReader4 = std::async(
135         std::launch::async,
136         [&]{ getNumbers(allNames, teleBook); });
137     auto futReader5 = std::async(
138         std::launch::async,
139         [&]{ getNumbers(allNames, teleBook); });
140   auto futReader6 = std::async(
141         std::launch::async,
142         [&]{ getNumbers(allNames, teleBook); });
143     auto futReader7 = std::async(
144         std::launch::async,
145         [&]{ getNumbers(allNames, teleBook); });
146     auto futReader8 = std::async(
147         std::launch::async,
148         [&]{ getNumbers(allNames, teleBook); });
149     auto futReader9 = std::async(
150         std::launch::async,
151         [&]{ getNumbers(allNames, teleBook); });
152
153     auto futWriter = std::async(
154         std::launch::async,
155         [&]{
156             for (const auto& name: tenthOfAllNames)
157                 teleBook.setNewNumber(name);
158         });
159
160     futReader0.get(), futReader1.get(), futReader2.get(),
161     futReader3.get(), futReader4.get(), futReader5.get(),
162     futReader6.get(), futReader7.get(), futReader8.get(),
163     futReader9.get(), futWriter.get();
164
165     std::chrono::duration<double> duration =
166         std::chrono::steady_clock::now() - start;
167
168     std::cout
169         << "Process time: "
170         << duration.count()
171         << " seconds"
172         << std::endl
173         << std::endl;
174 }
```

Начнём с конструктора класса `TeleBook` (строки 24–33). Он открывает файл, читает его содержимое и создаёт телефонную книгу. Функция `getRandomNames` (строки 89–99) генерирует случайную перестановку имён из телефонной книги. Каждое десятое имя заносится в контейнер `tenthOfAllNames` – именно у этих абонентов будет изменён телефонный номер. Перейдём теперь к наиболее интересной части программы – строкам с 122–168. Каждый из десяти фьючерсов `futReader0`–`futReader9` запускается в отдельном потоке. Каждый из них читает данные из телефонной книги в случайном порядке

посредством функции getNumbers (строки 101–105). Тем временем ещё один фьючерс futWriter выполняет в своём потоке лямбда-функцию, заданную в строках 155–158. Когда все фьючерсы завершают свою работу, строка 168 печатает общее время обработки. Остаётся подчеркнуть, что все функции из интерфейса класса TeleBook (функции get, getNumber и setNewNumber) используют для синхронизации общий мьютекс teleBookMutex (строка 19) типа std::mutex. Этот мьютекс объявлен с ключевым словом mutable, и, следовательно, его состояние можно менять в константных функциях-членах.

Теперь перейдём к оптимизации. Функции-члены get (строки 35–38) и getNumber (строки 40–43) не меняют состояние телефонной книги и, следовательно, могут пользоваться блокировкой в режиме чтения. Конечно же, эта оптимизация неприменима к функции setNewNumber (строки 45–48). Для краткости ниже приведена только подвергшаяся оптимизации часть класса TeleBook, так как остальной текст программы не меняется.

Телефонная книга с раздельной блокировкой на чтение и запись

```
1   // sharedLockingTelebook.cpp
2   ...
3   class TeleBook{
4       mutable std::shared_timed_mutex teleBookMutex;
5       mutable map teleBook;
6       const std::string teleBookFile;
7
8   public:
9       TeleBook(const std::string& teleBookFile_):
10      teleBookFile(teleBookFile_) {
11          auto fileStream = openFile(teleBookFile);
12          auto fileContent = readFile(std::move(fileStream));
13          teleBook = createTeleBook(fileContent);
14          std::cout
15              << "teleBook.size(): "
16              <<  teleBook.size()
17              << std::endl;
18      }
19
20      map get() const {
21          std::shared_lock<std::shared_timed_mutex>
22              lockTele(teleBookMutex);
23          return teleBook;
24      }
25
26      int getNumber(const std::string& name) const {
27          std::shared_lock<std::shared_timed_mutex>
28              lockTele(teleBookMutex);
29          return teleBook[name];
30      }
31
32      void setNewNumber(const std::string& name) {
33          std::lock_guard< std::shared_timed_mutex>
34              lockTele(teleBookMutex);
```

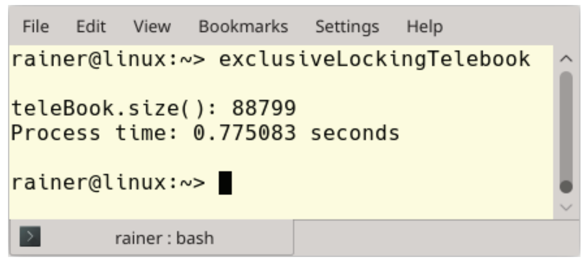
```
35          teleBook[name]++;
36      }
37  ...
```

Нет смысла сравнивать между собой производительность программы, собранной компилятором GCC под операционной системой Linux, с производительностью той же программы, собранной компилятором cl.exe под операционной системой Windows, если только они не запускаются на одинаковых компьютерах. Однако на каждой платформе можно сравнить производительность изначальной и оптимизированной версий программы. Результат оказывается весьма удивительным.

12.1.3.1. Производительность в ОС Linux

12.1.3.1.1. Исключительная блокировка

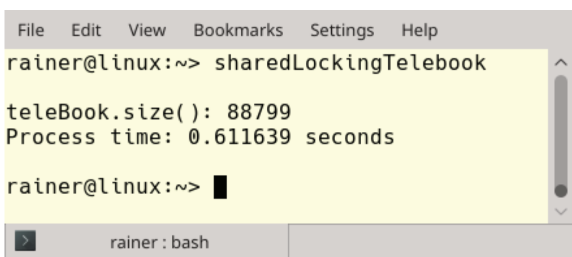
Результат запуска первоначальной версии программы показан на рисунке:



Первоначальная версия программы

12.1.3.1.2. Блокировка на чтение и запись

На следующем рисунке показан результат запуска усовершенствованной версии программы.



Версия с раздельной блокировкой чтения и записи

Таким образом, блокировка в режимах чтения и записи даёт прирост производительности примерно в 15 %. Это меньше, чем можно было ожидать, поскольку операции чтения и записи соотносятся как 100 к 1.

12.1.3.2. Производительность в ОС Windows

12.1.3.2.1. Исключительная блокировка

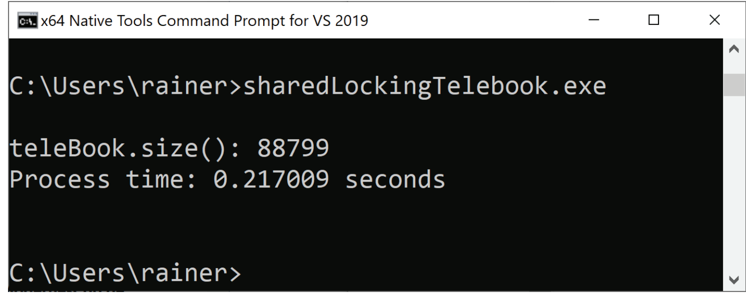
Результат работы неоптимизированной версии программы показан на рисунке.

```
x64 Native Tools Command Prompt for VS 2019

C:\Users\rainer>exclusiveLockingTelebook.exe

teleBook.size(): 88799
Process time: 0.121072 seconds


C:\Users\rainer>
```

Первоначальная версия программы

12.1.3.1.2. Блокировка на чтение и запись

Ниже показан результат работы программы с внесёнными изменениями.



Версия с раздельной блокировкой чтения и записи

Полученные в ОС Windows показатели сильно удивляют, ведь оптимизированная программа работает вдвое медленнее, чем программа с исключительной блокировкой на каждую операцию. Возможно, дело в высоких накладных расходах на более сложный мьютекс, которые превышают затраты на полезную работу под блокировкой.

12.1.4. Избегание прорех

Не следует показывать клиентам внутренние детали структур данных. Просачивание деталей реализации может произойти при передаче результатов функции по ссылке или указателю. Ещё один трудно поддающийся обнару-

жению путь к утечке деталей реализации открывается с возможностью передачи в структуру данных произвольного вызываемого объекта.

Дыра в интерфейсе

```
1  // lockDouble.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <mutex>
6
7  class LockDouble{
8      public:
9          double get() const {
10             std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
11             return lockDoub;
12         }
13
14         void set(double val) {
15             std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
16             lockDoub = val;
17         }
18
19         template <typename Func>
20         void apply(Func func) {
21             std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
22             func(lockDoub);
23         }
24
25     private:
26         double lockDoub{};
27         mutable std::mutex lockDoubMutex;
28 };
29
30 int main() {
31     LockDouble lck1;
32
33     auto fut1 = std::async([&lck1]{ lck1.set(20.11); });
34     auto fut2 = std::async(
35         [&lck1]{ std::cout << lck1.get() << std::endl; });
36
37     double* loophole = nullptr;
38     lck1.apply([&loophole](double& d) mutable { loophole = &d; });
39     *loophole = 11.22;
40
41     auto fut3 = std::async(
42         [&lck1]{ std::cout << lck1.get() << std::endl; });
43 }
```

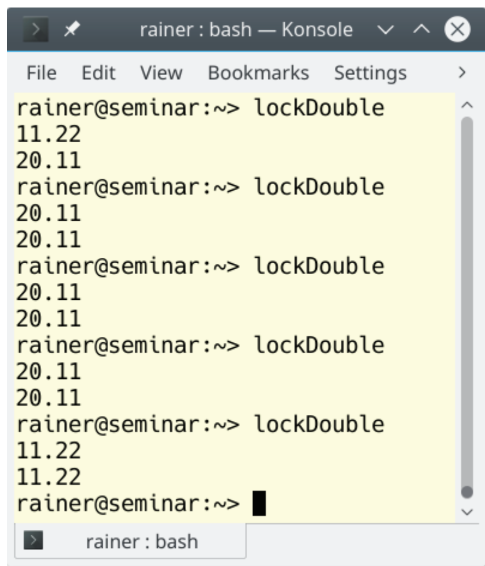
Класс `LockDouble` обладает понятным интерфейсом. Всякое обращение к переменной-члену `lockDoub` защищено мьютексом `lockDoubMutex`, помещённым в блокировщик `std::lock_guard`. Функция-член `get` возвращает копию завёр-

нутых в объект данных, а не ссылку на них. Если бы она возвращала ссылку на переменную `lockDoub`, клиент легко мог бы спровоцировать гонку данных, как показано ниже.

```
...
double& get() {
    std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
    return lockDoub;
}
...
LockDouble lck;
lck.set(22.11);
double& d = lck.get();
d = 11.22;
```

Конечно, проблема здесь состоит в том, что с помощью ссылки `d` клиент может изменить переменную-член `lockDoub`, доступ к которой должен быть синхронизирован мьютексом `lockDoubMutex`. Эту дыру в интерфейсе обнаружить довольно просто. Сложнее обстоит дело с другой.

Функция-член `apply` открывает дорогу бесконтрольному вмешательству клиента в детали реализации объекта. Так, в строке 38 объект принимает в себя лямбда-функцию, которую он должен применить к своему внутреннему состоянию, но эта функция похищает и выдаёт наружу указатель на переменную-член. Оператор присваивания в следующей строке модифицирует эту переменную без синхронизации. Здесь со всей очевидностью имеет место гонка данных. На следующем рисунке показан результат несинхронизированного доступа к переменной `lockDoub`.



Несинхронизированный доступ
к внутренним данным объекта

Инструмент ThreadSanitizer[1] показывает гонку данных в явном виде: диагностические сообщения приведены на следующем рисунке.

```
rainer : bash — Konsole
File  Edit  View  Bookmarks  Settings  Help
    #11 make_shared<std::__future_base::_Async_state_impl<std::thread::_Invoker<std::tuple<main()::<
lambda()> > >, void>, std::thread::_Invoker<std::tuple<main()::<lambda()> > > > /usr/local/include/c
++/8.2.0/bits/shared_ptr.h:723 (lockDouble+0x403e44)
    #12 _S_make_async_state<std::thread::_Invoker<std::tuple<main()::<lambda()> > > > /usr/local/inc
lude/c++/8.2.0/future:1705 (lockDouble+0x4036ea)
    #13 async<main()::<lambda()> > /usr/local/include/c++/8.2.0/future:1719 (lockDouble+0x402d20)
    #14 async<main()::<lambda()> > /usr/local/include/c++/8.2.0/future:1749 (lockDouble+0x4028b6)
    #15 main /home/rainer/lockDouble.cpp:37 (lockDouble+0x4026f9)

SUMMARY: ThreadSanitizer: data race /home/rainer/lockDouble.cpp:37 in operator()
==================
11.22
11.22
ThreadSanitizer: reported 1 warnings
rainer@seminar:~>
rainer : bash
```

Обнаружение гонки данных инструментом ThreadSanitizer

12.1.5. Конкуренция потоков

Как часто в разрабатываемую структуру данных поступают одновременные запросы от клиентов? Если конкуренция потоков незначительна, простейшие примитивы синхронизации наподобие мьютексов и блокировщиков оказываются достаточно быстрыми. В этом случае использование тонких и трудных для понимания механизмов, основанных на атомарных переменных, может оказаться чрезмерным. Прежде чем переходить к таким усложнённым решениям, стоит измерить производительность системы. Чтобы составить представление о величине накладных расходов на блокировку, проведём несложный эксперимент.

Этот эксперимент был описан в разделе 6.1. Программа заполняет вектор миллионом случайных чисел от 1 до 10 с равномерным распределением, затем несколькими способами подсчитывает их сумму. Для целей этого раздела интересны два из них.

12.1.5.1. Суммирование в один поток без синхронизации

Приведём ещё раз простейшее, основанное на цикле решение задачи.

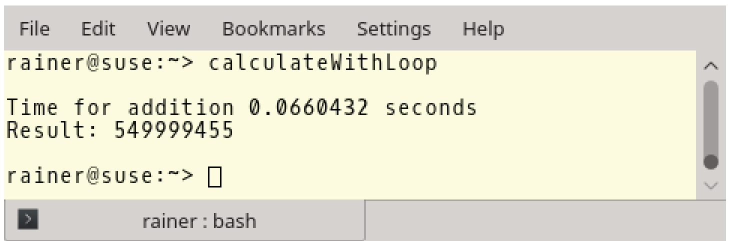
Суммирование в цикле по диапазону

```
1  // calculateWithLoop.cpp
2
3  #include <chrono>
```

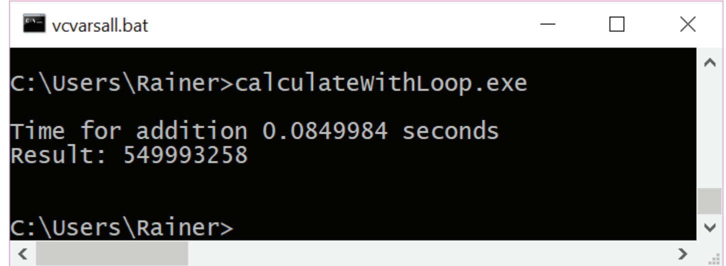
[1] https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

```
4  #include <iostream>
5  #include <random>
6  #include <vector>
7
8  constexpr long long size = 100000000;
9
10 int main(){
11   std::cout << std::endl;
12
13   std::vector<int> randValues;
14   randValues.reserve(size);
15
16   // random values
17   std::random_device seed;
18   std::mt19937 engine(seed());
19   std::uniform_int_distribution<> uniformDist(1, 10);
20   for (long long i = 0 ; i < size ; ++i)
21        randValues.push_back(uniformDist(engine));
22
23   const auto sta = std::chrono::steady_clock::now();
24
25   unsigned long long sum = {};
26   for (auto n: randValues) sum += n;
27
28   const std::chrono::duration<double> dur =
29         std::chrono::steady_clock::now() - sta;
30
31   std::cout << "Time for addition " << dur.count()
32             << " seconds" << std::endl;
33   std::cout << "Result: " << sum << std::endl;
34
35   std::cout << std::endl;
36 }
```

Результат работы этой программы с показателем производительности в ОС Linux и Windows показан на следующих рисунках.



Суммирование в цикле по диапазону в системе Linux



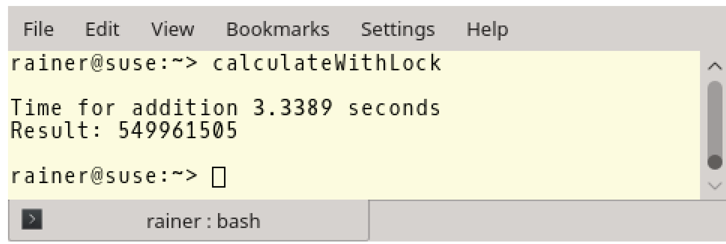
Суммирование в цикле по диапазону в системе Windows

12.1.5.2. Суммирование в один поток с синхронизацией

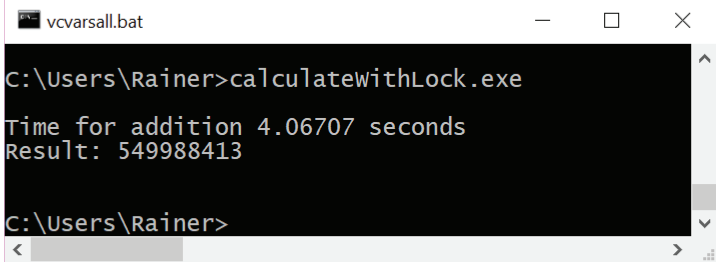
Для сравнения рассмотрим работу такого же цикла по диапазону с единственным отличием – синхронизацией с помощью мьютекса. Ниже показана только отличающаяся часть кода.

```
// calculateWithLock.cpp
...
  std::mutex myMutex;

  for (auto i: randValues){
      std::lock_guard<std::mutex> myLockGuard(myMutex);
      sum += i;
  }
```

Показатели производительности этой программы в двух ОС показаны на рисунках.



Суммирование с блокировкой в системе Linux



Суммирование с блокировкой в системе Windows

12.1.5.3. Анализ результатов измерений

Конечно, несинхронизированная программа работает в 50–150 раз быстрее, чем программа с блокировками. Казалось бы, числа говорят сами за себя и ясно свидетельствуют против стандартных блокировок. Однако следует учесть, что этот алгоритм захватывает мьютекс миллион раз, каждый раз выполняя под блокировкой одну очень быструю операцию. Если же захват блокировки требуется лишь изредка, стандартные примитивы могут оказаться наилучшим и достаточно быстрым решением.

12.1.6. Масштабируемость

Как меняются показатели производительности структуры данных с ростом числа использующих её параллельных потоков? Как они меняются с ограничением объёма данных? На эти два вопроса стоит найти точный ответ. Идеал масштабируемости – линейный рост пропускной способности структуры данных с ростом числа клиентов.

Пусть, например, есть потокобезопасная очередь, способная в каждый момент времени обслуживать одного производителя или одного потребителя. Все прочие производители или потребители вынуждены ждать, пока не закончит работу предыдущий клиент. Это ограничение преодолевается очередью, способной работать со множеством производителей и потребителей одновременно. Этот второй сценарий будем для краткости называть сценарием $n \times n$ (где n может, в частном случае, равняться единице). Такой сценарий имеет место, когда производители обеспечивают всех потребителей, и наоборот, ведь в противном случае либо производителям, либо потребителям пришлось бы тратить время на ожидание, что противоречит масштабируемости.

Если потокобезопасная очередь ограничена в размере, от неё нельзя ожидать идеальной масштабируемости, так как при достижении предельного числа элементов в очереди система застопоривается. Введение буфера между производителями и потребителями помогает ослабить связь между ними, но не решает проблему полностью.

Разберём ответ на два поставленных вопроса в случае потокобезопасной очереди `ThreadSafeQueue` из главы 10.

Объект-монитор

```
1  #include <condition_variable>
2  #include <functional>
3  #include <queue>
4  #include <iostream>
5  #include <mutex>
6  #include <random>
7  #include <thread>
8
9  template <typename T>
10 class Monitor {
11 public:
```

```
12      void lock() const {
13          monitMutex.lock();
14      }
15      void unlock() const {
16          monitMutex.unlock();
17      }
18
19      void notify_one() const noexcept {
20          monitCond.notify_one();
21      }
22      void wait() const {
23          std::unique_lock<std::recursive_mutex> monitLock(monitMutex);
24          monitCond.wait(monitLock);
25      }
26
27  private:
28      mutable std::recursive_mutex monitMutex;
29      mutable std::condition_variable_any monitCond;
30  };
31
32  template <typename T>
33  class ThreadSafeQueue: public Monitor<ThreadSafeQueue<T> > {
34  public:
35      void add(T val) {
36          derived.lock();
37          myQueue.push(val);
38          derived.unlock();
39          derived.notify_one();
40      }
41
42      T get() {
43          derived.lock();
44          while (myQueue.empty()) derived.wait();
45          auto val = myQueue.front();
46          myQueue.pop();
47          derived.unlock();
48          return val;
49      }
50
51  private:
52      std::queue<T> myQueue;
53      ThreadSafeQueue<T>& derived =
54          static_cast<ThreadSafeQueue<T>&>(*this);
55  };
56
```

Функция-член `add` (строка 35) добавляет элемент типа `T` к контейнеру `std::queue`, а функция-член `get` (строка 42) удаляет элемент из него. Класс `ThreadSafeQueue` воплощает шаблон «объект-монитор». Данный шаблон проектирования предписывает синхронизировать выполнение функций-членов объекта таким образом, чтобы не более одной интерфейсной функции выполнялось в любой момент времени. Когда производитель заканчивает

добавление нового элемента в очередь, он оповещает одного потребителя посредством переменной условия (строка 29). Рекурсивный мьютекс, объявленный в строке 28, защищает структуру данных от несогласованных модификаций. Этого описания должно быть достаточно, чтобы ответить на два поставленных вопроса[1] (больше подробностей об этом шаблоне проектирования можно найти в разделе 10.2).

- Вполне очевидно, что потребитель может оказаться заблокированным в строке 44, если очередь пуста.
- Ответ на второй вопрос ещё очевиднее, так как эта структура данных не ограничена в размере.

12.1.7. Инварианты

Инвариант – это условие или соотношение, которое должно оставаться истинным во всё время существования некоторой структуры данных. Например, сумма кредитов и дебетов по всем счетам в любой момент времени должна быть равна нулю. Должна – но в приведённой ниже программе не равняется.

[1] К приведённому здесь анализу масштабируемости можно многое добавить. Рассмотрим сначала структуру данных неизменного размера, причём для наших целей совершенно не важно, что она собой представляет – список, дерево или что-либо ещё. Пусть каждый запрос от клиента обслуживается одну секунду, не считая времени ожидания, и требует исключительной блокировки. Пусть клиентский поток в среднем делает к этой структуре данных один запрос в 100 секунд. Производительность системы с одним клиентским потоком возьмём за точку отсчёта. С появлением второго потока его запросы с в большинстве случаев не будут пересекаться во времени с запросами первого потока, однако с вероятностью 1 % одному из потоков придётся ждать завершения запроса от другого потока. Далее можно вычислить среднее время ожидания при наличии 2, 3 потоков и т. д. Наконец, когда число потоков-клиентов достигает известного предела, потоки будут проводить больше времени в ожидании, чем за полезной работой. Таким образом, первый аспект масштабируемости, связанный с числом потоков-клиентов, определяется временем выполнения запроса, интенсивностью потока запросов и тем, может ли структура данных блокироваться по частям, чтобы дать возможность параллельного выполнения нескольких клиентских запросов. Говоря о втором аспекте масштабируемости, лучше исходить из самого по себе размера структуры данных, а не из факта его ограниченности. Пусть, например, телефонная книга реализована в виде линейного списка пар «имя–номер». Тогда при увеличении размера телефонной книги в k раз среднее время поиска абонента по имени возрастёт также в k раз. Если же телефонная книга реализована в виде сбалансированного двоичного дерева, то двукратное увеличение объёма данных приводит лишь к линейному (на одну операцию сравнения) росту среднего времени поиска. Наконец, если телефонная книга выполнена на основе хеш-таблицы, то при достаточно хорошей хеш-функции и достаточно хороших данных время поиска практически не зависит от объёма хранимых данных (хотя имеется риск деградации производительности до уровня линейного списка при феноменальном невезении). Поэтому хеш-таблица обладает наилучшей масштабируемостью в среднем случае, сбалансированное дерево даёт наилучшую гарантированную масштабируемость, а линейный список следует на практике считать плохо масштабируемой структурой данных. – *Прим. перев.*

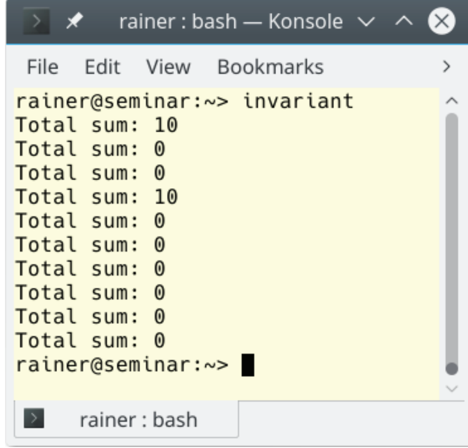
Нарушение инварианта

```
1  // invariant.cpp
2
3  #include <functional>
4  #include <iostream>
5  #include <mutex>
6  #include <numeric>
7  #include <random>
8  #include <thread>
9  #include <vector>
10
11 class Accounts{
12 public:
13     void deposit(int account) {
14         std::lock_guard<std::mutex> lockAcc(mutAcc);
15         accounts[account] += 10;
16     }
17
18     void takeOff(int account) {
19         std::lock_guard<std::mutex> lockAcc(mutAcc);
20         accounts[account] -= 10;
21     }
22
23     int getSum() const {
24         std::lock_guard<std::mutex> lockAcc(mutAcc);
25         return std::accumulate(accounts.begin(), accounts.end(), 0);
26     }
27
28 private:
29     std::vector<int> accounts = std::vector<int>(100, 0);
30     mutable std::mutex mutAcc;
31 };
32
33 class Dice{
34 public:
35     int operator()() { return rand(); }
36 private:
37     std::function<int()> rand = std::bind(
38         std::uniform_int_distribution<>(0, 99),
39         std::default_random_engine());
40 };
41
42 using namespace std::chrono_literals;
43
44 int main() {
45     constexpr auto TRANS = 1000;
46     constexpr auto OBS = 10;
47     Accounts acc;
48     Dice dice;
49
50     std::vector<std::thread> transactions(TRANS);
51     for (auto& thr: transactions) thr = std::thread([&acc, &dice]{
```

```
52          acc.deposit(dice());
53          std::this_thread::sleep_for(10ns);
54          acc.takeOff(dice()); }
55      );
56
57      std::mutex coutMutex;
58
59      std::vector<std::thread> observers(OBS);
60      for (auto& thr: observers) thr = std::thread([&acc, &coutMutex]{
61          std::lock_guard<std::mutex> coutLock(coutMutex);
62          std::this_thread::sleep_for(1ms);
63          std::cout << "Total sum: " << acc.getSum() << std::endl; }
64      );
65
66      for (auto& thr: transactions) thr.join();
67      for (auto& thr: observers) thr.join();
68  }
```

Класс `Accounts` держит под управлением 100 банковских счетов, которым в момент создания устанавливается начальное значение 0 – см. строку 29. Пользователь может пополнить счёт на 10 единиц с помощью функции `deposit` (строки 13–16). Функция-член `takeOff` (строки 18–21) позволяет снять со счёта также 10 единиц. Наконец, функция `getSum` (строки 23–26) помогает проверить соблюдение инварианта. Все эти функции-члены синхронизируются между собой посредством мьютекса `mutAcc`.

Посмотрим теперь, как можно вызвать нарушение инварианта. Создадим тысячу потоков (строки 50–55), каждый из которых кладёт деньги на случайно выбранный счёт и затем снимает их с другого случайно выбранного счёта. Ещё десять потоков следят за суммой по всем счетам (строки 59–64). Чем дольше выдерживается пауза между операциями `deposit` и `takeOff` в строке 53, тем больше вероятность наблюдать нарушение инварианта, как на следующем рисунке.



Нарушение инварианта

Конечно же, если бы вызовы функций `deposit` и `takeOff` находились в критической секции, это бы гарантировало истинность инварианта.

12.1.8. Исключения

Как должна вести себя структура данных в случае исключения? Ответ на этот вопрос зависит от применяемых средств управления потоками. Чаще всего используются следующие варианты:

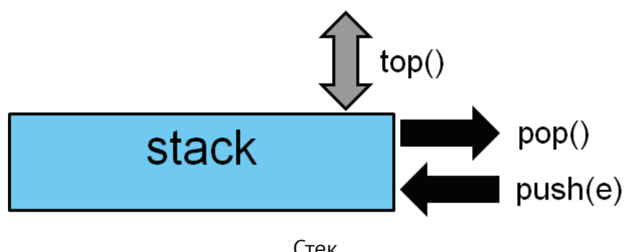
- тип `std::thread` – если из потока выходит необработанное исключение, в главном потоке вызывается функция `std::terminate`;
- функция `std::async`, типы `std::packaged_task` и `std::promise` допускают выброс исключения из асинхронного задания – в этом случае оно запоминается в объекте-фьючерсе для последующей обработки;
- параллельные алгоритмы стандартной библиотеки. Если при выполнении такого алгоритма с явно заданной политикой выполнения возникает исключение, вызывается функция `std::terminate`.

Функция `std::terminate` вызывает установленный в программе обработчик завершения `std::terminate_handler`[1]. По умолчанию это приводит к вызову функции `std::abort`[2], которая завершает программу аварийным образом.

Теперь пора посмотреть, как решение этих вопросов выглядит на практике. Наиболее часто используемые параллельные структуры данных – это стеки и очереди.

12.2. Потокобезопасный стек

Прежде всего ответим на вопрос, что такое стек. Стек – это структура данных, работающая по принципу «последним пришёл – первым ушёл» (аргл. LIFO, last in first out). В стандартной библиотеке языка C++ есть специальный шаблон класса[3], для использования которого нужно подключить заголовочный файл `<stack>`. Из его функций-членов главные три.



Стек

[1] https://en.cppreference.com/w/cpp/error/terminate_handler.
[2] https://en.cppreference.com/w/cpp/utility/program/abort.
[3] http://en.cppreference.com/w/cpp/container/stack.

Вызов `sta.push(e)` помещает новый элемент `e` на вершину стека `sta`, вызов `sta.pop()` удаляет из стека верхний элемент, а вызов `sta.top()` возвращает ссылку на верхний элемент. Кроме того, стек в стандартной библиотеке поддерживает операции сравнения на равенство, неравенство и порядок, а также функцию-член, возвращающую его размер. Следующий код иллюстрирует работу со стеком.

```
#include <stack>
...
std::stack<int> myStack;

std::cout << myStack.empty() << '\n'; // true
std::cout << myStack.size() << '\n';  // 0

myStack.push(1);
myStack.push(2);
myStack.push(3);
std::cout << myStack.top() << '\n';   // 3

while (!myStack.empty()){
    std::cout << myStack.top() << " ";
    myStack.pop();
}                                     // 3 2 1

std::cout << myStack.empty() << '\n'; // true
std::cout << myStack.size() << '\n';  // 0
```

Построим потокобезопасный стек шаг за шагом.

12.2.1. Упрощённая реализация

Первая реализация поддерживает только функцию-член `push`. Класс `ConcurrentStackPush` представляет собой лишь тонкую обёртку над стандартным типом `std::stack`.

Потокобезопасный стек с операцией вталкивания

```
1  // concurrentStackPush.cpp
2
3  #include <list>
4  #include <mutex>
5  #include <stack>
6  #include <string>
7  #include <vector>
8  #include <utility>
9
10 template <
11     typename T,
12     template <typename, typename> class Cont = std::deque>
13 class ConcurrentStackPush {
14 public:
15     void push(T val) {
```

```
16            std::lock_guard<std::mutex> lockStack(mutexStack);
17            myStack.push(std::move(val));
18        }
19
20        ConcurrentStackPush() = default;
21        ConcurrentStackPush(const ConcurrentStackPush&) = delete;
22        ConcurrentStackPush& operator= (const ConcurrentStackPush&)
23            = delete;
24
25    private:
26        mutable std::mutex mutexStack;
27        std::stack<T, Cont<T, std::allocator<T>>> myStack;
28    };
29
30    int main() {
31        ConcurrentStackPush<int> conStack;
32        conStack.push(5);
33
34        ConcurrentStackPush<double, std::vector> conStack2;
35        conStack2.push(5.5);
36
37        ConcurrentStackPush<std::string, std::list> conStack3;
38        conStack3.push("hello");
39    }
```

С точки зрения параллельного программирования здесь существенно, что функция-член `push` этого класса, объявленная в строках 15–18, копирует новый элемент в подобъект `myStack`, объявленный в строке 27. Благодаря мьютексу, объявленному в строке 26, эта операция потокобезопасна. Возможно, у читателя вызывает недоумение второй параметр шаблона. Параметр `Cont` представляет собой так называемый параметр-шаблон. По умолчанию в него подставляется шаблон `std::deque`. Этот параметр-шаблон задаёт контейнер, в котором будут храниться элементы стека. Шаблон, подставляемый в параметр `Cont`, сам принимает два параметра: тип элемента и аллокатор (вспомогательный тип, отвечающий за распределение памяти). Данный параметр-шаблон применяется в строке 27 – сначала в него подставляются тип элемента и тип аллокатора, а затем получившийся тип подставляется в качестве второго аргумента в шаблон `std::stack`. В строках 31, 34 и 37 показано, как объявить экземпляры шаблона `ConcurrentStackPush`, подставляя в него различные типы контейнеров – соответственно `std::deque`, `std::vector` и `std::list`.

Стандартные шаблоны `std::stack` и `std::queue` – это так называемые адаптеры контейнеров, поскольку они служат обёртками над другими контейнерными типами и придают им интерфейс стека или очереди.

Читатель может спросить, отчего в приведённом выше примере потокобезопасный стек используется в однопоточном контексте и почему не показан результат работы программы. Ответ прост: когда у класса есть лишь одна функция-член, этого слишком мало для сколько-нибудь интересного примера. Показанный здесь класс `ConcurrentStackPush` может служить лишь отправной точкой для полной реализации.

12.2.2. Полная реализация

Из общего определения стека можно заключить, что наша потокобезопасная реализация должна поддерживать три функции-члена: push, pop и top. На первый взгляд кажется, что задачу решит простое расширение класса ConcurrentStackPush, однако это оказывается не так. Рассмотрим следующий код.

Некорректная реализация

```
template <
    typename T,
    template <typename, typename> class Cont = std::deque>
class ConcurrentStackBroken {
public:
    void push(T val) {
        std::lock_guard<std::mutex> lockStack(mutexStack);
        myStack.push(std::move(val));
    }

    void pop() {
        std::lock_guard<std::mutex> lockStack(mutexStack);
        myStack.pop();
    }

    T& top() {
        std::lock_guard<std::mutex> lockStack(mutexStack);
        return myStack.top();
    }

    ConcurrentStackBroken() = default;
    ConcurrentStackBroken(const ConcurrentStackBroken&) = delete;
    ConcurrentStackBroken& operator= (const ConcurrentStackBroken&)
        = delete;

private:
    mutable std::mutex mutexStack;
    std::stack<T, Cont<T, std::allocator<T>>> myStack;
};
```

В классе ConcurrentStackBroken, помимо рассмотренной ранее функции push, появляются ещё две функции-члена: pop и top. Все они пользуются одним и тем же мьютексом. Однако данная реализация некорректна, в ней есть по меньшей мере две проблемы. Одна из них очевидна, другая требует большей проницательности.

Во-первых, функция-член top возвращает ссылку. Поток может безопасным образом получить ссылку на верхний элемент стека, а затем, за пределами блокировки, использовать её для модификации значения. Это приводит к гонке данных, как показано в следующем примере.

Гонка данных при возврате ссылки

```
ConcurrentStackBroken<int> conStack;
conStack.push(5);
```

```
auto fut1 = std::async(
    std::launch::async,
    [&conStack]{ conStack.top() += 5; });
auto fut2 = std::async(
    std::launch::async,
    [&conStack]{ std::cout << conStack.top() << std::endl; });
```

В этом фрагменте кода модификация верхнего элемента стека в первом асинхронном задании не синхронизирована с чтением его значения во втором задании.

Вторая проблема состоит в том, что поставленные подряд вызовы функций `top` и `pop` вместе не составляют неделимую операцию. Следующий фрагмент кода поможет понять, чем это грозит.

Небезопасное комбинирование потокобезопасных операций

```
ConcurrentStackBroken<int> conStack;
constexpr auto SENTINEL = std::numeric_limits<int>::min();
conStack.push(SENTINEL);
conStack.push(5);

auto safeRemove = [&conStack] {
    if (conStack.top() != SENTINEL) conStack.pop();
};

auto fut1 = std::async(std::launch::async, safeRemove);
auto fut2 = std::async(std::launch::async, safeRemove);
auto fut3 = std::async(std::launch::async, safeRemove);
```

В этой программе сделана попытка защититься от некорректной операции – взятия элемента из пустого стека. Для этого в стек первым помещается специальное значение `SENTINEL`, которое должно служить признаком близости к исчерпанию стека. Наличие в стеке хотя бы значения `SENTINEL` – инвариант, который должен сохраняться всё время существования объекта. Функция `safeRemove`, на первый взгляд, сохраняет инвариант – она изымает элемент из стека только тогда, когда этот элемент отличен от «сторожевого» значения `SENTINEL`. Проблема, однако, состоит в том, что при выполнении этой функции несколькими потоками может выполниться несколько операций `pop` подряд, исчерпав стек. Выполнение операций из тела функции `safeRemove` может при параллельном выполнении перемежаться, в том числе и показанным ниже образом.

Чередование операций между потоками

```
conStack.top() // fut1
conStack.top() // fut2
conStack.top() // fut3
conStack.pop() // fut1
conStack.pop() // fut2 // (2)
conStack.pop() // fut3 // (3)
```

Эта последовательность операций ведёт к фатальным последствиям. Сначала каждый из трёх контейнеров выполняет операцию `top` и убеждается, что

текущий элемент в стеке – не сторожевое значение и его можно безопасно изымать. Затем каждый поток переходит к выполнению своей операции pop. Вызов (2) удаляет из стека сторожевой элемент и тем самым нарушает инвариант. Тогда вызов (3) приводит к операции pop_back над пустым контейнером, а это влечёт за собой неопределённое поведение.

Трудности, связанные с использованием структуры данных из нескольких параллельных потоков, часто удаётся преодолеть за счёт изменения её интерфейса. В данном примере стоит поменять гранулярность интерфейса и превратить функции top и pop в одну, которую назовём topAndPop. Конечно, такое укрупнение функций в некоторой степени противоречит принципу единственной ответственности[1].

Потокобезопасный стек, полная реализация

```
1  // concurrentStack.cpp
2
3  #include <future>
4  #include <limits>
5  #include <iostream>
6  #include <mutex>
7  #include <stack>
8  #include <stdexcept>
9  #include <utility>
10
11 template <
12     typename T,
13     template <typename, typename> class Cont = std::deque>
14 class ConcurrentStack {
15 public:
16     void push(T val) {
17         std::lock_guard<std::mutex> lockStack(mutexStack);
18         myStack.push(std::move(val));
19     }
20
21     T topAndPop() {
22         std::lock_guard<std::mutex> lockStack(mutexStack);
23         if ( myStack.empty() )
24             throw std::out_of_range("The stack is empty!");
25         auto val = myStack.top();
26         myStack.pop();
27         return val;
28     }
29
30     ConcurrentStack() = default;
31     ConcurrentStack(const ConcurrentStack&) = delete;
32     ConcurrentStack& operator= (const ConcurrentStack&) = delete;
33
34 private:
35     mutable std::mutex mutexStack;
```

[1] https://ru.wikipedia.org/wiki/Принцип_единственной_ответственности.

```
36     std::stack<T, Cont<T, std::allocator<T>>> myStack;
37 };
38
39 int main() {
40     ConcurrentStack<int> conStack;
41
42     auto fut0 = std::async([&conStack]{ conStack.push(2011); });
43     auto fut1 = std::async([&conStack]{ conStack.push(2014); });
44     auto fut2 = std::async([&conStack]{ conStack.push(2017); });
45
46     auto fut3 = std::async([&conStack]{ return conStack.topAndPop(); });
47     auto fut4 = std::async([&conStack]{ return conStack.topAndPop(); });
48     auto fut5 = std::async([&conStack]{ return conStack.topAndPop(); });
49
50     fut0.get(), fut1.get(), fut2.get();
51
52     std::cout << fut3.get() << std::endl;
53     std::cout << fut4.get() << std::endl;
54     std::cout << fut5.get() << std::endl;
55 }
```

Функция `topAndPop` (строки 21–28) возвращает копию значения, снятого с верхушки стека, а не ссылку на него, как делала функция `top` из предыдущего примера. Попытка взять значение из пустого контейнера есть неопределённое поведение, поэтому такую ситуацию нужно предотвратить. В данном примере для этого выбрасывается исключение типа `std::out_of_range` (строка 24). Возврат специального значения-признака или использование типа `std::optional`[1] для возвращаемого значения функции тоже может быть подходящим вариантом. У возврата значения посредством копирования есть свой недостаток: если конструктор копирования выбросит исключение (например, типа `std::bad_alloc`[2]), значение окажется потерянным.

Вызов функции-члена `get` для всех ранее запущенных заданий в строке 50 гарантирует, что соответствующие асинхронные задания отработают. Если не задать политику запуска, эти задания могут быть выполнены ленивым способом в вызвавшем потоке. Под ленивым выполнением нужно понимать, что задание выполнится только тогда, когда поток потребует этого через объект-фьючерс с помощью функции `get` или `wait` (если это вообще произойдёт). Задания можно запустить также и в отдельных потоках, если явно указать политику запуска, как показано ниже.

Запуск заданий в отдельных потоках

```
auto fut0 = std::async(
    std::launch::asnyc, [&conStack]{ conStack.push(2011); });
auto fut1 = std::async(
    std::launch::asnyc, [&conStack]{ conStack.push(2014); });
auto fut2 = std::async(
    std::launch::asnyc, [&conStack]{ conStack.push(2017); });
```

---

[1] https://en.cppreference.com/w/cpp/utility/optional.

[2] https://en.cppreference.com/w/cpp/memory/new/bad_alloc.

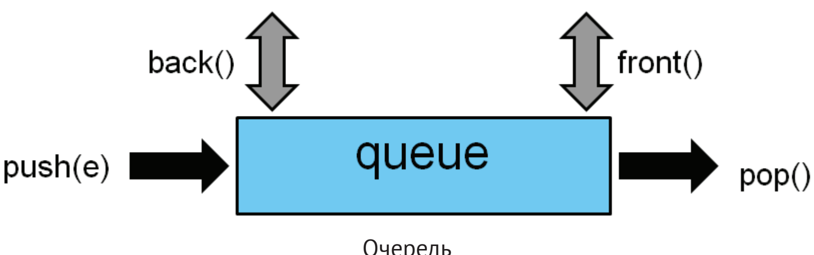
Результат запуска программы показан на следующем рисунке.

```
x64 Native Tools Command Prompt for VS 2019

C:\Users\rainer>concurrentStack.exe
2017
2014
2011

C:\Users\rainer>
```

Работа потокобезопасного стека

12.3. Потокобезопасная очередь

Как и в предыдущем разделе, начнём с ответа на вопрос, что такое очередь. Стандартный адаптер контейнера `std::queue`[1], объявленный в заголовочном файле `<queue>`, воплощает принцип «первым пришёл – первым ушёл» (англ. *FIFO – first in first out*). В его интерфейсе четыре основные функции.



Очередь

Функция-член `push` вставляет элемент в конец очереди, функция `pop` удаляет элемент из её начала; функция `back` позволяет получить ссылку на последний элемент очереди, а функция `front` – на первый. Кроме того, есть ещё вспомогательные функции, возвращающие размер очереди и позволяющие сравнивать очереди между собой. Пример использования очереди показан ниже.

```
#include <queue>
...
std::queue<int> myQueue;

std::cout << myQueue.empty() << '\n';      // true
std::cout << myQueue.size() << '\n';       // 0
```

[1] http://en.cppreference.com/w/cpp/container/queue.

```
myQueue.push(1);
myQueue.push(2);
myQueue.push(3);

std::cout << myQueue.back() << '\n';        // 3
std::cout << myQueue.front() << '\n';       // 1

while (!myQueue.empty()){
    std::cout << myQueue.back() << " ";
    std::cout << myQueue.front() << " : ";
    myQueue.pop();
}                                           // 3 1 : 3 2 : 3 3

std::cout << myQueue.empty() << '\n';       // true
std::cout << myQueue.size() << '\n';        // 0
```

Первый вариант реализации потокобезопасной очереди весьма похож на рассмотренную выше реализацию потокобезопасного стека.

12.3.1. Блокировка очереди целиком

Начнём с наиболее очевидной реализации. Объединим функции-члены `front` и `pop` в функцию `frontAndPop`. Функция `push` остаётся без изменений и добавляет элемент в конец очереди. Что же касается функции `back`, которая возвращает последний элемент очереди, она для очереди в общем случае не обязательна[1], и есть большие сомнения, нужно ли её вообще поддерживать в данном примере. Вот некоторые основания для этого.

1. Поддержка функции `back` налагает на программиста дополнительные обязательства, тогда как клиенты редко нуждаются в этой операции.
2. Комбинированная операция `backAndPush` должна была бы возвращать значение, которое было в очереди последним перед добавлением нового элемента в конец. Такая составная операция могла бы выглядеть многообещающе, но обладает двумя серьёзными недостатками. Во-первых, значение, бывшее в очереди последним, эта функция должна возвращать путём копирования, поскольку возврат по ссылке или указателю открыл бы дорогу гонке данных. Однако копирование может нанести удар по производительности. Во-вторых, конструктор копирования может выбросить исключение.
3. Наличие двух отдельных функций `back` и `push` делает возможной гонку данных. В самом деле, пусть клиентский код делает некоторые предположения, основываясь на значении последнего элемента. Тогда может возникнуть ситуация, подобная той, что наблюдалась бы при наличии отдельных операций `front` и `pop`. Здесь справедливы те же соображения, которые мы разобрали выше в связи с операциями `pop` и `top` для потокобезопасного стека. Хотя подобный способ использования очереди выглядит необычным и маловероятен на практике, лучше поступить

[1] https://ru.wikipedia.org/wiki/Очередь_%28программирование%29.

осторожно и не оставить даже теоретической возможности для такой ошибки.

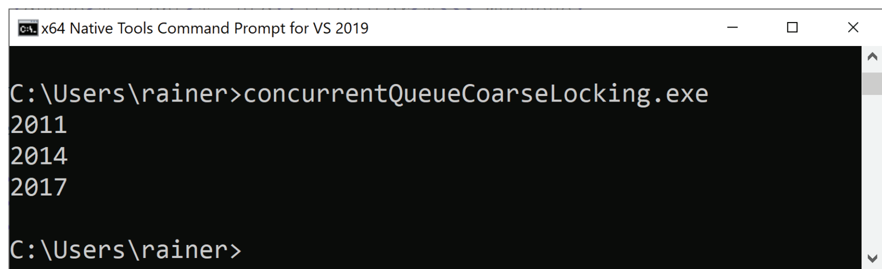
Реализация потокобезопасной очереди довольно проста и весьма похожа на реализацию стека. Код показан ниже.

Потокобезопасная очередь с полной блокировкой

```
1  // concurrentQueueCoarseLocking.cpp
2
3  #include <future>
4  #include <limits>
5  #include <iostream>
6  #include <mutex>
7  #include <queue>
8  #include <stdexcept>
9  #include <utility>
10
11 template <
12     typename T,
13     template <typename, typename> class Cont = std::deque>
14 class ConcurrentQueue {
15 public:
16     void push(T val) {
17         std::lock_guard<std::mutex> lockQueue(mutexQueue);
18         myQueue.push(std::move(val));
19     }
20
21     T frontAndPop() {
22         std::lock_guard<std::mutex> lockQueue(mutexQueue);
23         if ( myQueue.empty() )
24             throw std::out_of_range("The queue is empty!");
25         auto val = myQueue.front();
26         myQueue.pop();
27         return val;
28     }
29
30     ConcurrentQueue() = default;
31     ConcurrentQueue(const ConcurrentQueue&) = delete;
32     ConcurrentQueue& operator= (const ConcurrentQueue&) = delete;
33
34 private:
35     mutable std::mutex mutexQueue;
36     std::queue<T, Cont<T, std::allocator<T>>> myQueue;
37 };
38
39 int main() {
40     ConcurrentQueue<int> conQueue;
41
42     auto fut0 = std::async([&conQueue]{ conQueue.push(2011); });
43     auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
44     auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });
45
```

```
46      auto fut3 = std::async(
47          [&conQueue]{ return conQueue.frontAndPop(); });
48      auto fut4 = std::async(
49          [&conQueue]{ return conQueue.frontAndPop(); });
50      auto fut5 = std::async(
51          [&conQueue]{ return conQueue.frontAndPop(); });
52
53      fut0.get(), fut1.get(), fut2.get();
54
55      std::cout << fut3.get() << std::endl;
56      std::cout << fut4.get() << std::endl;
57      std::cout << fut5.get() << std::endl;
58  }
```

Без пространных пояснений покажем сразу результат запуска программы.



Работа потокобезопасной очереди

Можно ли считать задачу полностью решённой? Нет, так как у представленной здесь реализации имеется потенциал для оптимизации.

12.3.2. Раздельная блокировка концов очереди

В отличие от стека, где добавление и изъятие элементов выполняются с одного и того же конца контейнера, в случае очереди операции `push` и `pop` работают с разными его концами.

12.3.2.1. Некорректная реализация

Вместо того чтобы брать блокировку на всю очередь посредством единого мьютекса, можно было бы попытаться использовать отдельные мьютексы для начала и конца очереди в надежде, что это уменьшит общее количество синхронизаций.

Некорректная реализация очереди с раздельной блокировкой

```
template <
    typename T,
    template <typename, typename> class Cont = std::deque>
```

```
class ConcurrentQueue {
public:
    void push(T val) {
        std::lock_guard<std::mutex> lockQueue(mutexBackQueue);
        myQueue.push(std::move(val));
    }

    T frontAndPop() {
        std::lock_guard<std::mutex> lockQueue(mutexFrontQueue);
        if (myQueue.empty())
            throw std::out_of_range("The queue is empty!");
        auto val = myQueue.front();
        myQueue.pop();
        return val;
    }

    ConcurrentQueue() = default;
    ConcurrentQueue(const ConcurrentQueue&) = delete;
    ConcurrentQueue& operator= (const ConcurrentQueue&) = delete;

private:
    mutable std::mutex mutexFrontQueue;
    mutable std::mutex mutexBackQueue;
    std::queue<T, Cont<T, std::allocator<T>>> myQueue;
};
```

Эта реализация содержит ошибку, которая делает её полностью некорректной. Если очередь пуста, её начало и конец совпадают, и операции `push` и `pop` работают над одним и тем же элементом, что приводит к гонке данных. Добавление в очередь фиктивного элемента, который всегда разделяет начало и конец, могло бы решить проблему.

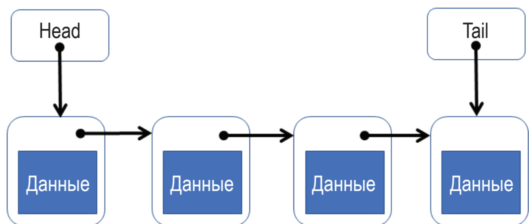
Потокобезопасную очередь с раздельной блокировкой начала и конца невозможно построить, основываясь на абстракциях, предоставляемых стандартным типом `std::queue`. Следовательно, структуру данных, лежащую в основе очереди, придётся реализовать самостоятельно. Прежде всего разберём, как можно реализовать очередь, а затем заделаем её потокобезопасной.

12.3.2.2. Простая реализация очереди

Самая очевидная структура данных, на основе которой удобно реализовать очередь, – это односвязный список. Односвязным называют список, в котором у каждого элемента есть указатель на следующий элемент, но не на предыдущий. Для управления списком нужны указатель `head` на первый элемент списка и указатель `tail` на последний[1] (см. рисунок). Такой список легко превратить в очередь, если изымать элементы операцией `pop` из начала списка, а добавлять операцией `push` в конец. Для удаления первого элемента

[1] Такое употребление термина tail не совсем традиционно: обычно словом tail называют часть списка за исключением первого элемента, а для обозначения последнего элемента используют слово last. – *Прим. перев.*

из списка достаточно продвинуть указатель head на один элемент вперёд. Эта операция также должна возвращать значение удаляемого элемента. Для добавления элемента в конец очереди нужно в бывшем последнем элементе проставить указатель на новый элемент и поставить указатель tail на этот новый элемент. Операции добавления и изъятия элемента должны также обрабатывать ситуацию, когда в очереди нет ни одного элемента. Код реализации представлен ниже.



Очередь на основе односвязного списка

Простая реализация очереди

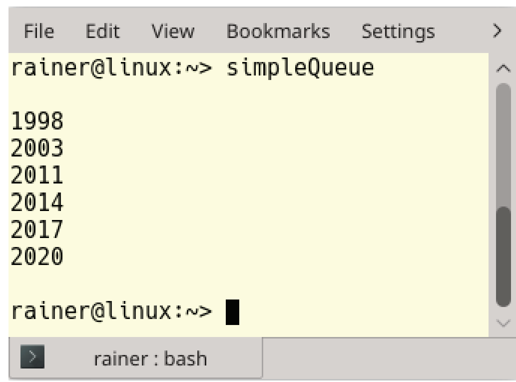
```
1  // simpleQueue.cpp
2
3  #include <iostream>
4  #include <memory>
5  #include <utility>
6
7  template <typename T>
8  class Queue {
9  private:
10     struct Node {
11         T data;
12         std::unique_ptr<Node> next;
13         Node(T data_): data(std::move(data_)){}
14     };
15
16     std::unique_ptr<Node> head;
17     Node* tail;
18
19 public:
20     Queue(): tail(nullptr) {};
21     std::unique_ptr<T> pop() {
22         if (!head) throw std::out_of_range("The queue is empty!");
23         std::unique_ptr<T> res =
24             std::make_unique<T>(std::move(head->data));
25         std::unique_ptr<Node> oldHead = std::move(head);
26         head = std::move(oldHead->next);
27         if (!head) tail = nullptr;
28         return res;
29     }
30
31     void push(T val) {
```

```
32         std::unique_ptr<Node> newNode =
33             std::make_unique<Node>(Node(std::move(val)));
34         Node* newTail = newNode.get();
35         if (tail) tail->next= std::move(newNode);
36         else head = std::move(newNode);
37         tail = newTail;
38     }
39 
40     Queue(const Queue& other) = delete;
41     Queue& operator=(const Queue& other) = delete;
42 };
43 
44 int main() {
45     std::cout << std::endl;
46 
47     Queue<int> myQueue;
48     myQueue.push(1998);
49     myQueue.push(2003);
50     std::cout << *myQueue.pop() << std::endl;
51     std::cout << *myQueue.pop() << std::endl;
52     myQueue.push(2011);
53     myQueue.push(2014);
54     std::cout << *myQueue.pop() << std::endl;
55     myQueue.push(2017);
56     myQueue.push(2020);
57     std::cout << *myQueue.pop() << std::endl;
58     std::cout << *myQueue.pop() << std::endl;
59     std::cout << *myQueue.pop() << std::endl;
60 
61     std::cout << std::endl;
62 }
```

Для автоматического управления временем жизни элементов списка используется умный указатель `std::unique_ptr`. Переменная-член `tail`, однако, имеет тип обычного указателя, так как узел, на который она указывает, уже имеет владельца. Функция-член `push` (строки 31–38) добавляет новый элемент в очередь. Для этого сперва создаётся новый объект типа `Node` (строка 32). Ему предстоит стать последним элементом списка (строка 34). Если в старом состоянии списка последний элемент существует (т. е. если список не пуст), то указатель на следующий элемент в этом бывшем последнем элементе ставится на новый элемент (строка 35). В противном случае список пуст – тогда новый элемент становится первым элементом списка (строка 36). Наконец, указатель `tail` устанавливается на только что добавленный элемент (строка 37).

Функция-член `pop` (строки 21–29) изымает из списка первый элемент и возвращает содержащиеся в нём данные (строка 28). Если список пуст, функция бросает исключение (строка 22). В строке 23 создаётся значение, которое функция должна будет вернуть, затем первый элемент списка перемещается в промежуточную переменную `oldHead` (строка 25). Адрес `oldHead->next` становится новым первым элементом списка (строка 26). Наконец, если список

содержал единственный элемент, его изъятие делает список пустым – в этом случае нужно обнулить указатель `tail` (строка 27). На следующем рисунке показан результат запуска программы.



Работа простой очереди на односвязном списке

Может возникнуть вопрос, почему очередь реализована именно таким образом – ведь она обладает тем же недостатком, о котором говорилось ранее: если очередь содержит ровно один элемент, её первый и последний элементы `head` и `tail` совпадают. В этом случае перемежающиеся вызовы операций `pop` и `push` из разных потоков могут привести к гонке данных. Например, обращение к члену-переменной `tail->next` в строке 35 может произойти одновременно с обращением к той же переменной через `oldHead->next` в строке 26. В конечном счёте это означает, что для корректной работы списка всё равно нужен единый мьютекс, блокирующий весь список при каждой операции. Ответ на этот вопрос таков. Действительно, эту реализацию очереди нельзя сделать потокобезопасной с помощью двух независимых мьютексов, но она послужит основой для такой раздельной блокировки. Для этого понадобится небольшая хитрость: нужно отделить первый элемент списка от последнего.

12.3.2.3. Очередь с фиктивным элементом

Хитрость состоит в том, чтобы держать в очереди лишний элемент. Благодаря ему конец очереди никогда не совпадает с её началом, поэтому одновременные обращения к указателю `head` и к члену `next` по указателю `tail` никогда не приводят к гонке данных. Конечно же, за это приходится платить усложнением реализации, поскольку фиктивный элемент нужно как-то обрабатывать.

Простая очередь с фиктивным элементом

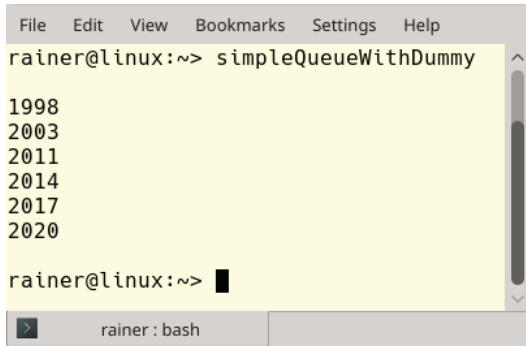
```
1  // simpleQueueWithDummy.cpp
2
3  #include <iostream>
4  #include <memory>
```

```
5  #include <utility>
6
7  template <typename T>
8  class Queue {
9  private:
10     struct Node {
11         T data;
12         std::unique_ptr<Node> next;
13         Node(T data_): data(std::move(data_)) {}
14     };
15
16     std::unique_ptr<Node> head;
17     Node* tail;
18
19 public:
20     Queue(): head(new Node(T{})), tail(head.get()) {};
21
22     std::unique_ptr<T> pop() {
23         if (head.get() == tail)
24             throw std::out_of_range("The queue is empty!");
25         std::unique_ptr<T> res =
26             std::make_unique<T>(std::move(head->data));
27         std::unique_ptr<Node> oldHead = std::move(head);
28         head = std::move(oldHead->next);
29         return res;
30     }
31
32     void push(T val) {
33         std::unique_ptr<Node> dummyNode =
34             std::make_unique<Node>(Node(T{}));
35         Node* newTail = dummyNode.get();
36         tail->next= std::move(dummyNode);
37         tail->data = val;
38         tail = newTail;
39     }
40
41     Queue(const Queue& other) = delete;
42     Queue& operator=(const Queue& other) = delete;
43 };
44
45 int main() {
46     std::cout << std::endl;
47
48     Queue<int> myQueue;
49     myQueue.push(1998);
50     myQueue.push(2003);
51     std::cout << *myQueue.pop() << std::endl;
52     std::cout << *myQueue.pop() << std::endl;
53     myQueue.push(2011);
54     myQueue.push(2014);
55     std::cout << *myQueue.pop() << std::endl;
56     myQueue.push(2017);
```

```
57      myQueue.push(2020);
58      std::cout << *myQueue.pop() << std::endl;
59      std::cout << *myQueue.pop() << std::endl;
60      std::cout << *myQueue.pop() << std::endl;
61
62      std::cout << std::endl;
63  }
```

Отличие этой реализации от предыдущей невелико. Во-первых, указатели `head` и `tail` изначально указывают на фиктивный элемент. Посмотрим внимательнее на функцию `pop`. В строке 23 делается проверка, является ли очередь логически пустой, т. е. содержит ли она элементы, помимо фиктивного. Функция `push` претерпевает более серьёзные изменения. Первым делом в строке 34 создаётся новый фиктивный элемент, на который затем будет установлен указатель `tail` (строки 36 и 39). Элемент, который ранее был фиктивным, теперь начинает указывать на новый фиктивный элемент как на следующий (строка 37) и получает себе новое значение `val` (строка 38).

Как и следовало ожидать, запуск этой программы приводит к такому же результату, что и запуск предыдущей реализации, не содержавшей фиктивного элемента.



Работа очереди с фиктивным элементом

На этом преобразование кода закончено. Важно иметь в виду, что в новой реализации функции `push` и `pop` работают почти исключительно над разными концами очереди. Это позволит нам в будущем использовать в них два разных мьютекса. Лишь одна операция – проверка на пустоту очереди в строке 23 – требует обоих мьютексов. Это не так плохо, ведь эта критическая секция занимает мало времени.

Теперь, имея на руках все элементы головоломки, соберём их вместе и построим потокобезопасную очередь с раздельной блокировкой начала и конца.

12.3.2.4. Окончательная реализация

Синхронизация этой очереди основана на двух мьютексах. Один мьютекс защищает доступ к первому элементу очереди, другой – к последнему. Ещё

один важный вопрос: в каких местах лучше поставить их блокировку? Чтобы добиться наилучшей производительности, критические секции нужно сделать как можно короче. Функцию pop придётся защитить блокировкой целиком, однако в функции push защищать нужно только ту часть, где используется переменная tail. Остальные операции выполняются над локальными переменными и потому не нуждаются в синхронизации.

Потокобезопасная очередь с раздельной блокировкой операций

```
1  // concurrentQueueFineLocking.cpp
2
3  #include <future>
4  #include <iostream>
5  #include <memory>
6  #include <mutex>
7  #include <utility>
8
9  template <typename T>
10 class ConcurrentQueue {
11 private:
12     struct Node {
13         T data;
14         std::unique_ptr<Node> next;
15         Node(T data_): data(std::move(data_)) {}
16     };
17
18     std::unique_ptr<Node> head;
19     Node* tail;
20     std::mutex headMutex;
21     std::mutex tailMutex;
22
23 public:
24     ConcurrentQueue (): head(new Node(T{})), tail(head.get()) {};
25
26     std::unique_ptr<T> pop() {
27         std::lock_guard<std::mutex> headLock(headMutex);
28         {
29             std::lock_guard<std::mutex> tailLock(tailMutex);
30             if (head.get() == tail)
31                 throw std::out_of_range("The queue is empty!");
32         }
33
34         std::unique_ptr<T> res =
35             std::make_unique<T>(std::move(head->data));
36         std::unique_ptr<Node> oldHead = std::move(head);
37         head = std::move(oldHead->next);
38         return res;
39     }
40
41     void push(T val) {
42         std::unique_ptr<Node> dummyNode =
43             std::make_unique<Node>(Node(T{}));
```

```
44          Node* newTail = dummyNode.get();
45          std::lock_guard<std::mutex> tailLock(tailMutex);
46          tail->next= std::move(dummyNode);
47          tail->data = val;
48          tail = newTail;
49      }
50
51      Queue(const Queue& other) = delete;
52      Queue& operator=(const Queue& other) = delete;
53  };
54
55  int main() {
56      std::cout << std::endl;
57
58      ConcurrentQueue<int> conQueue;
59
60      auto fut = std::async([&conQueue]{ conQueue.push(2011); });
61      auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
62      auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });
63
64      auto fut3 = std::async([&conQueue]{ return *conQueue.pop(); });
65      auto fut4 = std::async([&conQueue]{ return *conQueue.pop(); });
66      auto fut5 = std::async([&conQueue]{ return *conQueue.pop(); });
67
68      fut.get(), fut1.get(), fut2.get();
69
70      std::cout << fut3.get() << std::endl;
71      std::cout << fut4.get() << std::endl;
72      std::cout << fut5.get() << std::endl;
73
74      std::cout << std::endl;
75  }
```

В первую очередь зададимся вопросом, потокобезопасна ли эта реализация. Класс `ConcurrentQueue` обладает только двумя функциями-членами. Два мьютекса защищают от одновременного доступа односвязный список, состоящий из объектов типа `Node`. Мьютекс `headMutex` отвечает за доступ к первому элементу списка, а мьютекс `tailMutex` – за доступ к последнему элементу. Единственная операция, которая работает с указателями `head` и `tail` одновременно, защищена обоими мьютексами. Следовательно, данный тип свободен от гонки данных.

Далее, следует избегать захвата более чем одного мьютекса, потому что это может привести к мёртвой блокировке, если мьютексы захватываются не всегда в одном порядке. Хотя функция `pop` захватывает сначала мьютекс `headMutex`, а затем мьютекс `tailMutex`, возможности для мёртвой блокировки здесь нет, потому что вторая функция-член захватывает только один мьютекс. Таким образом, представленная здесь структура данных полностью потокобезопасна. Программа работает, как и ожидалось, результат её запуска показан на рисунке.

```
File  Edit  View  Bookmarks  Settings  Help
rainer@linux:~> concurrentQueueFineLocking

2014
2017
2011

rainer@linux:~> concurrentQueueFineLocking

2014
2017
2011

rainer@linux:~> concurrentQueueFineLocking

2014
2011
2017

rainer@linux:~> concurrentQueueFineLocking

terminate called after throwing an instance of 'std::out_of_range'
  what():  The queue is empty!
Abgebrochen (Speicherabzug geschrieben)
rainer@linux:~>
rainer : bash
```

Работа потокобезопасной очереди

12.3.2.5. Ожидание значения из очереди

Воспользовавшись переменной условия, можно сделать так, чтобы в случае пустой очереди функция pop ждала появления в ней элемента.

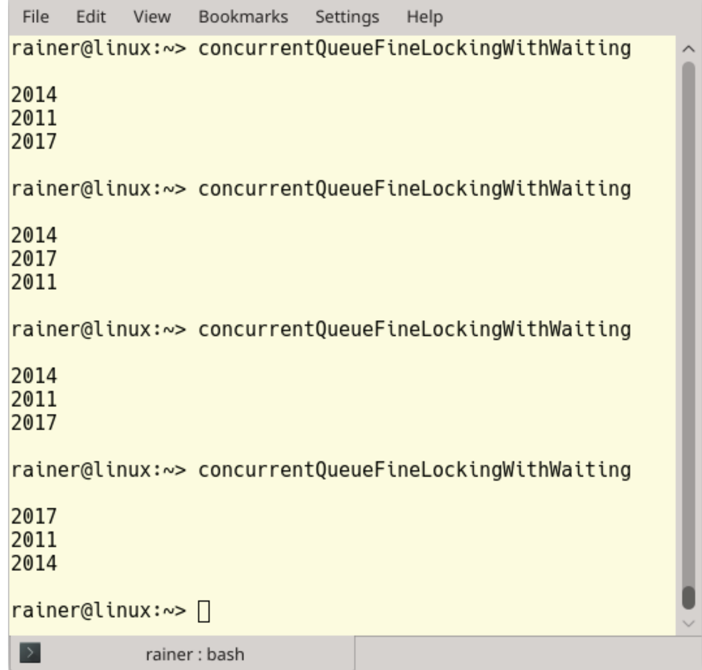
Потокобезопасная очередь с ожиданием

```
1  // concurrentQueueFineLockingWithWaiting.cpp
2
3  #include <condition_variable>
4  #include <future>
5  #include <iostream>
6  #include <memory>
7  #include <mutex>
8  #include <utility>
9
10 template <typename T>
11 class Queue {
12 private:
13     struct Node {
14         T data;
15         std::unique_ptr<Node> next;
16         Node(T data_): data(std::move(data_)) {}
17     };
```

```
18
19     std::unique_ptr<Node> head;
20     Node* tail;
21     std::mutex headMutex;
22     std::mutex tailMutex;
23     std::condition_variable condVar;
24
25 public:
26     Queue(): head(new Node(T{})), tail(head.get()) {};
27
28     std::unique_ptr<T> pop() {
29         std::lock_guard<std::mutex> headLock(headMutex);
30         {
31             std::unique_lock<std::mutex> tailLock(tailMutex);
32             if (head.get() == tail)
33                 condVar.wait(tailLock);
34         }
35
36         std::unique_ptr<T> res =
37             std::make_unique<T>(std::move(head->data));
38         std::unique_ptr<Node> oldHead = std::move(head);
39         head = std::move(oldHead->next);
40         return res;
41     }
42
43     void push(T val) {
44         std::unique_ptr<Node> dummyNode =
45             std::make_unique<Node>(Node(T{}));
46         Node* newTail = dummyNode.get();
47         {
48             std::unique_lock<std::mutex> tailLock(tailMutex);
49             tail->next= std::move(dummyNode);
50             tail->data = val;
51             tail = newTail;
52         }
53
54         condVar.notify_one();
55     }
56
57     Queue(const Queue& other) = delete;
58     Queue& operator=(const Queue& other) = delete;
59 };
60
61 int main() {
62     std::cout << std::endl;
63
64     Queue<int> conQueue;
65
66     auto fut = std::async([&conQueue]{ conQueue.push(2011); });
67     auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
68     auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });
69
70     auto fut3 = std::async([&conQueue]{ return *conQueue.pop(); });
```

```
71      auto fut4 = std::async([&conQueue]{ return *conQueue.pop(); });
72      auto fut5 = std::async([&conQueue]{ return *conQueue.pop(); });
73
74      fut.get(), fut1.get(), fut2.get();
75
76      std::cout << fut3.get() << std::endl;
77      std::cout << fut4.get() << std::endl;
78      std::cout << fut5.get() << std::endl;
79
80      std::cout << std::endl;
81  }
```

Изменения по сравнению с предыдущей программой минимальны. Функция `pop` в строке 33 ждёт оповещения о том, что в очереди появился новый элемент. Переменной условия нужен блокировщик типа `std::unique_lock` вместо использованного в предыдущей версии типа `std::lock_guard`. Функция `push` оповещает ровно один ожидающий поток о том, что в очереди появился новый элемент (строка 54). Напомним, что операция `notify_one` над переменной условия сама по себе синхронизации не требует. Возможно, читателя насторожило отсутствие предиката при ожидании в строке 33. Предикат при ожидании переменной условия нужен для защиты от утерянных и ложных пробуждений. Однако здесь с этим справляется предшествующая проверка на логическую пустоту. Остаётся завершить рассказ о потокобезопасной очереди примером запуска этой программы.



Работа потокобезопасной очереди с ожиданием

12.4. Краткие итоги

- Реализация потокобезопасных структур данных, даже на основе блокировок, – трудное дело, которым должны заниматься профессионалы.
- Типичные примеры структур данных, которые есть смысл делать потокобезопасными, – это стек и очередь.
- Операции над структурой данных могут блокировать её полностью или более мелкими частями.

Дополнительные сведения

13. Сложности параллельного программирования

Параллельному программированию внутренне присуща повышенная сложность. Это справедливо в том числе и при использовании средств, появившихся в стандартах C++ 11 и C++ 14, даже если не брать во внимание модель памяти. Поэтому стоит целую главу посвятить анализу специфических трудностей параллельного программирования, чтобы читатель был заранее готов к ним.

13.1. Проблема ABA

Обозначение «ABA» означает, что программа читает из некоторой переменной дважды, и оба раза получает одно и то же значение A. Поэтому алгоритм решает, что между этими моментами времени переменная своего значения не меняла. Однако на самом деле алгоритм пропускает значение B, побывавшее в переменной между двумя операциями чтения. Для более детального знакомства с данной проблемой опишем сначала простой сценарий.

13.1.1. Наглядное объяснение

Представим себе водителя, который сидит за рулём своего автомобиля, ожидая зелёного сигнала светофора. Зелёный сигнал играет роль значения B, а красный – значения A. Что может произойти в описанной ситуации?

1. Водитель смотрит на светофор и видит красный сигнал A.
2. Заскучав, водитель принимается читать новости на телефоне, забыв о времени.
3. Подняв взгляд на светофор, водитель обнаруживает по-прежнему красный свет – значение A.

Конечно же, сигнал светофора успел побыть зелёным (B) между двумя проверками. То, что показалось водителю одним непрерывным красным сигналом, было на самом деле циклом смены сигналов. Что означает проблема ABA для потоков или процессов? Опишем это более строго.

1. Поток 1 читает из переменной `var` значение A.
2. Выполнение потока 1 приостанавливается, управление получает поток 2.
3. Поток 2 меняет значение переменной `var` сначала на B, затем снова устанавливает ей значение A.
4. Поток 1 продолжает своё выполнение и снова читает из переменной `var` значение A, установленное последним.

Часто это явление вообще не представляет собой проблемы, и на него можно просто не обращать внимания.

13.1.2. Некритические случаи эффекта ABA

Использование функции `compare_exchange_strong` (а также `compare_exchange_weak`) в следующей программе подвержено эффекту ABA, однако в данном контексте он не составляет проблемы. Функция `fetch_mult` умножает значение атомарной переменной `shared` на значение `mult`.

Атомарное умножение

```
1  // fetch_mult.cpp
2
3  #include <atomic>
4  #include <iostream>
5
6  template <typename T>
7  T fetch_mult(std::atomic<T>& shared, T mult) {
8    T oldValue = shared.load();
9    while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10   return oldValue;
11 }
12
13 int main() {
14   std::atomic<int> myInt{5};
15   std::cout << myInt << std::endl;
16   fetch_mult(myInt,5);
17   std::cout << myInt << std::endl;
18 }
```

Суть состоит в том, что между первоначальным чтением старого значения переменной в строке 8 и его использованием в строке 9 есть небольшой зазор. Следовательно, другой поток может вклиниться между этими операциями, изменить значение переменной на какое-то иное, а затем вернуть старое значение. С точки зрения эффекта ABA старое значение переменной соответствует значению A, а временное промежуточное – B.

Часто появление у переменной кратковременного нового значения между операциями чтения не играет никакой роли, если операции чтения возвращают одно и то же значение. Однако в случае неблокирующих потокобезопасных структур данных эффект ABA может представлять серьёзную проблему.

13.1.3. Неблокирующая структура данных

Не будем подробно описывать здесь устройство неблокирующей структуры данных. Возьмём неблокирующий стек, реализованный на основе односвязного списка. Стек поддерживает две операции:

- изъять из стека верхний элемент и вернуть указатель на него (`pop`);
- положить новый объект на вершину стека (`push`).

Чтобы понять, как на этой структуре данных проявляется эффект ABA, опишем сначала работу операции `pop`. Эта операция должна состоять из следующих шагов:

- получить указатель на верхний элемент стека `head`;
- получить указатель `headNext` на следующий элемент;
- если указатель head по-прежнему указывает на верхний элемент стека, изменить его значение на `headNext`.

Покажем, как может проявиться эффект ABA.

13.1.4. Эффект ABA в действии

Пусть стек имеет следующий вид:

```
Stack: TOP -> A -> B -> C
```

Допустим, поток 1 хочет взять из стека первый элемент. Для этого он сохраняет у себя указатели:

- `head = A`,
- `headNext = B`.

Прежде чем поток 1 успеет завершить операцию `pop`, вклинивается поток 2. Он изымает из стека элемент A, в результате чего стек принимает вид

```
Stack: TOP -> B -> C
```

Затем поток 2 изымает элемент B и удаляет его из памяти:

```
Stack: TOP -> C
```

Теперь поток 2 возвращает элемент A на вершину стека:

```
Stack: TOP -> A -> C
```

Наконец, возобновляется выполнение потока 1. Он проверяет равенство `A == head`. Поскольку это равенство выполняется, новой вершиной стека ста-

новится элемент `headNext`, то есть элемент B. Однако элемент B уже удалён, поэтому программа в целом обладает неопределённым поведением.

Существует несколько способов справиться с проблемами эффекта ABA.

13.1.5. Исправление эффекта ABA

Причину описанной в предыдущем разделе проблемы понять просто. Элемент стека, в данном случае элемент B, уже удалён, тогда как другой элемент, а именно элемент A, по-прежнему содержит указатель на него. Очевидное решение состоит в том, чтобы предотвратить преждевременное удаление объекта. Для этого есть несколько методов, они будут разобраны в следующих подразделах.

13.1.5.1. Ссылка на помеченное состояние

Неиспользуемые биты адреса можно использовать в качестве счётчика модификаций элемента списка. Это приведёт к тому, что операция сравнения и обмена потерпит неудачу, даже если указатели указывают на один и тот же объект. Впрочем, это решение не всегда работоспособно, поскольку неиспользуемых битов может оказаться недостаточно, и со временем произойдёт их переполнение. В архитектурах, где атомарные операции сравнения и обмена поддерживаются для чисел большой разрядности, можно зарезервировать для счётчика достаточно много битов, чтобы не опасаться переполнения.

Ссылки на помеченные состояния обычно используются для реализации транзакционной памяти. Следующие три подхода основаны на идее отложенного удаления объекта.

13.1.5.2. Сборка мусора

Сборка мусора – это механизм, который гарантирует, что объект в динамической памяти уничтожается только тогда, когда он становится не нужен программе. Звучит многообещающе, но реализация связана с существенными трудностями. Большая часть сборщиков мусора не являются потокобезопасными, поэтому даже при использовании потокобезопасной структуры данных система в целом таковой не является.

13.1.5.3. Списки опасных указателей

Как написано в английской Википедии[1], каждый поток добавляет свой список в единую систему управления опасными указателями; это адреса объектов, к которым поток в настоящее время обращается. Во многих системах этот «список» оказывается ограничен одним или двумя элементами. Ни один поток не должен модифицировать или удалять объекты из списка опасных

[1] https://en.wikipedia.org/wiki/Hazard_pointer.

указателей. Если поток желает удалить элемент структуры данных, он помещает его в список объектов, предназначенных для последующего удаления, однако не освобождает занимаемую им память немедленно до тех пор, пока хотя бы один из других потоков содержит адрес объекта в своём списке опасных указателей. Удалением таких объектов может заниматься выделенный поток сборки мусора, если список объектов, предназначенных к удалению, находится в общем доступе потоков. Иное решение состоит в том, чтобы очистка списка удаляемых объектов выполнялась каждым потоком отдельно, как часть операций наподобие `pop`.

13.1.5.4. Механизм чтения-копирования-модификации

Этот способ синхронизации, известный также под названием RCU (англ. *Read*, *Copy*, *Update*), хорошо подходит для структур данных, которые редко подвергаются изменениям и используются в основном для чтения. Этот механизм предложен Полом МакКенни и используется в ядре Linux с 2002 года.

Идея данного подхода проста и вполне ясна из названия. Когда возникает необходимость внести изменение в данные, с них снимается копия, и изменения вносятся в неё. Тем временем все потоки-читатели продолжают работать с исходными данными. Когда читатели прекращают свою работу, можно быстро подменить данные модифицированной копией. Чтобы изучить метод RCU в подробностях, стоит обратиться к статье «Что такое RCU в самом деле?» автора подхода, Пола МакКенни.

Два предложения к стандарту

Среди обширного пакета предложений к будущим версиям стандарта C++ в части средств параллельного программирования есть предложение P0233r0[1] об опасных указателях и предложение P0461R0[2] о поддержке механизма RCU.

13.2. Тонкости блокировок

Основную мысль можно выразить предельно ясно: переменные условия всегда следует использовать вместе с предикатом. В противном случае программа может стать жертвой ложного или утерянного пробуждения.

Если переменную условия использовать без предиката, поток-отправитель может послать оповещение перед тем, как поток-получатель начнёт его ожидать. В этом случае поток обречён ожидать далее до бесконечности. Это явление называют утерянным пробуждением. Ниже приведён код, иллюстрирующий эту проблему.

---

[1] http://www.modernescpp.com/open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0233r0.pdf.

[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0461r0.pdf.

Блокировка на переменной условия

```
1  // conditionVariableBlock.cpp
2
3  #include <iostream>
4  #include <condition_variable>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex mutex_;
9  std::condition_variable condVar;
10
11 bool dataReady;
12
13
14 void waitingForWork() {
15     std::cout << "Worker: Waiting for work." << std::endl;
16
17     std::unique_lock<std::mutex> lck(mutex_);
18     condVar.wait(lck);
19     // do the work
20     std::cout << "Work done." << std::endl;
21 }
22
23 void setDataReady() {
24     std::cout << "Sender: Data is ready."  << std::endl;
25     condVar.notify_one();
26 }
27
28 int main() {
29   std::cout << std::endl;
30
31   std::thread t1(setDataReady);
32   std::thread t2(waitingForWork);
33
34   t1.join();
35   t2.join();
36
37   std::cout << std::endl;
38 }
```

По счастливому стечению обстоятельств, первый запуск этой программы отработал нормально. Однако при повторном запуске программа зависла, поскольку оповещение было послано одним потоком раньше, чем другой стал готов его принять.

```
File   Edit   View   Bookmarks   Settings   Help
rainer@suse:~> conditionVariableBlock

Worker: Waiting for work.
Sender: Data is ready.
Work done.

rainer@suse:~> conditionVariableBlock

Sender: Data is ready.
Worker: Waiting for work.

rainer : conditionVariab
```

Блокировка на переменной условия

Конечно, состояние гонок часто приводит как к мёртвым блокировкам, так и к динамическим тупикам. Попадание в мёртвую блокировку обычно зависит от порядка, в котором перемежаются операции, выполняемые разными потоками, это событие может произойти или не произойти в зависимости от случайного стечения обстоятельств. Динамический тупик отчасти похож на мёртвую блокировку. Однако если мертвая блокировка полностью останавливает работу некоторых потоков, то в состоянии динамического тупика кажется, что потоки продолжают выполнять работу – но именно «кажется». В качестве примера можно привести транзакционную память, которую пытаются модифицировать два потока. Всякий раз, когда один из них пытается подтвердить свою транзакцию, он обнаруживает конфликт и оказывается вынужден повторить транзакцию с самого начала. Тем самым потоки тратят больше времени на бесплодные попытки синхронизироваться, чем на полезную работу. Напомним, что о транзакционной памяти речь шла в разделе 7.3.

13.3. Нарушение инварианта программы

Инвариант программы – это соотношение, которое должно оставаться истинным всё время, пока программа выполняется. В состоянии гонок инвариант программы может нарушаться. Рассмотрим пример. В приведённой ниже программе инвариант состоит в том, что общая сумма по всем счетам должна оставаться постоянной величиной. В данном примере это 200 единиц, поскольку каждый из двух счетов в момент создания кредитуется на 100 единиц. Программа должна работать так, чтобы во время пересылки денег с одного счёта на другой деньги не возникали из ниоткуда даже на короткое время и не исчезали в никуда.

Нарушение инварианта программы

```
1  // breakingInvariant.cpp
2
3  #include <atomic>
4  #include <functional>
5  #include <iostream>
6  #include <thread>
7
8  struct Account{
9    std::atomic<int> balance{100};
10 };
11
12 void transferMoney(int amount, Account& from, Account& to) {
13   using namespace std::chrono_literals;
14   if (from.balance >= amount) {
15     from.balance -= amount;
16     std::this_thread::sleep_for(1ns);
17     to.balance += amount;
18   }
19 }
20
21  void printSum(Account& a1, Account& a2) {
22   std::cout << (a1.balance + a2.balance) << std::endl;
23 }
24
25 int main() {
26   std::cout << std::endl;
27
28   Account acc1;
29   Account acc2;
30
31   std::cout << “Initial sum: “;
32   printSum(acc1, acc2);
33
34   std::thread thr1(transferMoney, 5, std::ref(acc1), std::ref(acc2));
35   std::thread thr2(transferMoney, 13, std::ref(acc2), std::ref(acc1));
36   std::cout << “Intermediate sum: “;
37   std::thread thr3(printSum, std::ref(acc1), std::ref(acc2));
38
39   thr1.join();
40   thr2.join();
41   thr3.join();
42
43   std::cout << “     acc1.balance: “ << acc1.balance << std::endl;
44   std::cout << “     acc2.balance: “ << acc2.balance << std::endl;
45
46   std::cout << “Final sum: “;
47   printSum(acc1, acc2);
48
49   std::cout << std::endl;
50 }
```

В начале программы общая сумма по счетам составляет 200 единиц. Эта величина выводится на печать в строке 32. Затем запускаются три потока, два из которых переводят деньги между счетами, а третий в это же время печатает общую сумму, делая нарушение инварианта очевидным. Из-за ничтожной задержки в одну наносекунду в строке 16 поток `thr3` успевает заметить промежуточное состояние, в котором сумма по счетам составляет лишь 182 единицы. Однако к концу программы ситуация снова приходит в норму: оба счёта имеют корректные значения, а общая сумма по-прежнему составляет 200 единиц. Результат запуска программы показан на рисунке.

```
Initial sum: 200
Intermediate sum: 182
    acc1.balance: 108
    acc2.balance: 92
Final sum: 200
```

Нарушение инварианта программы

13.4. Гонка данных

Гонкой данных называют ситуацию, когда по крайней мере два потока одновременно обращаются к общей переменной, причём по меньшей мере один из потоков пытается изменить её значение.

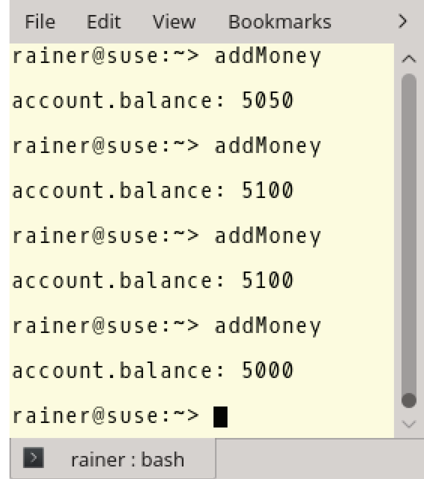
Если в программе имеет место гонка данных, её поведение считается неопределённым. Это означает, что запуск программы может привести к абсолютно любому результату, и всякие дальнейшие рассуждения о её поведении теряют смысл. Рассмотрим пример программы с гонкой данных.

Гонка данных

```
1  // addMoney.cpp
2
3  #include <functional>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  struct Account{
9    int balance{100};
10 };
11
12 void addMoney(Account& to, int amount) {
13   to.balance += amount;
14 }
15
16 int main() {
17   std::cout << std::endl;
```

```
18
19    Account account;
20
21    std::vector<std::thread> vecThreads(100);
22
23    for (auto& thr: vecThreads)
24        thr = std::thread(addMoney, std::ref(account), 50);
25
26    for (auto& thr: vecThreads) thr.join();
27
28    std::cout << "account.balance: " << account.balance << std::endl;
29
30    std::cout << std::endl;
31 }
```

Сто потоков пытаются перечислить по 50 единиц на один и тот же счёт. Для этого потоки используют функцию `addMoney`. Решающее значение имеет тот факт, что запись значения в переменную происходит без какой-либо синхронизации. Это означает, что в программе имеет место гонка данных, и результату работы такой программы доверять нельзя. Программа обладает неопределённым поведением, в частности окончательная сумма на счету при разных запусках программы оказывается различной, от 5000 до 5100 единиц.



Гонка данных

13.5. Мёртвые блокировки

Мёртвой блокировкой называется ситуация, когда по крайней мере один поток заблокирован навсегда в ожидании ресурса, который гарантированно никогда не будет освобождён. У мёртвых блокировок две основные причины:

- нужный потоку мьютекс захватывается, но не освобождается другим потоком;
- потоки захватывают одни и те же мьютексы в различном порядке.

Для преодоления второй трудности в классическом языке C++ использовались техники наподобие иерархических блокировщиков[1].

Более подробные сведения о мёртвых блокировках и способах борьбы с ними содержатся в подразделе 3.3.2.

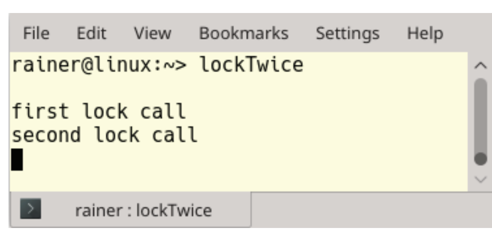
Повторный захват нерекурсивного мьютекса

Если один и тот же поток пытается дважды захватить мьютекс (помимо рекурсивного, специально созданного для таких сценариев использования), результатом становится неопределённое поведение.

Повторный захват мьютекса

```
1  // lockTwice.cpp
2
3  #include <iostream>
4  #include <mutex>
5
6  int main(){
7    std::mutex mut;
8
9    std::cout << std::endl;
10
11    std::cout << “first lock call” << std::endl;
12
13    mut.lock();
14
15    std::cout << “second lock call” << std::endl;
16
17    mut.lock();
18
19    std::cout << “third lock call” << std::endl;
20 }
```

На практике обычно повторный захват мьютекса ведёт к мёртвой блокировке, как видно по следующему рисунку.



Мёртвая блокировка
при повторном захвате нерекурсивного мьютекса

[1] http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2008/0801/071201hs01/071201hs01.html.

13.6. Неявные связи между данными

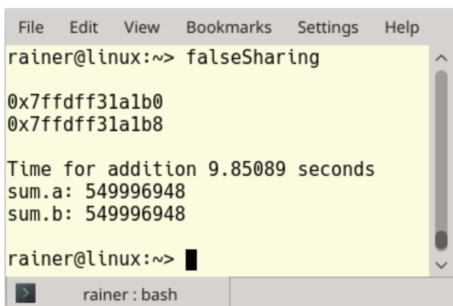
Когда процессор запрашивает из оперативной памяти переменную, например типа `int`, из памяти читается больший объём данных, чем одно целое число. В буфер сверхоперативной памяти (кеш) читается сразу целая строка, обычно составляющая 64 байта. Если два потока в одно и то же время работают с двумя разными переменными, находящимися в одной строке кеша, между этими переменными возникает непредусмотренная логикой программы связь. При каждом обращении к любой из этих переменных происходит сравнительно дорогостоящая операция аппаратной синхронизации кеша. В результате этого вычисления дают правильный результат, но производительность программы заметно страдает. Именно это происходит в следующей программе.

Неявные связи между данными

```
1  // falseSharing.cpp
2
3  #include <algorithm>
4  #include <chrono>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <vector>
9
10 constexpr long long size{100000000};
11
12 struct Sum {
13     long long a{0};
14     long long b{0};
15 };
16
17 int main() {
18     std::cout << std::endl;
19
20     Sum sum;
21
22     std::cout << &sum.a << std::endl;
23     std::cout << &sum.b << std::endl;
24
25     std::cout << std::endl;
26
27     std::vector<int> randValues, randValues2;
28     randValues.reserve(size);
29     randValues2.reserve(size);
30
31     std::mt19937 engine;
32     std::uniform_int_distribution<> uniformDist(1,10);
33
34     int randValue;
```

```
35     for (long long i = 0; i < size; ++i) {
36         randValue = uniformDist(engine);
37         randValues.push_back(randValue);
38         randValues2.push_back(randValue);
39     }
40 
41     auto sta = std::chrono::steady_clock::now();
42 
43     std::thread t1([&sum, &randValues]
44         { for (auto val: randValues) sum.a += val; });
45     std::thread t2([&sum, &randValues2]
46         { for (auto val: randValues2) sum.b += val; });
47 
48     t1.join(), t2.join();
49 
50     std::chrono::duration<double> dur =
51         std::chrono::steady_clock::now() - sta;
52     std::cout << "Time for addition " << dur.count()
53             << " seconds" << std::endl;
54 
55     std::cout << "sum.a: " << sum.a << std::endl;
56     std::cout << "sum.b: " << sum.b << std::endl;
57 
58     std::cout << std::endl;
59 }
```

Переменные `a` и `b`, объявленные в строках 13 и 14, располагаются в памяти рядом и опадают в одну строку кеша, поскольку имеют тип `long long`, для которого по умолчанию используется выравнивание по 8 байт. Поток `t1` использует переменную `a` для суммирования случайных чисел из контейнера `randValues`, а поток `t2` делает то же самое с переменной `b` и контейнером `randValues2`. Контейнеры содержат по 100 миллионов чисел из диапазона от 1 до 10. Запуск программы даёт интересный результат.

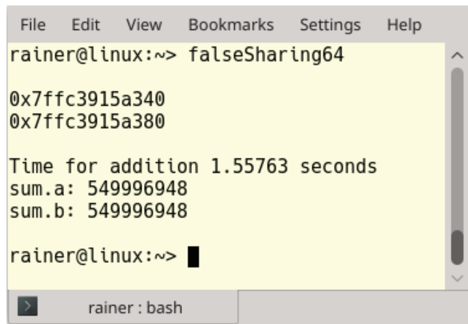


Неявные связи между данными

Теперь потребуем для переменных `a` и `b` выравнивания по 64-байтной границе (размер строки кеша на данном компьютере) и посмотрим, что получится. Для этого нужно внести в объявление типа `Sum` небольшое изменение.

```
struct Sum {
    alignas(64) long long a{0};
    alignas(64) long long b{0};
};
```

Поскольку генератор случайных чисел никак не инициализируется, программа вычислит тот же результат.

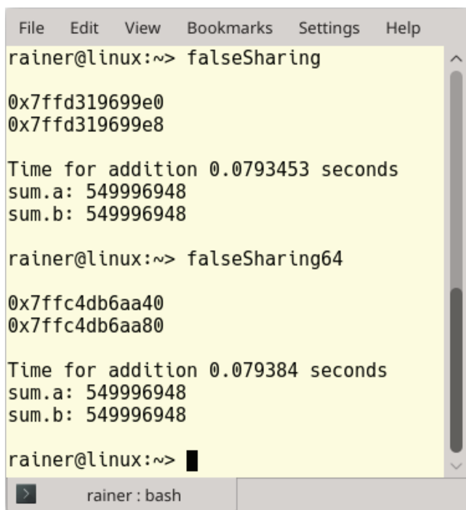


Устранение неявной связи данных

Теперь программа работает в шесть раз быстрее. Причина ускорения в том, что сейчас переменные `a` и `b` попадают в различные строки кеша.

Устранение неявной связи компилятором

Если компилировать эти программы с максимальным уровнем оптимизации, оптимизатор обнаруживает неявную связь между переменными и сам устраняет её. Это означает, что обе версии программы, с выравниванием переменных по умолчанию и с явным выравниванием на 65 байт, покажут одинаковую производительность. Результат запуска оптимизированных программ показан на рисунке.



Устранение неявной связи компилятором

Получение параметров аппаратного кеша

В стандарте C++ 17 появились константы `std::hardware_destructive_interference_size` и `std::hardware_constructive_interference_size`. Они позволяют переносимым образом работать с размером строки кеша. Первая из них представляет собой наименьшее смещение в байтах между двумя объектами, при котором гарантируется отсутствие неявной связи между ними. Вторая константа, напротив, представляет собой наибольшее смещение между объектами, при котором гарантируется их попадание в одну строку кеша – там, где это желательно из соображений эффективности. С использованием этих средств тип `Sum` из предыдущего примера можно переписать следующим образом:

```
struct Sum {
  alignas(std::hardware_destructive_interference_size) long long a{0};
  alignas(std::hardware_destructive_interference_size) long long b{0};
};
```

13.7. Проблемы со временем жизни объектов

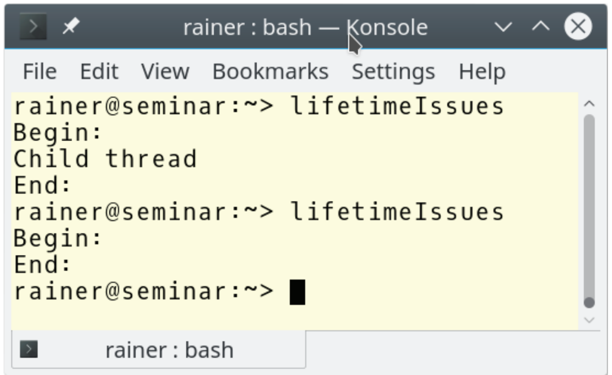
Создать на языке C++ пример с ошибкой, связанной со временем жизни объекта, довольно просто. Пусть поток `t` продолжает своё выполнение независимо от создавшего его потока, т. е. отсоединяется от потока-создателя с помощью функции `detach`, и пусть он сделал половину своей работы. Тем временем поток-создатель завершается, не дожидаясь окончания потока `t`. В этом случае нужно быть чрезвычайно осторожным и никак не обращаться из потока `t` к объектам, которыми владеет поток-создатель.

Проблемы со временем жизни объектов

```
1  // lifetimeIssues.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <thread>
6
7  int main() {
8    std::cout << "Begin:" << std::endl;
9
10   std::string mess{"Child thread"};
11
12   std::thread t([&mess]{ std::cout << mess << std::endl;});
13   t.detach();
14
15   std::cout << "End:" << std::endl;
16 }
```

Этот пример слишком прост, чтобы его подробно комментировать. В потоке `t` используется объект `std::cout` и переменная `mess`. Последний представляет собой локальную переменную функции `main` и может быть уничтожен

раньше, чем поток t успеет вывести его значение. Как видно из следующего рисунка, при втором запуске программы из потока t ничего не выводится, печатаются лишь сообщения «Begin» и «End» из главного потока.



Проблемы со временем жизни объектов

13.8. Перемещение потоков

Перемещение объектов-потоков делает проблемы, связанные со временем жизни объектов, ещё более острыми. Тип std::thread поддерживает семантику перемещения, но не семантику копирования. В самом деле, как должно было бы вести себя копирование потока в то время, когда поток заблокирован в ожидании мьютекса? Поэтому конструктор копирования в этом классе в явном виде удалён:

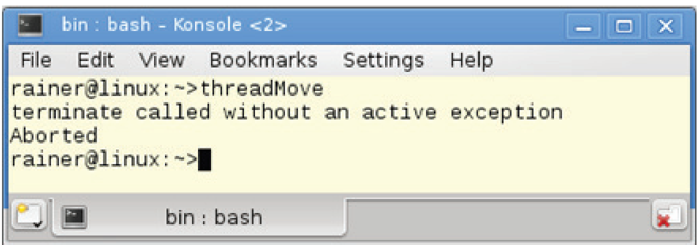
```
thread(const thread&) = delete;
```

Рассмотрим теперь перемещение объекта-потока.

Ошибочное перемещение потока

```
1  // threadMoved.cpp
2
3  #include <iostream>
4  #include <thread>
5  #include <utility>
6
7  int main() {
8    std::thread t([]{std::cout << std::this_thread::get_id();});
9    std::thread t2(([]{std::cout << std::this_thread::get_id();});
10
11   t = std::move(t2);
12   t.join();
13   t2.join();
14 }
```

Оба потока должны сделать очень простую вещь – напечатать свои идентификаторы. Кроме того, поток t2 перемещается в объект t. В конце главная функция берёт на себя заботу о созданных ею потоках и присоединяет их, чтобы дождаться их завершения. Однако поведение программы оказывается далёким от ожиданий, как показано на рисунке.



Ошибочное перемещение потока

Что могло пойти не так? В этой программе две проблемы:

1) в результате операции перемещения объект t должен получить под своё управление новый выполняемый объект. Для этого старый выполняемый объект должен быть уничтожен. Поскольку поток находится в присоединяемом состоянии, это приводит к вызову функции `std::terminate`, подобно вызову деструктора для потока в присоединяемом состоянии;
2) после перемещения объект t2 более не содержит в себе выполняемого объекта. Поэтому попытка применить функцию-член `join` вызывает исключение `std::system_error`.

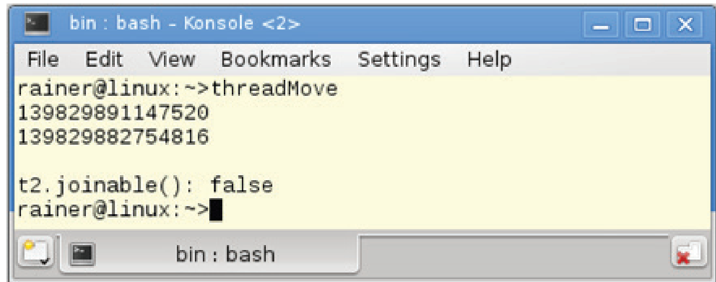
Имея в виду эти два факта, можно легко исправить обе ошибки, как показано ниже.

Перемещение потока без ошибок

```
1  // threadMovedFixed.cpp
2
3  #include <iostream>
4  #include <thread>
5  #include <utility>
6
7  int main() {
8    std::thread t([]{std::cout << std::this_thread::get_id() << '\n';});
9    std::thread t2([]{std::cout << std::this_thread::get_id() << '\n';});
10
11   t.join();
12   t = std::move(t2);
13   t.join();
14
15   std::cout << "\n";
16   std::cout
17     << std::boolalpha
```

```
18      << "t2.joinable(): "
19      << t2.joinable()
20      << '\n';
21  }
```

Результат работы этой программы свидетельствует о том, что поток t2 к концу работы главной функции уже не находится в присоединяемом состоянии.



Перемещение потока без ошибок

13.9. Состояние гонки

Состояние гонки – это ситуация, при которой результат некоторой операции зависит от относительного порядка выполнения операций в разных потоках.

Состояние гонок бывает трудно обнаружить в программе. Видимые признаки этого состояния зависят от множества трудно поддающихся учёту факторов: количество ядер процессора, нагрузка на систему, уровень оптимизации, задержки ввода-вывода могут повлиять на работу программы, в которой имеет место гонка.

В самом по себе состоянии гонки может не быть ничего плохого. Такова природа потоков: операции нескольких потоков могут чередоваться каким угодно непредсказуемым образом. Однако иногда гонки способны приводить к серьёзным проблемам. В этом случае их можно называть злокачественными гонками. Типичные примеры злокачественных гонок – это рассмотренные выше гонки данных, нарушение инвариантов программы, проблемы с блокировкой потоков и со временем жизни объектов.

14. Библиотека для работы со временем

Книга, посвящённая параллельному программированию на современном языке C++, не может считаться полной без отдельной главы, посвящённой средствам для измерения и вычисления времени. Стандартная библиотека языка C++ содержит для этого три основные сущности: момент времени, промежуток времени и часы. Эти понятия тесно связаны между собой.

14.1. Взаимосвязь моментов, промежутков времени и часов

Момент времени определяется точкой отсчёта – так называемым началом эпохи[1] – и промежутком времени, отсчитываемым от начала эпохи.

Промежуток времени – это промежуток между двумя моментами времени. Он измеряется количеством определённых единиц.

Часы характеризуются начальной точкой и единицей измерения. Они позволяют определить текущий момент времени как промежуток от начальной точки, измеренный в данных единицах.

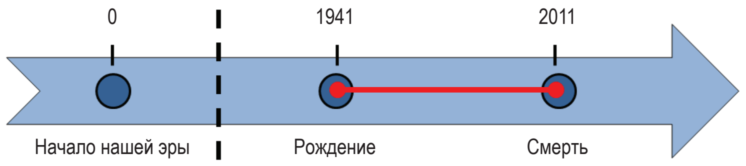
Моменты времени можно между собой сравнивать. Также можно к моменту времени прибавить промежуток и получить другой момент времени. Единица измерения определяет точность, с которой часы могут измерять промежутки времени. Например, в нашей традиции за начало эпохи берут год предполагаемого рождения Христа; для многих практических задач подходящей единицей измерения является год.

Эти три понятия можно проиллюстрировать на примере биографии Денниса Ритчи[2]. Создатель языка C родился в 1941 году и ушёл из жизни в 2011 году. Для простоты будем считать время лишь с точностью до года.

---

1 https://en.wikipedia.org/wiki/Epoch_(reference_date).

2 https://ru.wikipedia.org/wiki/Ритчи,_Деннис.



Годы жизни Денниса Ритчи

За начало нашей эры берётся Рождество Христово. Тогда точки, отмеченные числами 1941 и 2011, определяются началом отсчёта и принятой в данном контексте единицей измерения. Конечно же, начало эпохи тоже представляет собой момент времени. Если вычесть момент времени 1941 из момента 2011, получим промежуток времени. В нашем примере продолжительность измеряется с точностью до года. Как видно из этого вычисления, Деннис Ритчи прожил 70 лет.

Перейдём к более подробному рассмотрению компонентов этой библиотеки.

14.2. Моменты времени

Моменты времени моделируются объектами типа `std::chrono::time_point`. Это шаблон с двумя параметрами. Первый параметр задаёт тип часов. Второй параметр – тип промежутка времени – необязательный: по умолчанию тип промежутка берётся из типа часов.

Шаблон класса `std::chrono::time_point`

```
template<
    class Clock,
    class Duration= typename Clock::duration
>
class time_point;
```

С часами связаны четыре особых момента времени:

- **epoch** – точка отсчёта часов;
- **now** – текущий момент времени;
- **min** – наименьший (наиболее давний) момент времени, который может быть измерен этими часами;
- **max** – наибольший момент времени, который может быть измерен этими часами.

Точность, наименьший и наибольший моменты времени могут быть различны у разных часов. В стандартной библиотеке имеются часы `std::chrono::system_clock`, `std::chrono::steady_cock` и `std::chrono::high_resolution_clock`.

Стандарт языка C++ не даёт гарантий относительно точности, точки отсчёта и диапазона часов. За начало отсчёта часов `std::chrono::system_clock`

обычно берётся 1 января 1970 года – начало эпохи UNIX[1]. Как явствует из названия, часы std::chrono::high_resolution_clock имеют наибольшую точность.

14.2.1. Перевод моментов времени в календарный формат

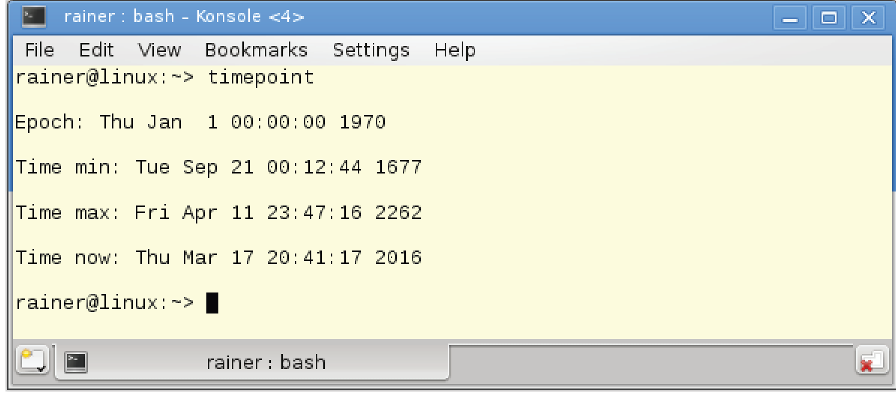
У часов std::chrono::system_clock есть функция to_time_t, которая позволяет преобразовывать моменты времени, отмеренные с помощью этих часов, в значения типа std::time_t. Это значение, в свою очередь, можно преобразовать функцией std::gmtime[2] в календарное время, выраженное в стандарте UTC[3] (всемирное координированное время). Наконец, это календарное время можно передать функции std::asctime[4], чтобы получить текстуальное представление даты и времени.

Отображение даты и времени в текстовом виде

```
1  // timepoint.cpp
2
3  #include <chrono>
4  #include <ctime>
5  #include <iostream>
6  #include <string>
7
8  int main() {
9      std::cout << std::endl;
10
11     std::chrono::time_point<std::chrono::system_clock> sysTimePoint;
12     std::time_t tp= std::chrono::system_clock::to_time_t(sysTimePoint);
13     std::string sTp= std::asctime(std::gmtime(&tp));
14     std::cout << "Epoch: " << sTp << std::endl;
15
16     tp= std::chrono::system_clock::to_time_t(sysTimePoint.min());
17     sTp= std::asctime(std::gmtime(&tp));
18     std::cout << "Time min: " << sTp << std::endl;
19
20     tp= std::chrono::system_clock::to_time_t(sysTimePoint.max());
21     sTp= std::asctime(std::gmtime(&tp));
22     std::cout << "Time max: " << sTp << std::endl;
23
24     sysTimePoint= std::chrono::system_clock::now();
25     tp= std::chrono::system_clock::to_time_t(sysTimePoint);
26     sTp= std::asctime(std::gmtime(&tp));
27     std::cout << "Time now: " << sTp << std::endl;
28 }
```

[1] https://ru.wikipedia.org/wiki/Unix-время.
[2] http://en.cppreference.com/w/cpp/chrono/c/gmtime.
[3] https://ru.wikipedia.org/wiki/Всемирное_координированное_время.
[4] http://en.cppreference.com/w/cpp/chrono/c/asctime.

Эта программа выводит на печать допустимый диапазон часов `std::chrono::system_clock`. На компьютере автора под управлением ОС Linux эти часы имеют своей точкой отсчёта начало эры UNIX и могут представлять даты в интервале от 1677 до 2262 года.



Свойства стандартных системных часов

К моментам времени можно прибавлять промежутки времени. Прибавление промежутка, выводящее момент времени за пределы допустимого диапазона часов, представляет собой неопределённое поведение.

14.2.2. Выход за пределы допустимого диапазона часов

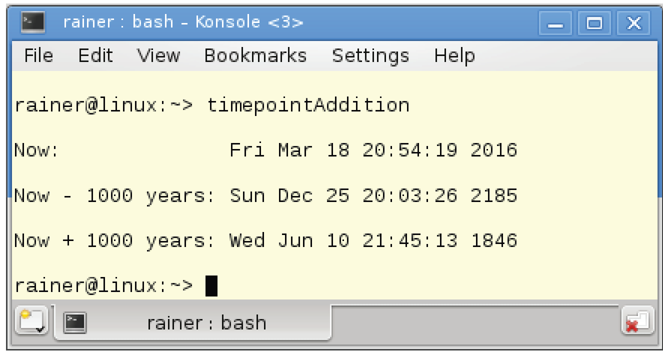
Следующая программа берёт текущий момент времени и прибавляет или отнимает от него 1000 лет. Для простоты не будем обращать внимания на високосные годы и положим, что каждый год содержит ровно 365 дней.

Выход за пределы допустимого диапазона часов

```
1  // timepointAddition.cpp
2
3  #include <chrono>
4  #include <ctime>
5  #include <iostream>
6  #include <string>
7
8  using namespace std::chrono;
9  using namespace std;
10
11 string timePointAsString(const time_point<system_clock>& timePoint){
12   time_t tp= system_clock::to_time_t(timePoint);
13   return asctime(gmtime(&tp));
14 }
```

```
15
16 int main(){
17   cout << endl;
18
19   time_point<system_clock> nowTimePoint= system_clock::now();
20   cout
21     << “Now:                “
22     << timePointAsString(nowTimePoint)
23     << endl;
24
25   const auto thousandYears=  hours(24*365*1000);
26   time_point<system_clock>  historyTimePoint =
27     nowTimePoint - thousandYears;
28   cout
29     << “Now - 1000 years: “
30     << timePointAsString(historyTimePoint)
31     << endl;
32
33   time_point<system_clock>  futureTimePoint =
34     nowTimePoint + thousandYears;
35   cout
36     << “Now + 1000 years: “
37     << timePointAsString(futureTimePoint)
38     << endl;
39 }
```

Для удобства чтения пространство имён `std::chrono` предполагается по умолчанию. Запуск программы демонстрирует, что переполнение счётчика ведёт к неверным результатам. Вычитание тысячи лет из текущего момента времени даёт момент времени в будущем, а прибавление тысячи лет – напротив, переносит в прошлое.



Переполнение при вычислениях со временем

Два разных момента времени, измеренных по одним часам, различаются промежутком, отделяющим момент времени от начала отсчёта. Промежутки поддерживают основные арифметические операции и могут быть представлены в различных единицах измерения.

14.3. Промежутки времени

Для моделирования промежутков времени предназначен шаблон класса `std::chrono::duration`, принимающий два параметра: арифметический тип `Rep` для количества единиц времени и тип `Period`, который определяет единицу времени относительно секунды.

Шаблон класса std::chrono::duration

```
template<
    class Rep,
    class Period = std::ratio<1>
> class duration;
```

По умолчанию за единицу измерения берётся тип `std::ratio<1>`, что соответствует одной секунде и может также быть записано в виде `std::ratio<1, 1>`. Остальное довольно просто: тип `std::ratio<60>` соответствует минуте, а тип `std::ratio<1, 1000>` – миллисекунде. Если в качестве типа `Rep` взят тип с плавающей запятой, возможно измерять время также дробным числом единиц.

В стандарте языка C++ предопределены следующие наиболее важные типы промежутков времени:

```
typedef duration<signed int, nano> nanoseconds;
typedef duration<signed int, micro> microseconds;
typedef duration<signed int, millT> milliseconds;
typedef duration<signed int> seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

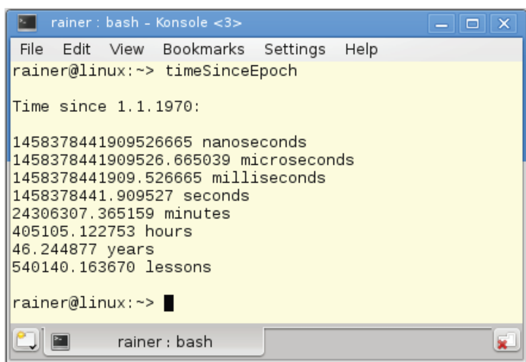
Попытаемся определить, сколько времени прошло с начала эры UNIX (т. е. с 1 января 1970 года). Благодаря псевдонимам для различных типов промежутков времени ответить на этот вопрос довольно просто. Для простоты не будем принимать во внимание високосные годы и примем длительность года за 365 дней.

Измерение промежутка времени в разных единицах

```
1  // timeSinceEpoch.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  using namespace std;
7
8  int main() {
9    cout << fixed << endl;
10
11   cout << “Time since 1.1.1970:\n” << endl;
12
13   const auto timeNow= chrono::system_clock::now();
14   const auto duration= timeNow.time_since_epoch();
15   cout << duration.count() << “ nanoseconds “ << endl;
```

```
16
17    typedef chrono::duration<long double, ratio<1, 1000000>>
18      MyMicroSecondTick;
19    MyMicroSecondTick micro(duration);
20    cout << micro.count() << " microseconds" << endl;
21
22    typedef chrono::duration<long double, ratio<1, 1000>>
23      MyMilliSecondTick;
24    MyMilliSecondTick milli(duration);
25    cout << milli.count() << " milliseconds" << endl;
26
27    typedef chrono::duration<long double> MySecondTick;
28    MySecondTick sec(duration);
29    cout << sec.count() << " seconds " << endl;
30
31    typedef chrono::duration<double, ratio<60>> MyMinuteTick;
32    MyMinuteTick myMinute(duration);
33    cout << myMinute.count() << " minutes" << endl;
34
35    typedef chrono::duration<double, ratio<60*60>> MyHourTick;
36    MyHourTick myHour(duration);
37    cout << myHour.count() << " hours" << endl;
38
39    typedef chrono::duration<double, ratio<60*60*24*365>> MyYearTick;
40    MyYearTick myYear(duration);
41    cout << myYear.count() << " years" << endl;
42
43    typedef chrono::duration<double, ratio<60*45>> MyLessonTick;
44    MyLessonTick myLesson(duration);
45    cout << myLesson.count() << " lessons" << endl;
46
47    cout << endl;
48  }
```

В этой программе объявляются собственные типы, соответствующие единицам измерения времени: микросекунде, миллисекунде, секунде, минуте, часу и году. Кроме того, объявляется ещё одна единица времени – академический час (45 минут). Результат работы программы показан на рисунке.



Измерение промежутка времени от начала эпохи

Проводить вычисления с промежутками времени довольно удобно, этому будет посвящён следующий раздел.

14.3.1. Вычисления с промежутками времени

Типы промежутков времени поддерживают основные арифметические операции. В частности, промежуток времени можно умножать или делить на число. Конечно, промежутки можно сравнивать между собой. Следует подчеркнуть, что все эти операции проводятся с учётом единиц измерения.

Начиная со стандарта C++ 14 работа с промежутками времени становится ещё удобнее. В этой версии стандарта появились литералы для единиц измерения времени.

Предопределённые литералы для единиц времени

Тип	Суффикс	Пример
`std::chrono::hours`	h	5h
`std::chrono::minutes`	min	5min
`std::chrono::seconds`	s	5s
`std::chrono::milliseconds`	ms	5ms
`std::chrono::microseconds`	us	5us
`std::chrono::nanoseconds`	ns	5ns

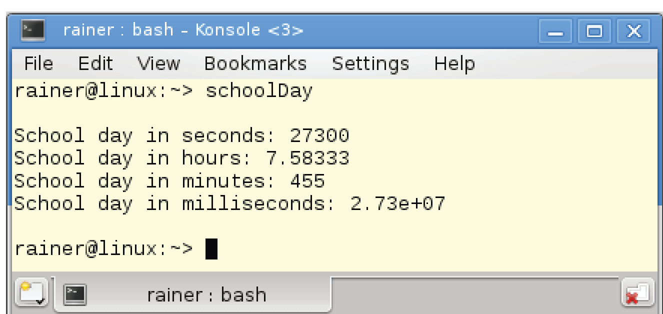
Автор заинтересовался, сколько времени его семнадцатилетний сын Мариус посвящает учёбе школе каждый день. Следующая программа вычисляет ответ и выводит его в различных единицах.

Продолжительность школьного дня в разных единицах

```
1  // schoolDay.cpp
2
3  #include <iostream>
4  #include <chrono>
5
6  using namespace std::literals::chrono_literals;
7  using namespace std::chrono;
8  using namespace std;
9
10 int main(){
11   cout << endl;
12
13   constexpr auto schoolHour= 45min;
14
15   constexpr auto shortBreak= 300s;
16   constexpr auto longBreak= 0.25h;
17
18   constexpr auto schoolWay= 15min;
19   constexpr auto homework= 2h;
20
```

```
21    constexpr auto schoolDaySec =
22        2 * schoolWay +
23        6 * schoolHour +
24        4 * shortBreak +
25        longBreak + homework;
26
27    cout << "School day in seconds: " << schoolDaySec.count() << endl;
28
29    constexpr duration<double, ratio<3600>> schoolDayHour = schoolDaySec;
30    constexpr duration<double, ratio<60>> schoolDayMin = schoolDaySec;
31    constexpr duration<double, ratio<1,1000>> schoolDayMilli= schoolDaySec;
32
33    cout << "School day in hours: " << schoolDayHour.count() << endl;
34    cout << "School day in minutes: " << schoolDayMin.count() << endl;
35    cout << "School day in milliseconds: " << schoolDayMilli.count() << endl;
36
37    cout << endl;
38  }
```

Здесь объявлены единицы времени, соответствующие академическому часу, короткой перемене, длинной перемене, продолжительности дороги в школу или из школы, а также продолжительности подготовки домашних заданий. Результат вычислений доступен даже на этапе компиляции. Результат запуска программы показан на рисунке.



Продолжительность школьного дня в разных единицах

Вычисления на этапе компиляции

Промежутки времени, заданные литералами, результат вычислений в секундах schoolDaySec, а также этот промежуток, выраженные в других единицах, являются константами этапа компиляции, о чём свидетельствует ключевое слово constexpr. Таким образом, все вычисления выполняются при сборке программы, и лишь вывод результата происходит на этапе выполнения.

Точность, с которой можно измерить промежутки времени, зависит от используемых для этого часов. В стандартной библиотеке языка C++ определены три типа часов, о них пойдёт речь в следующем разделе.

14.4. Типы часов

Наличие в стандарте трёх типов часов не может не вызвать вопрос, чем они между собой отличаются.

Тип `std::chrono::system_clock` соответствует общесистемным часам реального времени или, как их ещё называют, настенным часам системы. Только этот тип обладает статическими функциями `to_time_t` и `from_time_t`, позволяющими преобразовывать измеренные этими часами моменты времени в календарное время и обратно.

Тип `std::chrono::steady_clock` – единственный тип часов, гарантирующий монотонность показаний. В отличие от общесистемных часов, которые пользователь может перевести назад, эти часы переводить нельзя. Следовательно, именно эти часы лучше всего использовать для измерения промежутков времени между событиями.

Тип `std::chrono::high_resolution_clock` – это часы с наибольшей доступной точностью, однако это может быть псевдоним для одного из двух предыдущих типов.

Отсутствие гарантий у стандартных часов

Стандарт языка C++ не предоставляет никаких гарантий относительно точности, точки отсчёта или допустимого диапазона этих типов часов. Чаще всего в типе `std::chrono::system_clock` используется начало эры UNIX (1 января 1970 года), а в типе `std::chrono::steady_clock` за точку отсчёта берётся время запуска операционной системы на машине пользователя.

14.4.1. Точность и монотонность часов

Весьма интересно узнать, какие из часов монотонны и какую точность измерений они обеспечивают. Монотонность означает, что часы не могут быть переведены, т. е. что их показания не могут уменьшаться. Ответы на эти вопросы можно получить у самих часов.

Точность и монотонность трёх типов часов

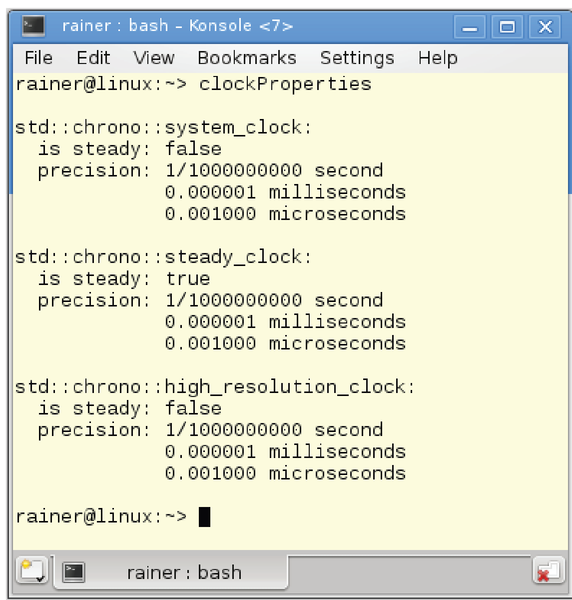
```
1  // clockProperties.cpp
2
3  #include <chrono>
4  #include <iomanip>
5  #include <iostream>
6
7  using namespace std::chrono;
8  using namespace std;
9
10 template <typename T>
11 void printRatio() {
12     cout
13         << "  precision: "
```

```
14          << T::num
15          << "/"
16          << T::den
17          << " second "
18          << endl;
19      typedef typename ratio_multiply<T,kilo>::type MillSec;
20      typedef typename ratio_multiply<T,mega>::type MicroSec;
21      cout << fixed;
22      cout
23          << "             "
24          << static_cast<double>(MillSec::num)/MillSec::den
25          << " milliseconds "
26          << endl;
27      cout
28          << "             "
29          << static_cast<double>(MicroSec::num)/MicroSec::den
30          << " microseconds " << endl;
31  }
32
33  int main(){
34      cout << boolalpha << endl;
35
36      cout << "std::chrono::system_clock: " << endl;
37      cout << "  is steady: " << system_clock::is_steady << endl;
38      printRatio<chrono::system_clock::period>();
39
40      cout << endl;
41
42      cout << "std::chrono::steady_clock: " << endl;
43      cout << "  is steady: " << chrono::steady_clock::is_steady << endl;
44      printRatio<chrono::steady_clock::period>();
45
46      cout << endl;
47
48      cout << "std::chrono::high_resolution_clock: " << endl;
49      cout << "  is steady: " << chrono::high_resolution_clock::is_steady
50           << endl;
51      printRatio<chrono::high_resolution_clock::period>();
52
53      cout << endl;
54  }
```

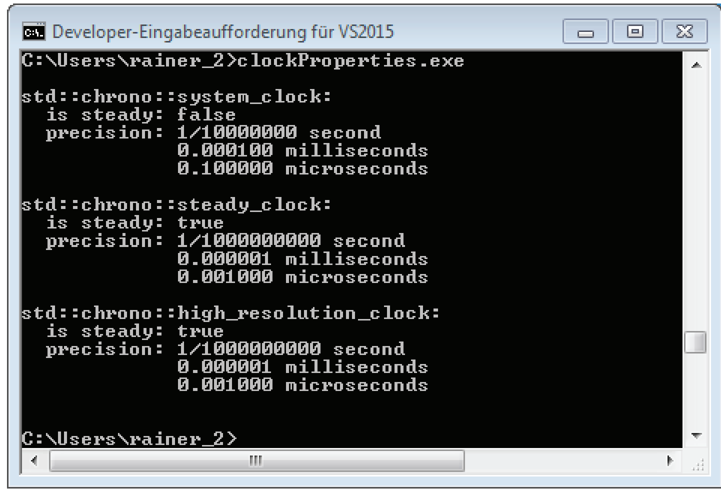
Для каждого типа часов эта программа сначала печатает, являются ли они монотонными. Функция `printRatio` немного труднее для понимания. В первую очередь она печатает точность часов в секундах в виде простой дроби. Затем с помощью шаблона функции `std::ratio_multiply` и констант `std::kilo` и `std::mega` эта величина приводится к миллисекундам и микросекундам и выводится в виде десятичной дроби. Подробности о вычислениях с рациональными числами можно найти на сайте-справочнике cppreference.com[1].

[1] http://en.cppreference.com/w/cpp/numeric/ratio.

Программа ведёт себя различным образом в системах Linux и Windows. Часы `std::chrono::system_clock` намного точнее в системе Linux, а в системе Windows часы `std::chrono::high_resultion_clock` оказываются монотонными.



Свойства стандартных часов в системе Linux



Свойства стандартных часов в системе Windows

Стандарт языка C++ ничего не говорит о том, какой момент времени должен быть взят за точку отсчёта в тех или иных часах. Это значение можно вычислить в программе.

14.4.2. Нахождение точки отсчёта часов

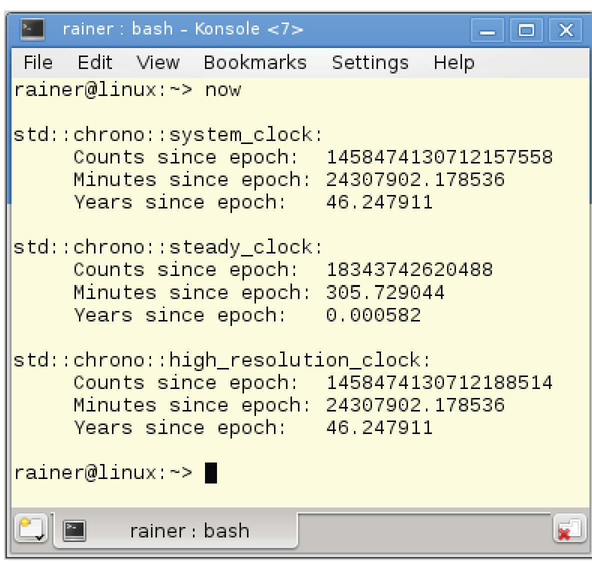
Благодаря вспомогательной функции `time_since_epoch` у часов каждого типа можно узнать, сколько времени прошло с начала их эпохи.

Вычисление точки отсчёта каждого типа часов

```
1  // now.cpp
2
3  #include <chrono>
4  #include <iomanip>
5  #include <iostream>
6
7  using namespace std::chrono;
8
9  template <typename T>
10 void durationSinceEpoch(const T dur) {
11     std::cout
12         << "     Counts since epoch:  "
13         << dur.count()
14         << std::endl;
15     typedef duration<double, std::ratio<60>> MyMinuteTick;
16     const MyMinuteTick myMinute(dur);
17     std::cout << std::fixed;
18     std::cout
19         << "     Minutes since epoch: "
20         << myMinute.count()
21         << std::endl;
22     typedef duration<double, std::ratio<60*60*24*365>> MyYearTick;
23     const MyYearTick myYear(dur);
24     std::cout
25         << "     Years since epoch:   "
26         << myYear.count()
27         << std::endl;
28 }
29
30 int main() {
31
32     std::cout << std::endl;
33
34     system_clock::time_point timeNowSysClock = system_clock::now();
35     system_clock::duration timeDurSysClock =
36         timeNowSysClock.time_since_epoch();
37     std::cout << "system_clock: " << std::endl;
38     durationSinceEpoch(timeDurSysClock);
39
40     std::cout << std::endl;
41
```

```
42      const auto timeNowStClock = steady_clock::now();
43      const auto timeDurStClock= timeNowStClock.time_since_epoch();
44      std::cout << "steady_clock: " << std::endl;
45      durationSinceEpoch(timeDurStClock);
46
47      std::cout << std::endl;
48
49      const auto timeNowHiRes = high_resolution_clock::now();
50      const auto timeDurHiResClock= timeNowHiRes.time_since_epoch();
51      std::cout << "high_resolution_clock: " << std::endl;
52      durationSinceEpoch(timeDurHiResClock);
53
54      std::cout << std::endl;
55
56  }
```

Переменные `timeDurSysClock`, `timeDurStClock` и `timeDurHiResClock` содержат промежутки времени, прошедшие после начала эпохи соответствующих часов. Если бы не ключевое слово `auto` и автоматический вывод типов, типы моментов времени и промежутков пришлось бы записывать в явном виде, что слишком многословно. Функция `durationSinceEpoch` отображает промежуток времени в различных единицах: сначала в собственных единицах самих часов, затем в минутах и, наконец, в годах. При этом для простоты продолжительность года взята за 365 дней, високосные годы не учитываются. Выводимые программой данные различаются в системах Linux и Windows.



Вычисление точки отсчёта часов в системе Linux

```
Developer-Eingabeaufforderung für VS2015
C:\Users\rainer_2>now

std::chrono::system_clock:
     Counts since epoch:  14584738731148801
     Minutes since epoch: 24307897.885248
     Years since epoch:   46.247903

std::chrono::steady_clock:
     Counts since epoch:  23516425251961
     Minutes since epoch: 391.940421
     Years since epoch:   0.000746

std::chrono::high_resolution_clock:
     Counts since epoch:  23516430130632
     Minutes since epoch: 391.940502
     Years since epoch:   0.000746


C:\Users\rainer_2>
```

Вычисление точки отсчёта часов в системе Windows

Чтобы правильно проинтерпретировать полученные данные, нужно отметить, что компьютер под управлением ОС Linux к моменту запуска программы был включен около 5 часов (305 минут), а компьютер с ОС Windows работал около 6 часов (391 минуту).

По этим рисункам видно, что на машине с ОС Linux часы `std::chrono::system_clock` и `std::chrono::high_resolution_clock` ведут отсчёт от начала эры UNIX, а часы `std::chrono::steady_clock` – от включения компьютера. Если в ОС Linux часы `std::high_resolution_clock`, по-видимому, представляют собой псевдоним для часов `std::system_clock`, то под ОС Windows это похоже на псевдоним для часов `std::chrono::steady_clock`. Этот вывод хорошо согласуется с результатами, полученными в предыдущем разделе при определении точности и монотонности часов.

Библиотечные средства для работы со временем удобно использовать для погружения потока в сон. В качестве аргумента соответствующим функциям передаются моменты времени и промежутки времени.

14.5. Приостановка и ограниченное ожидание

Многочисленные функции для ожидания блокировщиков, переменных условия, фьючерсов имеют одну общую особенность: все они так или иначе работают со временем.

14.5.1. Соглашения об именовании

Имена функций, погружающих поток в ограниченное по времени ожидание, в основном образованы по единому шаблону. Функции, имена которых за-

канчиваются на «_for», принимают в качестве параметра промежуток времени; функции же с именами, заканчивающимися на «_until», принимают момент времени. Имена некоторых функций, впрочем, не имеют таких суффиксов. Пусть in2min – это момент времени в будущем, отстоящий на две минуты от настоящего (несмотря на использование ключевого слова auto, эта запись всё равно выглядит излишне многословной):

```
auto in2min= std::chrono::steady_clock::now() + std::chrono::minutes(2);
```

Ниже приведён перечень таких функций-членов из разных классов, относящихся к управлению потоками, примитивами блокировки и заданиями.

Сущность	Функция _until	Функция _for
std::thread th	th.sleep_until(in2min)	th.sleep_for(2s)
std::unique_lock lk	lk.try_lock_until(in2min)	lk.try_lock(2s)
std::condition_variable cv	cv.wait_until(in2min)	cv.wait_for(2s)
std::future fu	fu.wait_until(in2min)	fu.wait_for(2s)
std::shared_future shFu	shFu.wait_until(in2min)	shFu.wait_for(2s)

Здесь очень пригодились появившиеся в стандарте C++ 14 литералы для единиц времени: запись 2с в этом примере означает 2 секунды.

Рассмотрим теперь различные стратегии ожидания.

14.5.2. Стратегии ожидания

Основная идея следующей программы состоит в том, что один и тот же объект-обещание поставляет результат четырём фьючерсам. Это возможно, если фьючерсы имеют тип std::shared_future. Каждый из четырёх фьючерсов использует свою стратегию ожидания. Как обещание, так и все фьючерсы выполняются в отдельных потоках. Для простоты будем далее говорить об ожидающих потоках, хотя на самом деле ожидает оповещения именно фьючерс. За подробными сведениями об обещаниях и фьючерсах можно обратиться к разделу 3.9. Ожидающие потоки используют следующие четыре стратегии:

- ждать обещания не более 4 секунд;
- ждать обещания до 20 секунд;
- запросить результат обещания и, если он не готов, заснуть на 700 миллисекунд, затем повторить;
- запросить результат обещания и, если он не готов, заснуть сперва на одну миллисекунду, при каждой следующей попытке удваивая время ожидания.

Различные стратегии ожидания

```
1   // sleepAndWait.cpp
2
3   #include <utility>
4   #include <iostream>
```

```
5   #include <future>
6   #include <thread>
7   #include <utility>
8
9   using namespace std;
10  using namespace std::chrono;
11
12  mutex coutMutex;
13
14  long double getDifference(
15      const steady_clock::time_point& tp1,
16      const steady_clock::time_point& tp2)
17  {
18      const auto diff= tp2 - tp1;
19      const auto res= duration <long double, milli> (diff).count();
20      return res;
21  }
22
23  void producer(promise<int>&& prom) {
24      cout << "PRODUCING THE VALUE 2011\n\n";
25      this_thread::sleep_for(seconds(5));
26      prom.set_value(2011);
27  }
28
29  void consumer(
30      shared_future<int> fut,
31      steady_clock::duration dur)
32  {
33      const auto start = steady_clock::now();
34      future_status status = fut.wait_until(steady_clock::now() + dur);
35      if ( status == future_status::ready ) {
36          lock_guard<mutex> lockCout(coutMutex);
37          cout
38              << this_thread::get_id()
39              << " ready => Result: " << fut.get()
40              << endl;
41      }
42      else{
43          lock_guard<mutex> lockCout(coutMutex);
44          cout
45              << this_thread::get_id()
46              << " stopped waiting."
47              << endl;
48      }
49
50      const auto end = steady_clock::now();
51      lock_guard<mutex> lockCout(coutMutex);
52      cout
53          << this_thread::get_id()
54          << " waiting time: "
55          << getDifference(start,end)
56          << " ms"
```

```
57          << endl;
58  }
59
60  void consumePeriodically(shared_future<int> fut) {
61      const auto start = steady_clock::now();
62      future_status status;
63      do {
64          this_thread::sleep_for(milliseconds(700));
65          status = fut.wait_for(seconds(0));
66          if (status == future_status::timeout) {
67              lock_guard<mutex> lockCout(coutMutex);
68              cout
69                  << “     “ << this_thread::get_id()
70                  << “ still waiting.”
71                  << endl;
72          }
73          if (status == future_status::ready) {
74              lock_guard<mutex> lockCout(coutMutex);
75              cout
76                  << “     “
77                  << this_thread::get_id()
78                  << “ waiting done => Result: “
79                  << fut.get()
80                  << endl;
81          }
82      } while (status != future_status::ready);
83
84      const auto end = steady_clock::now();
85      lock_guard<mutex> lockCout(coutMutex);
86      cout
87          << “     “
88          << this_thread::get_id()
89          << “ waiting time: “
90          << getDifference(start,end)
91          << “ ms”
92          << endl;
93  }
94
95  void consumeWithBackoff(shared_future<int> fut) {
96      const auto start = steady_clock::now();
97      future_status status;
98      auto dur = milliseconds(1);
99      do {
100         this_thread::sleep_for(dur);
101         status = fut.wait_for(seconds(0));
102         dur *= 2;
103         if (status == future_status::timeout) {
104             lock_guard<mutex> lockCout(coutMutex);
105             cout
106                 << “         “
107                 << this_thread::get_id()
108                 << “ still waiting.”
```

```
109                 << endl;
110         }
111         if (status == future_status::ready) {
112             lock_guard<mutex> lockCout(coutMutex);
113             cout
114                 << "          "
115                 << this_thread::get_id()
116                 << " waiting done => Result: "
117                 << fut.get()
118                 << endl;
119         }
120     } while (status != future_status::ready);
121 
122     const auto end = steady_clock::now();
123     lock_guard<mutex> lockCout(coutMutex);
124     cout
125         << "          "
126         << this_thread::get_id()
127         << " waiting time: "
128         << getDifference(start,end)
129         << " ms"
130         << endl;
131 }
132 
133 int main() {
134     cout << endl;
135 
136     promise<int> prom;
137     shared_future<int> future = prom.get_future();
138     thread producerThread(producer, move(prom));
139 
140     thread consumerThread1(consumer, future, seconds(4));
141     thread consumerThread2(consumer, future, seconds(20));
142     thread consumerThread3(consumePeriodically, future);
143     thread consumerThread4(consumeWithBackoff, future);
144 
145     consumerThread1.join();
146     consumerThread2.join();
147     consumerThread3.join();
148     consumerThread4.join();
149     producerThread.join();
150 
151     cout << endl;
152 }
```

В главной функции, в строке 133, создаётся объект-обещание. Затем на основе обещания в строке создаётся фьючерс 134, а обещание перемещается в отдельный поток (строка 135). Обещание необходимо именно переместить, поскольку обещания не поддерживают копирование. Для фьючерсов типа `std::shared_future` копирование разрешено, это выполняется в строках 137–140.

Прежде чем перейти к описанию алгоритмов, выполняющихся в потоках, нужно сказать несколько слов о вспомогательной функции `getDifference` (строки 14–21). Эта функция принимает два аргумента – момента времени – и возвращает длительность промежутка между ними в миллисекундах. Эта функция используется в программе несколько раз. Созданные в главной функции потоки выполняют каждый свой алгоритм.

- Поток `producerThread` выполняет функцию `producer` (строки 23–27), которая устанавливает результат объекта-обещания спустя 5 секунд ожидания. Именно этого результата ожидают фьючерсы.
- Поток `consumerThread1` выполняет функцию `consumer` (строки 29–58), которая ждёт результата от фьючерса не более 4 секунд и продолжает работу. Этого времени недостаточно, чтобы поток-производитель завершил свою работу и опубликовал результат.
- Поток `consumerThread2` выполняет ту же функцию `consumer` (строки 29–58), но на этот раз она ожидает появления результата не более 20 секунд.
- Поток `consumerThread3` выполняет функцию `consumePeriodically` (строки 60–93). Она засыпает на 700 миллисекунд (строка 64), затем без ожидания проверяет наличие результата в обещании (строка 65). Если результат фьючерса имеется в наличии, он выводится на печать, в противном случае функция повторяет цикл ожидания.
- Поток `consumerThread4` выполняет функцию `consumeWithBackoff` (строки 95–131). В первый раз она выжидает одну миллисекунду, затем увеличивает этот интервал вдвое на каждой итерации цикла ожидания. Во всём остальном стратегия подобна предыдущей.

Теперь необходимо разобраться, как в этой программе происходит синхронизация потоков. Как часы, измеряющие реальное время выполнения программы, так и объект `std::cout` доступны всем потокам, но это ещё не делает синхронизацию необходимой. Во-первых, функция `now` потокобезопасна. Во-вторых, стандарт гарантирует, что каждая операция вывода в объект `std::cout` потокобезопасна. Блокировка с мьютексом требуется только там, где подряд осуществляется несколько операций вывода[1].

Несмотря на то что потоки выводят свои сообщения последовательно, один за другим, общий результат работы программы оказывается непростым для понимания.

---

[1] Это утверждение ошибочно. Пока, например, функция `consumeWithBackoff` в одном потоке выполнения выполняет подряд несколько операций вывода в объект `std::cout`, в другом потоке функция `producer` без блокировки может выводить в него свой текст. Это означает, что операция вывода из функции `producer` может вклиниться между несколькими операциями вывода, выполняемыми функцией `consumeWithBackoff`. Скажем, последняя успевает вывести пробелы и идентификатор потока, затем выводится сообщение из функции `producer`, после чего функция `consumeWithBackoff` выполняет оставшиеся операции вывода. Если хотя бы в одном из потоков выполнения имеется несколько соединённых в цепочку операций вывода в объект `std::cout`, мьютексом должны быть защищены все операции вывода, даже одиночные. – *Прим. перев.*

```
rainer : bash - Konsole <4>
File  Edit  View  Bookmarks  Settings  Help

rainer@linux:~> sleepAndWait

PRODUCING THE VALUE 2011

         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
         140135671097088 still waiting.
     140135679489792 still waiting.
         140135671097088 still waiting.
     140135679489792 still waiting.
         140135671097088 still waiting.
     140135679489792 still waiting.
     140135679489792 still waiting.
     140135679489792 still waiting.
140135696275200 stopped waiting.
140135696275200 waiting time: 4000.18 ms
         140135671097088 still waiting.
     140135679489792 still waiting.
     140135679489792 still waiting.
140135687882496 ready => Result: 2011
140135687882496 waiting time: 5000.3 ms
     140135679489792 waiting done => Result: 2011
     140135679489792 waiting time: 5601.76 ms
         140135671097088 waiting done => Result: 2011
         140135671097088 waiting time: 8193.81 ms

rainer@linux:~> █
rainer : bash
```

Различные стратегии ожидания

Первым выводится сообщение от потока-производителя, который управляет объектом-обещанием. Все остальные сообщения исходят от потоков-потребителей, которые запрашивают результат обещания через фьючерсы. Поток-потребитель `consumerThread4` пытается получить результат из фьючерса. Помимо прочего, он выводит идентификатор потока. Его сообщения сдвинуты вправо на 8 символов. Затем со сдвигом на 4 символа следует сообщение от потока `consumerThread3`. Сообщения от потоков `consumerThread1` и `consumerThread2` выводятся без отступа.

- Поток `consumerThread1` безуспешно ждёт 4000,18 мс и завершается, так и не получив результата от фьючерса.
- Поток `consumerThread2` имеет право ждать до 20 с, но получает ожидаемое значение уже через 5000,3 мс.
- Поток `consumerThread3` получает значение из фьючерса спустя 5601,76 мс. Это примерно равно 5600 мс, т. е. он 8 раз выполняет цикл ожидания по 700 мс.
- Наконец, поток `consumerThread4` завершается через 8193,81 мс. Иными словами, он ждёт лишних 3 с после того, как результат уже появляется во фьючерсе.

15. Обзор инструментального средства CppMem

Система CppMem[1] – это интерактивное инструментальное средство, позволяющее исследовать поведение небольших фрагментов кода с точки зрения модели памяти языка C++. Этот инструмент достоин занять место в арсенале каждого разработчика, всерьёз имеющего дела с моделями памяти. Системой можно пользоваться на сайте или установить её на локальную машину. Она может приносить программисту двоякую пользу.

1. С её помощью можно проверять корректность небольших участков кода. Основываясь на присущей языку C++ модели памяти, этот инструмент анализирует все возможные пути выполнения параллельной программы, представляет каждый из них визуальной схемой и снабжает эти схемы подробностями в текстовом виде.
2. Чрезвычайно точный анализ поведения программ, генерируемый этим инструментом, позволяет программисту глубоко понять модель памяти.

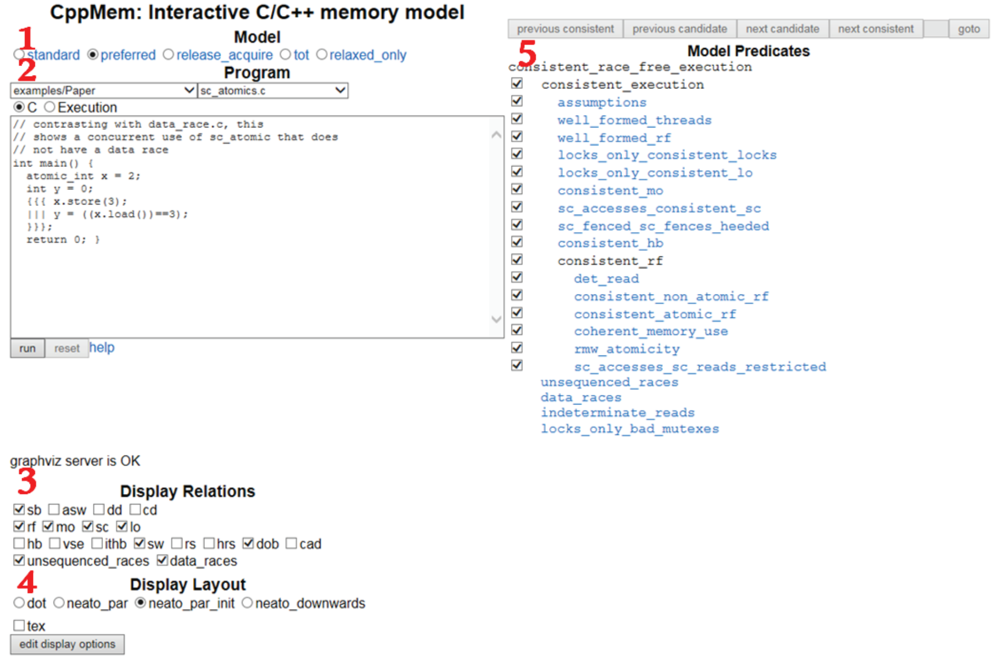
Общая особенность многих мощных инструментов такова, что пользователю приходится преодолеть ряд трудностей, прежде чем в полной мере воспользоваться богатыми возможностями. Инструмент CppMem выдает очень подробный анализ, относящийся к необычайно сложной области, и обладает множеством настраиваемых параметров. В следующих разделах подробно описаны различные аспекты этого инструмента.

15.1. Упрощённое введение

В этом разделе дадим краткое описание системы, основанное на конфигурации по умолчанию. Такое начальное введение может послужить лишь

[1] http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/.

основой для первых самостоятельных экспериментов и отправной точкой для последующего более подробного изложения.



Конфигурация системы CppMem по умолчанию

Для удобства дальнейшего изложения нужные места помечены на рисунке красными цифрами.

15.1.1. Выбор модели

Переключатель под цифрой 1 предназначен для выбора одной из определённых стандартом языка C++ моделей памяти. Под названием «preferred» (с англ. «предпочтительная») скрывается несколько упрощённая, но в целом эквивалентная модель из стандарта C++ 11.

15.1.2. Выбор программы

Программа для анализа должна быть представлена в упрощённом синтаксисе, похожем на синтаксис C++ 11. В общем случае невозможно скопировать в систему CppMem программу на языке C или C++ в неизменном виде.

Пользователь может выбрать из множества заранее заготовленных примеров программ, иллюстрирующих разные типовые сценарии многопоточного программирования. Подробное описание этих программ можно найти

в превосходно написанной статье[1] о математических моделях параллельного программирования на языке C++. Кроме того, пользователь, конечно, может подавать для анализа собственный код.

Система CppMem предназначена для анализа параллельных программ, и только для этого. Поэтому для всех основных элементов многопоточного программирования в ней предусмотрены удобные сокращённые обозначения:

- например, два параллельных потока легко определить с помощью обозначения вида `{{{ ... ||| ... }}}` (где троеточием обозначены операции, выполняемые потоками);
- выражение вида `x.readvalue(1)` указывает системе CppMem, что она должна проанализировать такие варианты выполнения программы, где переменная `x` имеет значение 1.

15.1.2.1. Отображаемые отношения

Настройки, показанные на рисунке под цифрой 3, определяют, какие отношения и связи между операциями (чтения, записи, модификации атомарных переменных, преодоление барьеров и захват блокировок) должны отображаться на схеме. Чтобы на графической схеме программы отображались те или иные связи, их нужно в явном виде включить на данной панели. Ниже перечислены отношения между операциями, отображение которых включено по умолчанию.

- Первичные отношения:
  - sb (sequenced-before) – операция происходит ранее другой операции в том же потоке выполнения;
  - rf (read from) – операция читает результат другой операции;
  - mo (modification order) – операции связаны между собой определённым порядком, так как модифицируют один и тот же объект;
  - sc (sequentially consistent) – последовательная согласованность: операции связаны единым глобальным порядком выполнения;
  - lo (lock order) – операции связаны порядком доступа к блокировке.
- Производные отношения:
  - sw (synchronises-with) – операция синхронизируется с другой операцией;
  - dob (dependency-ordered-before) – одна операция выполняется раньше, так как другая зависит от её результата;
  - unsequenced_races – гонка в пределах одного потока;
  - data_races – гонка данных между потоками.

15.1.2.2. Параметры отображения

Настройки в блоке 4 позволяют настроить способ отрисовки граф-схемы средством Doxygraph[2].

---

[1] http://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf.

[2] https://sourceforge.net/projects/doxygraph/.

15.1.2.3. Предикаты модели

Блок настроек, помеченный на рисунке цифрой 5, позволяет настраивать условия, способные вызвать несогласованное (т. е. подверженное гонкам данных) выполнение программы. Всякий раз, когда анализатор обнаруживает несогласованное поведение, пользователь может увидеть точное описание, почему оно считается несогласованным. В наших примерах эти настройки не используются.

Этого краткого введения должно быть достаточно, чтобы начать пользоваться системой CppMem. Более подробные сведения можно найти в документации[1]. Пришло время попробовать систему CppMem в действии. Вместе с системой идёт множество примеров.

15.1.3. Примеры программ

Примеры программ, включённые в поставку системы, иллюстрируют типичные ситуации, возникающие при многопоточном и, в частности, неблокирующем программировании. Примеры разделены на несколько категорий.

15.1.3.1. Примеры из статьи

Категория `examples/Paper` состоит из тех примеров, которые подробно разобраны в основополагающей статье[2] о математических моделях параллельного программирования на языке C++:

- `data_race.c` – гонка данных по одной переменной;
- `partial_sb.c` – упорядоченность операций в однопоточной программе;
- `unsequenced_race.c` – гонка данных по одной переменной в связи с неопределённым порядком вычисления;
- `sc_atomic.c` – корректное использование атомарных переменных;
- `thread_create_and_asw.c` – синхронизация в связи с созданием потока.

Начнём с первого из этих примеров.

15.1.3.1.1. Проверка работоспособности системы

Для проверки системы в действии на простом примере нужно из списка примеров выбрать программу `data_race.c`. Нажатие на кнопку запуска сразу обнаруживает гонку данных.

Для простоты объяснения некоторые места на этот рисунке помечены красными цифрами, будем на них ссылаться.

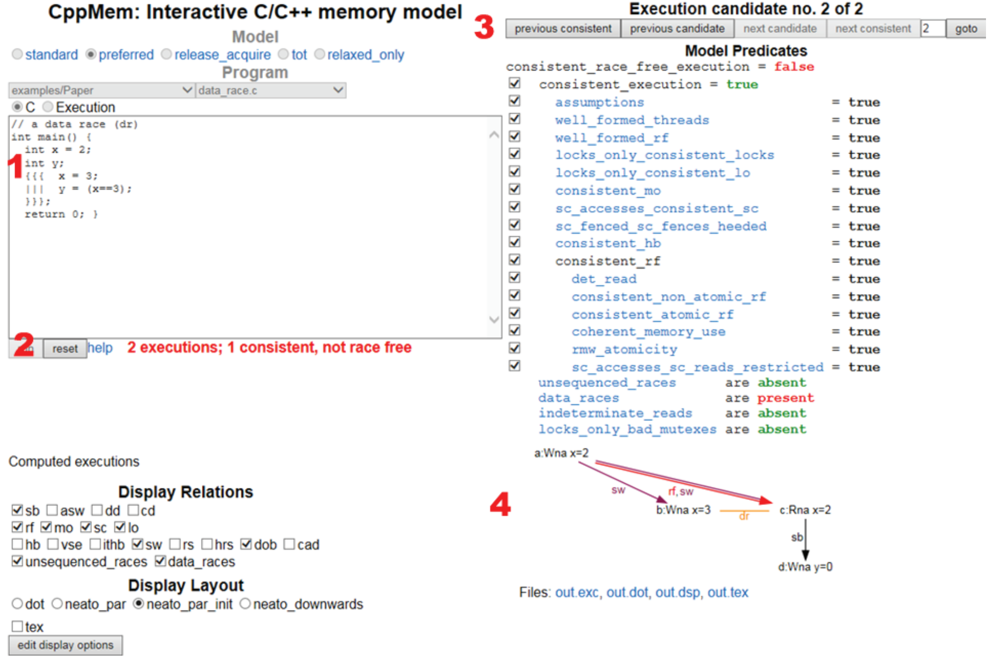
1. Гонку данных довольно легко заметить в коде программы невооружённым глазом. Один поток пишет, а другой читает значение переменной `x` без всякой синхронизации.
2. В соответствии с моделью памяти C++ возможны два возможных способа чередования операций. Один из них вполне согласуется с правилами

[1] http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/help.html.

[2] https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf.

выбранной модели памяти: а именно тот, при котором читающий поток видит значение переменной x, установленное пишущим потоком. Это отношение между операциями показано в виде стрелки с пометками rf и sw.

3. Переключение между разными вариантами выполнения потоков – завораживающее зрелище.
4. На графической схеме отображаются все виды отношений между операциями, выбранные в блоке настроек «Display Relations».
   a. Вершина, обозначенная на схеме как «`a: Wna x=2`», соответствует стартовой точке программы и представляет неатомарную операцию записи (Wna означает «write non-atomic»).
   b. Самое интересное в этом графе – ребро между вершинами b (неатомарная запись) и c (неатомарное чтение). Это и есть гонка данных по переменной x.



Обнаружение гонки данных инструментом CppMem

15.1.3.2. Другие категории примеров

Каждая из оставшихся категорий примеров посвящена одному определённому аспекту неблокирующего программирования. Примеры в каждой категории иллюстрируют соответствующее понятие с разных сторон, в том числе с использованием разных порядков доступа к памяти. Более подробное описание этих категорий можно найти в упоминавшейся выше статье о математическом моделировании параллельного программирования на языке

C++. Приведём ниже несколько примеров, отдавая предпочтение семантике последовательной согласованности.

15.1.3.2.1. Буферизация при записи

В этом примере из раздела examples/SB_store_buffering каждый из двух потоков записывает значение в свою переменную, а затем читает значение из переменной, которую записал другой поток. В комментарии в начале программы поставлен вопрос: могут ли оба потока прочитать из переменных значение 0.

```
// SB+sc_sc+sc_sc
// Store Buffering (or Dekker's), with all four accesses SC atomics
// Question: can the two reads both see 0 in the same execution?
int main() {
  atomic_int x=0; atomic_int y=0;
  {{{ { y.store(1,memory_order_seq_cst);
        r1=x.load(memory_order_seq_cst); }
  ||| { x.store(1,memory_order_seq_cst);
        r2=y.load(memory_order_seq_cst); } }}}
  return 0;
}
```

15.1.3.2.2. Передача сообщений

Пример взят из раздела `examples/MP_message_passing`. Один поток присваивает значение в неатомарную переменную и затем устанавливает атомарный флаг, другой поток ждёт этого флага и читает данные из неатомарной переменной. В комментарии спрашивается: гарантируется ли, что операция чтения непременно увидит в переменной новое значение 1, а не старое 0.

```
// MP+na_sc+sc_na
// Message Passing, of data held in non-atomic x,
// with sc atomic stores and loads on y giving
// release/acquire synchronisation
// Question: is the read of x required to see the new data value 1
// rather than the initial state value 0?
int main() {
int x=0; atomic_int y=0;
{{{ { x=1;
      y.store(1,memory_order_seq_cst); }
||| { r1=y.load(memory_order_seq_cst).readsvalue(1);
      r2=x; } }}}
  return 0;
}
```

15.1.3.2.3. Буферизация при чтении

Следующий пример взят из раздела examples/LB_load_buffering. Могут ли обе операции чтения в каждом из следующих потоков увидеть значения, установленные другим потоком?

```
// LB+sc_sc+sc_sc
// Load Buffering, with all four accesses sequentially
// consistent atomics
```

```
// Question: can the two reads both see 1 in the same execution?
int main() {
  atomic_int x=0; atomic_int y=0;
  {{{ { r1=x.load(memory_order_seq_cst);
        y.store(1,memory_order_seq_cst); }
  ||| { r2=y.load(memory_order_seq_cst);
        x.store(1,memory_order_seq_cst); } }}}
  return 0;
}
```

15.1.3.2.4. Зависимости между операциями чтения и записи

Первый поток присваивает новое значение переменной x. Второй поток читает значение этой переменной и присваивает его в переменную y. Третий поток читает переменные y и x. Увидит ли третий поток результат присваивания из первого потока? Пример взят из раздела examples/WRC.

```
// WRC
// the question is whether the final read is required to see 1
// With two release/acquire pairs, it is
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,mo_release);
  ||| { r1=x.load(mo_acquire).readsvalue(1);
        y.store(1,mo_release); }
  ||| { r2=y.load(mo_acquire).readsvalue(1);
        r3=x.load(mo_relaxed); }
  }}}
  return 0;
}
```

15.1.3.2.5. Независимые операции чтения и записи

Наконец, пример examples\IRIW. Два потока пишут в две различные атомарные переменные, ещё два потока читают значения этих переменных. Могут ли два читающих потока увидеть результат двух операций записи в различном порядке?

```
// IRIW with release/acquire
// the question is whether the reading threads have
// to see the writes to x and y in the same order.
// With release/acquire, they do not.
int main() {
  atomic_int x = 0; atomic_int y = 0;
  {{{ x.store(1, memory_order_release);
  ||| y.store(1, memory_order_release);
  ||| { r1=x.load(memory_order_acquire).readsvalue(1);
        r2=y.load(memory_order_acquire).readsvalue(0); }
  ||| { r3=y.load(memory_order_acquire).readsvalue(1);
        r4=x.load(memory_order_acquire).readsvalue(0); }
  }}};
  return 0;
}
```

16. Глоссарий

Представленный здесь перечень терминов никоим образом не может считаться полным, это лишь вспомогательный справочный материал по некоторым избранным понятиям, затронутым в книге.

ACID – этой аббревиатурой обозначается четвёрка основных требований, предъявляемых к транзакции. Транзакция должна быть атомарной (Atomicity), согласованной (Consistency), изолированной (Isolation) и прочной (Durability). Нет смысла требовать прочности от действий над оперативной памятью, поэтому для транзакционной памяти (в частности, в языке C++) остаются три свойства:

- **атомарность**: из действий, заключённых в транзакцию, в любой момент времени доступен результат либо всех, либо ни одного;
- **согласованность**: система всё время находится в согласованном состоянии. Все транзакции глобально упорядочены;
- **изолированность**: каждая транзакция выполняется полностью независимо от других транзакций;
- **прочность** (только для баз данных; неприменимо к транзакциям над памятью): если клиент получил подтверждение транзакции, то даже в случае внезапного краха системы после его повторного ввода в строй клиент увидит результат транзакции.

CAS (compare-and-swap) – сравнение и обмен, разновидность атомарной операции. Операция состоит в том, чтобы сравнить содержимое некоторого адреса в памяти с заданным значением и, если они совпадают, поместить по этому адресу новое значение. В стандартной библиотеке языка C++ эта операция представлена в двух разновидностях, сильной и слабой, функциями `std::compare_exchange_strong` и `std::compare_exchange_weak`.

RAII (Resource Acquisition Is Initialization, захват ресурса есть инициализация) –широко распространённая идиома программирования на языке C++, в которой захват ресурса и его освобождение оформлены как, соответственно, создание объекта и его уничтожение. Например, это означает захват мьютекса в конструкторе объекта-блокировщика и освобождение в деструкторе. Такой способ управления мьютексом называют ещё локальной блокировкой.

Самыми известными примерами использования идиомы RAII в стандартной библиотеке языка C++ являются блокировщики, управляющие удержанием мьютексов, умные указатели, управляющие временем жизни завёрнутых

в них объектов, а также контейнеры, управляющие временем жизни своих элементов[1].

Volatile – спецификатор типа данных, означающий, что объект может измениться независимо от нормального хода выполнения программы. Например, это могут быть ячейки памяти, связанные с физическим устройством, что особенно актуально при программировании встроенных устройств, или области, используемые при прямом отображении ввода-вывода на память. Работа с такими данными требует их чтения всякий раз непосредственно из устройства памяти и записи также непосредственно в память, минуя любые механизмы буферизации.

Асинхронность – способность системы выполнять несколько заданий, перекрывающихся во времени. Обобщение (истинной) параллельности.

Вызываемый объект (callable unit) – это сущность, которая ведёт себя подобно функции. К ним относятся и собственно функции, и функциональные объекты, и лямбда-выражения. Если вызываемый объект требует на вход одного аргумента, его называют унарным, при двух аргументах – бинарным. Предикат – частный случай вызываемого объекта с логическим типом возвращаемого значения.

Жадная стратегия вычислений – стратегия, при которой выражение вычисляется немедленно. Противоположная стратегия, называемая ленивой, состоит в том, чтобы вычислять значение выражения лишь тогда, когда это необходимо для дальнейшей работы алгоритма.

Исполнитель – объект, связанный с определённым контекстом выполнения. Он поддерживает одну или более функций для создания агентов выполнения для заданного вызываемого или функционального объекта.

Истинная параллельность – способность системы выполнять несколько операций одновременно. Это более узкое понятие, чем асинхронность, при которой может создаваться иллюзия параллельного выполнения разных потоков за счёт чередования их операций.

Критическая секция – участок кода, который не должен выполняться более, чем одним потоком.

Ленивая стратегия вычислений – стратегия вычислений[2], при которой значение выражения вычисляется только тогда, когда оно необходимо. Это противоположно жадной стратегии. Ленивые вычисления также называют вычислениями по необходимости.

Ложное пробуждение – ошибочная ситуация, когда потоку, ожидающему переменной условия, приходит оповещение, которого на самом деле ему никакой поток не посылал.

Лямбда-функции – это удобный механизм определять функцию (или функциональный объект) по месту использования. Поскольку компилятор видит определение функции непосредственно в том контексте, где она вызывается, он получает богатые возможности для оптимизации. Лямбда-функции могут получать аргументы по значению или по ссылке. Также они могут захватывать (т. е. делать частью своего состояния) переменные из контекста по ссылке и по значению. Ниже показан пример.

[1] https://en.cppreference.com/w/cpp/container.

[2] https://ru.wikipedia.org/wiki/Ленивые_вычисления.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::for_each(myVec.begin(), myVec.end(), [](int& i){ i= i*i; });
    // 1 4 9 16 25 36 49 64 81 100
```

Лямбда-функции стоит использовать почаще

Если возникает необходимость определить в своей программе вызываемый объект с коротким и очевидным кодом, следует оформить его в виде лямбда-функции. Лямбда-функция часто оказывается быстрее обычной функции или явно определённого функционального объекта и обычно проще для понимания.

Мёртвая блокировка – состояние, при котором по меньшей мере один поток заблокирован навсегда, поскольку он ожидает ресурса, который гарантированно никогда не освободится. Основных причин возникновения мёртвой блокировки две:

- поток захватывает, но никогда не освобождает мьютекс;
- два потока захватывают мьютексы в различном порядке.

Модель памяти – совокупность правил, определяющих отношения между объектами данных, областями памяти и операциями, выполняемыми над ними из различных потоков. Главный вопрос, на который отвечает модель памяти: что происходит, когда два потока одновременно осуществляют доступ к одной и той же области памяти.

Монада. Понятие, заимствованное современным программированием из теории категорий. Язык Haskell относится к чистым функциональным, все функции в нём – чистые. Главное свойство таких функций состоит в том, что они всегда возвращают одинаковый результат, когда их вызывают с одинаковыми аргументами. Вследствие этого свойства, также называемого ссылочной (также референциальной) прозрачностью[1], функции в языке Haskell не могут иметь побочных эффектов. Это создаёт концептуальное затруднение. Взаимодействие программы с внешним миром невозможно без побочных эффектов. Операции ввода-вывода могут завершаться неудачей, возвращать разное количество данных, а также зависеть от не поддающихся учёту факторов. Для преодоления этой концептуальной сложности в чистом функциональном языке используется понятие монады[2]. Классические монады из стандартной библиотеки языка Haskell включают:

---

[1] https://ru.wikipedia.org/wiki/Ссылочная_прозрачность.

[2] Сводить значение монад в программировании к внедрению побочных эффектов в чистый функциональный язык – слишком узкая трактовка. В чистых функциональных языках наподобие Haskell применения монад для чистых вычислений не менее разнообразны и интересны, чем для ввода-вывода и изменяемых состояний. Ели оставить в стороне чисто математическую сторону дела, то с точки зрения программирования монаду можно считать контейнерным типом, который умеет к каждому содержащемуся в нём элементу применить функцию-преобразователь, которая возвращает свой результат завёрнутым в такой же контейнер, а затем из полученных при этом контейнеров собрать новый контейнер – итоговый результат. Тем самым монада позволяет связывать между собой в цепочку вычисления, имеющие своими результатами контейнеры. Существенно, что, в отличие от контейнеров из стандартной библиотеки C++, здесь термин «контейнер» понимается предельно широко. Так, функцию можно считать контейнером вычисляемых ею результатов, асинхронное задание – контейнером своего будущего результата. Более подробную информацию можно найти в статьях https://habr.com/ru/post/125782 и https://habr.com/ru/post/445488. – *Прим. перев.*

- **IO** – средство ввода-вывода;
- **Maybe** – вычисления, которые могут завершиться как результатом, так и его отсутствием;
- **Error** – вычисления, которые могут завершиться результатом или ошибкой;
- **List** – структура данных «список» хорошо подходит для моделирования вычислений, которые могут дать множество возможных результатов;
- **State** – вычисления со вспомогательным изменяемым состоянием;
- **Reader** – вычисления, использующие некоторые глобальные параметры.

Понятие монады импортировано в программирование из теории категорий – области математики, в которой изучаются связи и отношения между объектами в полном отвлечении от природы этих объектов. Монаду можно считать семейством абстрактных типов данных, преобразующим простые значения в значения, обогащённые некоторым контекстом. Значения таких обогащённых типов называются монадическими значениями. Когда простое значение оказывается погружённым в монаду, его можно далее преобразовывать композицией операций, преобразующих простое значение в монадическое.

Композиция операций над монадами сохраняет её структуру. Например, монада Error прерывает всю цепочку вычислений, если ошибка возникает в каком-либо её звене; монада State позволяет строить цепочку вычислений, каждое звено которых читает из одного и того же состояния и модифицирует его.

Монада состоит из трёх частей:

- конструктор типа определяет, как значения простого типа становятся монадическими значениями;
- операции:
  - тождественная – превращает одно простое значение в монаду, содержащее только это значение;
  - связывание – применяет преобразователь завёрнутых в монаду простых значений в новые монады и объединяет их в новую монаду – итоговый результат;
- законы монад:
  - тождественная функция должна быть левой и правой единицей операции связывания;
  - композиция операций над монадами должна быть ассоциативной.

Например, чтобы тип конструктор типов Error сделать экземпляром класса типов Monad, он должен поддерживать тождественную функцию, которая превращает простое значение некоторого типа в монадическое значение, соответствующее безошибочному вычислению; кроме того, он должен поддерживать операцию монадического связывания, которая передаёт ошибку (если она возникла) без изменений далее по цепочке вычислений. Таким образом, обе функции определяют способ обработки ошибок. Если цепочка вычислений оформлена в монаду Error, обработка ошибок будет делаться неявным образом и автоматически.

Монадическое вычисление состоит из двух потоков передачи данных: явного, по которому передаются заключённые в монаде значения, и неявного, по которому передаются сопутствующие побочные эффекты.

Конечно, можно дать и другое определение монады: это просто-напросто моноид в категории эндофункторов.

Монады приобретают всё большее значение в современном программировании и, в частности, находят своё отражение в языке C++. Так, в стандарт C++ 17 добавлен шаблон `std::optional`[1], близкий аналог монады Maybe, в стандарте C++ 20 появились диапазоны[2] (ranges), а в стандарте C++ 23 ожидаются расширенные фьючерсы – ещё два примера монад.

Неблокирующие алгоритмы. Алгоритм называется неблокирующим, если крах или приостановка любого потока системы не может вызвать крах или остановку любого другого потока. Это определение взято из превосходной книги «Java Concurrency in Practice»[3].

Неблокирующий алгоритм – алгоритм, в котором нет блокировок и гарантируется продвижение вычислений вперёд в рамках системы в целом.

Независимость от адреса. Атомарные операции должны быть не только свободны от блокировок (lock-free), но и независимы от адресации (address-free). Это означает, что операции над одной и той же атомарной переменной должны оставаться атомарными, даже если их выполняют различные процессы.

Неожидающий алгоритм – алгоритм, гарантирующий продвижение к цели отдельно для каждого потока.

Неопределённое поведение – ситуация, в которой стандарт языка C++ устанавливает, что невозможно делать какие бы то ни было предположения о дальнейшем поведении программы. Программа, вошедшая в состояние неопределённого поведения, может дать корректный результат, ошибочный результат, потерпеть крах, может даже не откомпилироваться. Поведение такой программы имеет право меняться при переносе кода на другую платформу, при смене компилятора или в результате изменения кода, не имеющего отношения к данному месту.

О-большое – мера сложности алгоритма, т. е. времени (измеряемого количеством операций), необходимого для его завершения в зависимости от размера входных данных n. Так, сложность $O(1)$ означает, что время выполнения операции над контейнером постоянно и не зависит от числа содержащихся в нём элементов. Соответственно, $O(n)$ значит, что время работы алгоритма линейно зависит от размера контейнера. Также часто встречаются алгоритмы со сложностью $O(\log n)$, $O(n^2)$ и др.

Область памяти, согласно справочнику cppreference.com[4], – это любое из следующего:

- объект скалярного типа (арифметического типа, типа указателя, перечисления, а также типа `std::nullptr_t`);
- наиболее длинная непрерывная последовательность битовых полей ненулевой длины.

---

1 http://en.cppreference.com/w/cpp/utility/optional.

2 https://en.cppreference.com/w/cpp/ranges.

3 http://jcip.net. Есть русский перевод: Java Concurrency на практике / Гетц Б. и др. СПб.: Питер, 2020. 464 с.

4 http://en.cppreference.com/w/cpp/language/memory_model.

Полный порядок – это бинарное отношение ⩽ на некотором множестве X, рефлексивное, транзитивное, антисимметричное и тотальное:

- рефлексивность: $a \le a$;
- транзитивность: из $a \le b$ и $b \le c$ следует $a \le c$;
- антисимметричность: из $a \le b$ и $b \le a$ следует $a = b$;
- тотальность: либо $a \le b$, либо $b \le a$,

– при любых a, b, c из множества X.

В применении к параллельному программированию понятие полного порядка становится особенно полезным. Так, операции над атомарной переменной или транзакции над памятью образуют вполне упорядоченную последовательность. Это означает, что все потоки видят результаты этих операций в одном и том же порядке.

Порядок модификации. Все модифицирующие операции над атомарным объектом M образуют некоторую вполне упорядоченную последовательность. Она называется порядком модификации объекта M. Как следствие операции чтения атомарного объекта некоторым потоком никогда не могут увидеть значения, более старые, чем уже прочитанные.

Последовательная согласованность характеризуется двумя основными свойствами:

- инструкции программы выполняются в том порядке, в котором они записаны в её тексте;
- существует глобальный порядок операций для всех потоков программы.

Последовательность освобождения над атомарным объектом M, начинающаяся с операцией освобождения A, – это максимальная непрерывная последовательность побочных эффектов в порядке модификации объекта M, в которой первой стоит операция A, и каждая последующая операция * либо выполняется тем же потоком, который выполнил операцию A, либо * представляет собой атомарную операцию чтения-модификации-записи.

Поток выполнения – наименьшая единица в составе программы, состоящая из последовательности команд, выполнением которой планировщик может управлять независимо. Поток обычно находится под управлением операционной системы. Детали реализации потоков и процессов отличаются в разных операционных системах, но в большинстве случаев поток является составной частью процесса. В состав одного процесса может входить множество потоков, которые выполняются параллельно и совместно используют ресурсы – в первую очередь память. Потоки в пределах процесса могут выполнять один и тот же код и обращаться к одним и тем же переменным. Процессы, в отличие от потоков, не имеют общих ресурсов.

Предикат – вызываемый объект, возвращающий значение логического типа. Если он при этом обладает одним аргументом, его называют унарным предикатом, если двумя аргументами – бинарным.

Свойства алгебраических операций. Бинарная операция * на некотором множестве X называется

- **коммутативной**, если $x * y = y * x$ для любых x и y из множества X;
- **ассоциативной**, если $(x * y) * z = x * (y * z)$ для любых x, y и z из множества X.

Точка последовательности – точка в программе, в которой гарантируется, что эффекты всех предыдущих операций уже зафиксированы в памяти, а результаты ни одной последующей операции ещё в памяти не присутствуют.

Тривиально копируемые объекты – это объекты, для копирования которых достаточно скопировать содержимое занимаемой ими памяти (например, функцией `std::memmove`). Все типы данных, выразимые в языке C (так называемые POD – plain old data), тривиально копируемы. Более строго, именованное требование TriviallyCopyable включает в себя следующие условия:

- все конструкторы копирования тривиальны или удалены;
- все конструкторы перемещения тривиальны или удалены;
- все копирующие операции присваивания тривиальны или удалены;
- все перемещающие операции присваивания тривиальны или удалены;
- по меньшей мере один конструктор копирования, конструктор перемещения, копирующая операция присваивания или перемещающая операция присваивания не удалена;
- имеется тривиальный деструктор.

Эти условия, в частности, означают, что у класса отсутствуют виртуальные функции и виртуальные базовые классы. Под тривиальностью перечисленных выше функций-членов имеется в виду главным образом, что у них нет реализации, описанной программистом в явном виде. Эти же условия должны соблюдаться для всех базовых классов и нестатических данных-членов.

Утеря пробуждения – ситуация, при которой поток безвозвратно упускает оповещение, отправленное через переменную условия. Возникает, когда поток-отправитель посылает оповещение раньше, чем поток-получатель начинает его ожидать, и при этом получатель не использует для ожидания предикат.

Функциональный объект. В первую очередь подчеркнём, что их нельзя называть *функторами*: последний термин имеет строго определённое значение в теории категорий[1] – области математики, нашедшей применение в современном программировании. Функциональные объекты – это объекты, способные вести себя подобно функциям благодаря наличию перегруженной операции вызова. Поскольку во всём остальном это обычные объекты, они

[1] Категория состоит из *объектов* и *стрелок* между ними (также называемых *морфизмами*), причём для каждого объекта имеется особая стрелка id из этого объекта в него же, называемая *тождественной*, и определена операция композиции морфизмов: если f – стрелка из объекта A в объект B, а g – стрелка из B в C, то единственным образом определена их композиция $g \circ f$ (читается «g после f») – стрелка из A в C, причём тождественная стрелка является единицей операции композиции. Самая типичная для программирования – категория, в которой объекты суть типы данных, а стрелки – функции из типа аргумента в тип значения. Функтор – это отображение категорий, сохраняющее тождественные стрелки и композицию стрелок. С точки зрения программирования функтор выступает как контейнер, умеющий применять унарную функцию-преобразователь к каждому своему элементу и из полученных результатов формировать контейнер такой же структуры. Иными словами, функтор позволяет функцию над отдельными значениями поднять до функции над контейнерами. – *Прим. перев.*

могут обладать членами-данными, т. е. состоянием. Пример функционального объекта без состояния приведён ниже.

```
struct Square{
  void operator()(int& i){i= i*i;}
};

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::for_each(myVec.begin(), myVec.end(), Square());

for (auto v: myVec)
  std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

Необходимость создания функциональных объектов

Частая ошибка состоит в том, что имя функционального типа используют в алгоритме в качестве имени функции, тогда как использовать нужно объект (экземпляр) этого типа. Конечно же, компилятор этого не позволяет. Из следующих двух строк первая ошибочна, а вторая – правильна:

```
std::for_each(myVec.begin(), myVec.end(), Square)
std::for_each(myVec.begin(), myVec.end(), Square())
```

Шаблон проектирования «представляет собой трёхстороннее правило, выражающее отношение между определённым контекстом, проблемой и решением», как писал Кристофер Александер в основополагающей книге.

Предметный указатель

A

ABA, 552
abi_for_size, 409
accumulate, 233, 276
ACI(D), 396
ACID, 598
acquire (std::counting_semaphore), 179
adopt_lock, 150
all_of (std::simd), 408
any_of (std::simd), 408
arrive_and_drop (std::barrier), 188
arrive_and_wait (std::barrier), 188
arrive_and_wait (std::latch), 182
arrive (std::barrier), 188
asctime, 572
async, 192
atomic, 54
atomic<bool>, 55
atomic<std::shared_ptr>, 64
atomic<std::weak_ptr>, 64
atomic_bool, 70
atomic_cancel, 400
atomic_char, 70
atomic_char8_t, 70
atomic_char16_t, 70
atomic_char32_t, 70
atomic_clear, 73
atomic_clear_explicit, 73
atomic_commit, 400
atomic_flag_test_and_set, 73
atomic_flag_test_set_explicit, 73
atomic_int, 70
atomic_int8_t, 70
atomic_int16_t, 70
atomic_int32_t, 70
atomic_int64_t, 70
atomic_int_fast8_t, 70
atomic_int_fast16_t, 71
atomic_int_fast32_t, 71
atomic_int_fast64_t, 71
atomic_int_least8_t, 70
atomic_int_least16_t, 70
atomic_int_least32_t, 70
atomic_int_least64_t, 70
atomic_intmax_t, 71
atomic_intptr_t, 71
atomic_llong, 70
atomic_long, 70
atomic_noexcept, 400
atomic_ptrdiff_t, 71
atomic_ref, 76
atomic_schar, 70
atomic_shared_ptr, 64
atomic_short, 70
atomic_signal_fence, 116
atomic_signed_lock_free, 71
atomic_size_t, 71
atomic_thread_fence, 106
atomic_uchar, 70
atomic_uint, 70
atomic_uint8_t, 70
atomic_uint16_t, 70
atomic_uint32_t, 70
atomic_uint64_t, 70
atomic_uint_fast8_t, 71
atomic_uint_fast16_t, 71
atomic_uint_fast32_t, 71
atomic_uint_fast64_t, 71
atomic_uint_least8_t, 70

atomic_uint_least16_t, 70
atomic_uint_least32_t, 70
atomic_uint_least64_t, 70
atomic_uintmax_t, 71
atomic_uintptr_t, 71
atomic_ullong, 70
atomic_ulong, 70
atomic_unsigned_lock_free, 71
atomic_ushort, 70
atomic_wchar_t, 70
atomic_weak_ptr, 64
await_ready, 255
await_resume, 255
await_suspend, 255

B

back (операция над очередью), 535
barrier, 187
basic_osyncstream, 220
basic_streambuf, 220
basic_syncbuf, 220
binary_semaphore, 179
broken_promise, 213
bulk_execute (функция исполнителя), 381
bulk_then_execute (функция исполнителя), 381
bulk_twoway_execute (функция исполнителя), 381

C

call_once, 152
CAS, 60, 598
chrono, 571
clamp (std::simd), 407
clear (операция над атомарным типом), 71
clear (std::atomic_flag), 47
co_await, 266
compare and swap, 60
compare-and-swap, 598
compare_exchange_strong, 60
compare_exchange_strong (операция над атомарным типом), 71
compare_exchange_strong (std::atomic_ref), 82
compare_exchange_weak, 60
compare_exchange_weak (операция над атомарным типом), 71
compare_exchange_weak (std::atomic_ref), 82
compatible, 406
concat, 407
condition_variable, 160
condition_variable_any, 163
const_where_expression, 407
ContinuableFuture, 394
co_return, 261
count_down (std::latch), 182
co_yield, 263
CRTP, 463
current_exception, 213

D

defer_lock, 150
deferred (состояние фьючерса), 206
define_task_block, 402
define_task_block_restore_thread, 402
detach (std::thread), 120, 126
duration, 575

E

element_aligned, 407
emit (std::basic_osyncstream), 222
epoch (момент времени), 571
exchange (операция над атомарным типом), 71
exchange (std::atomic_ref), 82
exclusive_scan, 232
execute (функция исполнителя), 381

F

fetch_add (операция над атомарным типом), 71
fetch_add (std::atomic_ref), 82
fetch_and (операция над атомарным типом), 72
fetch_and (std::atomic_ref), 82
fetch_or (операция над атомарным типом), 72

fetch_or (std::atomic_ref), 82
fetch_sub (операция над атомарным типом), 71
fetch_sub (std::atomic_ref), 82
fetch_xor (операция над атомарным типом), 72
fetch_xor (std::atomic_ref), 82
FIFO, 535
final_suspend (использование в сопрограмме), 256
final_suspend noexcept (объект-обещание), 252
find_first_set (std::simd), 408
find_last_set (std::simd), 408
fixed_size, 406
foldl1 (Haskell), 238
foldl (Haskell), 238
for_each, 232
for_each_n, 232
front (операция над очередью), 535
future, 203, 205
FutureContinuation, 394
future_errc, 213
future_error, 213
future_status, 206

G

get_future (std::packaged_task), 201
get_future (std::promise), 205
get_id (std::thread), 126
get_return_object (объект-обещание), 252
get (std::future), 205
get_stop_source (std::jthread), 129
get_stop_token (std::jthread), 129
get_token (std::stop_source), 167
get_wrapped (std::basic_osyncstream), 222
gmtime, 572

H

h (литералы времени), 577
hardware_concurrency (std::thread), 126
hardware_constructive_interference_size, 566
hardware_destructive_interference_size, 566
High-Performance ParallelX, 240
high_resolution_clock, 579
hmax (std::simd), 408
hmin (std::simd), 408
hours, 575
HPX, 240

I

inclusive_scan, 232
initial_suspend, 256
initial_suspend (объект-обещание), 252
is_abi_tag, 408
is_always_lock_free (операция над атомарным типом), 71
is_always_lock_free (std::atomic_ref), 82
is_execution_policy, 227
is_lock_free (операция над атомарным типом), 71
is_lock_free (std::atomic_ref), 82
is_simd, 408
is_simd_flag_type, 408
is_simd_mask, 408

J

joinable (std::thread), 126
join (std::thread), 120, 126

L

latch, 182
LIFO, 528
load, 82
load (операция над атомарным типом), 71
LoadLoad, 107
LoadStore, 107
lock (функция), 148
lock_guard, 141
lock (std::unique_lock), 143

M

make_exception_ptr, 213
make_execptional_future, 388

make_ready_at_thread_exit (std::packaged_task), 201
make_ready_future, 388
map (Haskell), 238
max (момент времени), 571
max_fixed_size, 406
max (std::barrier), 188
max (std::counting_semaphore), 179
max (std::latch), 182
max (std::simd), 407
memory_alignment, 409
memory_order_acq_rel, 84
memory_order_acquire, 84
memory_order_consume, 84
memory_order_relaxed, 84
memory_order_release, 84
memory_order_seq_cst, 84
microseconds, 575
milliseconds, 575
min (литералы времени), 577
min (момент времени), 571
minmax (std::simd), 407
min (std::simd), 407
minutes, 575
ms (литералы времени), 577
mutex, 134
mutex (std::unique_lock), 143

N

nanoseconds, 575
native, 406
native_handle (std::condition_variable), 161
none_of (std::simd), 408
nostopstate_t, 167
notify_all (операция над атомарным типом), 72
notify_all (std::atomic_flag), 47
notify_all (std::atomic_ref), 82
notify_all (std::condition_variable), 161
notify_one (операция над атомарным типом), 72
notify_one (std::atomic_flag), 47
notify_one (std::atomic_ref), 82
notify_one (std::condition_variable), 161
now (момент времени), 571
ns (литералы времени), 577

O

once_flag, 152
osyncstream, 220
overaligned, 407
owns_lock (std::unique_lock), 143

P

packaged_task, 198
parallel_policy, 226
parallel_unsequenced_policy, 226
par (std::execution), 226
partial_sum, 233
par_unseq (std::execution), 226
pop (операция над очередью), 535
pop (операция над стеком), 529
popcount (std::simd), 408
promise, 203
Promise, 394
push (операция над очередью), 535
push (операция над стеком), 529

R

RAII, 598
ratio, 575
RCU, 556
ready (состояние фьючерса), 206
recursive_ mutex, 136
recursive_timed_mutex, 136
reduce, 232
reduce (std::simd), 408
release (std::counting_semaphore), 179
release (std::unique_lock), 143
request_stop (std::jthread), 129
request_stop (std::stop_source), 168
require (execution), 383
reset (std::packaged_task), 201
return_value (объект-обещание), 252
return_void (объект-обещание), 252

S

s (литералы времени), 577

scalar, 406
scanl1 (Haskell), 238
scanl (Haskell), 238
scoped_lock, 142
scoped_thread, 121
seconds, 575
SemiFuture, 394
seq (std::execution), 226
sequenced_policy, 226
set_exception_at_thread_exit (std::promise), 205
set_exception (std::promise), 205
set_value_at_thread_exit (std::promise), 205
set_value (std::promise), 205
shared_future, 207
SharedFuture, 394
shared_lock, 144
shared_mutex, 137
shared_ptr, 74
shared_timed_mutex, 137
share (std::future), 205
signal, 116
SIGTERM, 116
simd, 406
simd_cast, 407
simd_mask, 406
simd_size, 409
sleep_for, 585
sleep_for (std::this_thread), 126
sleep_until, 585
sleep_until (std::this_thread), 126
some_of (std::simd), 408
spinlock, 48
split, 407
static_simd_cast, 407
steady_clock, 579
stop_callback, 169
stop_possible (std::stop_source), 167
stop_possible (std::stop_token), 168
stop_requested (std::stop_source), 168
stop_requested (std::stop_token), 168
stop_source, 167
stop_token, 168
store (операция над атомарным типом), 71
StoreLoad, 107
store (std::atomic_ref), 82
StoreStore, 107
suspend_always, 255
suspend_never, 255
swap (std::packaged_task), 201
swap (std::unique_lock), 143
system_clock, 579

T

task_cancelled_exception, 404
TBB, 240
test_and_set (операция над атомарным типом), 71
test_and_set (std::atomic_flag), 47
test (std::atomic_flag), 47
then_execute (функция исполнителя), 381
thread, 175
Threading Building Blocks, 240
timed_mutex, 136
timeout (состояние фьючерса), 206
time_point, 571
time_since_epoch, 582
time_t, 572
to_compatible, 407
to_fixed_size, 407
to_native, 407
top (операция над стеком), 529
to_time_t, 572
transaction_safe, 401
transaction_unsafe, 401
transform_exclusive_scan, 232
transform_inclusive_scan, 232
transform_reduce, 232
try_acquire_for (std::counting_semaphore), 180
try_acquire (std::counting_semaphore), 179
try_acquire_until (std::counting_semaphore), 180
try_lock_for, 585
try_lock_for (std::unique_lock), 143
try_lock (std::unique_lock), 143
try_lock_until, 585

try_lock_until (std::unique_lock), 143
try_wait (std::latch), 182
twoway_execute (функция исполнителя), 381

U

unhandled_exception (объект-обещание), 252
unlock (std::unique_lock), 143
us (литералы времени), 577

V

valid (std::future), 205
valid (std::packaged_task), 201
vector_aligned, 407
volatile, 599

W

wait (операция над атомарным типом), 72
wait_for, 585
wait_for (std::condition_variable), 161
wait_for (std::future), 205
wait (std::atomic_flag), 47
wait (std::atomic_ref), 82
wait (std::barrier), 188
wait (std::condition_variable), 161
wait (std::future), 205
wait (std::latch), 182
wait_until, 585
wait_until (std::condition_variable), 161
wait_until (std::future), 205
where, 407
where_expression, 407
wosyncstream, 220

Y

yield (std::this_thread), 126
yield_value (объект-обещание), 252

А

Автоматическое присоединение к потоку, 129
Агент выполнения, 381
Активное ожидание, 52
Активный объект, 448
Анализ программы с блокировкой (CppMem), 320
Антишаблоны, 413
Асинхронность, 599
Асинхронные задания, 190
Ассоциативность, 603
Атомарность (свойство транзакции), 396
Атомарные обёртки для пользовательских типов, 62
Атомарные операции над типом std::shared_ptr, 75
Атомарные переменные, 43
  с ослабленной семантикой (CppMem), 333
  с последовательной согласованностью (CppMem), 321
  с семантикой захвата и освобождения (CppMem), 327
Атомарные типы с плавающей точкой, 66
Атомарные умные указатели, 63
Атомарные целочисленные типы, 67
Атомарный флаг, 47

Б

Барьер
  захвата, 108
  освобождения, 108
  памяти, 106
Барьеры, 106
Барьеры захвата и освобождения, 109
Бесконечный поток данных, 263
Библиотека для векторных вычислений, 405
Блоки заданий, 401
Блокировка с двойной проверкой, 300
Блокировщики, 141
Быстрая синхронизация потоков, 335

В

Векторные типы данных, 406

Виды атомарных операций, 84
Возобновляемые функции, 250
Временная сложность алгоритма, 602
Время жизни потоков, 119
Встраиваемые статические члены-данные, 438
Вызываемый объект, 599
Выполняющий ресурс, 380
Выход за пределы допустимого диапазона часов, 573
Вычисление суммы элементов вектора, 274
Вычисления с промежутками времени, 577

Г

Гонка данных, 560
Гранулярность интерфейса, 508

Д

Данные в совместном доступе, 133
Двоичный семафор, 179
Дейкстра, Эдсгер Вибе, 178
Дескриптор сопрограммы, 252
Диспетчер (шаблон проектирования), 466
Дополнительные сведения, 551

Ж

Жадная стратегия вычислений, 599
Жадный фьючерс, 261

З

Заместитель (шаблон), 450
Запустить и забыть, 195
Затруднения с мьютексами, 138
Захват семафора, 178
Защёлки и барьеры, 182

И

Избегание прорех, 517
Изолированность атомарности (свойство транзакции), 396
Инварианты (в потокобезопасных структурах данных), 525
Исключения (в потокобезопасных структурах данных), 528
Исполнители, 374
Исполнитель, 381, 599
Использование фьючерсов, 422
Истинная параллельность, 599

К

Кадр сопрограммы, 254
Коммутативность, 603
Конкуренция потоков, 520
Константные выражения, 151
Контекст выполнения, 380
Контракт (модель памяти), 40
Контроллер ожидания, 255
Кооперативное прерывание потоков, 166
Копирование значения (шаблон проектирования), 416
Краткий обзор, 22
Критическая секция, 599

Л

Ленивая стратегия вычислений, 599
Ложное пробуждение, 165, 599
Локальные блокировщики, 424
Локальные статические переменные, 156
Лямбда-функция, 599

М

Масштабируемость (потокобезопасных структур данных), 523
Мейерс (реализация объекта-одиночки), 304
Мёртвая блокировка, 561, 600
Механизм чтения-копирования-модификации, 556
Многопоточное суммирование с общей переменной, 281
Модель памяти, 38, 600
  эмпирические правила, 500

Модификации и обобщения генераторов, 355
Моменты времени, 571
Монада, 600
Монитор (шаблон), 457
Мьютекс, 134

Н

Нарушение инварианта программы, 558
Нахождение точки отсчёта часов, 582
Начальное представление о модели памяти, 38
Неатомарные переменные (CppMem), 314
Неблокирующий алгоритм, 602
Независимость от адреса, 602
Независимость от адресации, 48
Неожидающий алгоритм, 602
Неопределённое поведение, 602
Неявные связи между данными, 563

О

Обзор инструментального средства CppMem, 591
Область памяти, 602
О-большое, 602
Общие правила, 473
Общие соображения (потокобезопасные структуры данных), 505
Объект-заглушка, 431
Объект-монитор, 457
Объект-обещание, 252
Объект-сопрограмма, 251
Объекты-значения, 417
Ограничения на синхронизацию и порядок доступа, 85
Ограничения (сопрограмм), 251
Одиночка (шаблон), 299
Опасный указатель, 555
Оповещатель (шаблон проектирования), 466
Оповещения, 214
Ослабленный блок, 397
Особенности класса std::jthread, 175
Отличие заданий от потоков, 191
Охраняемая приостановка, 440
Очередь, 535

П

Параллельная архитектура, 447
Параллельное вычисление скалярного произведения, 196
Параллельные алгоритмы в стандартной библиотеке, 225
Параметризованные блокировщики, 426
Перевод моментов времени в календарный формат, 572
Передача аргументов при создании потока, 122
Передача исполнителей по цепочке, 392
Переменные условия, 160
Перемещение потоков, 567
Перенос зависимости, 103
Политики выполнения, 226
Полный барьер, 107, 108
Полный порядок, 603
Полный список атомарных операций, 82
Полусинхронная архитектура, 464
Помеченное состояние, 555
Порядок изменения, 103
Порядок модификации, 603
Последовательная согласованность, 603
Последовательно-согласованное выполнение, 86
Последовательность освобождения, 603
Поток выполнения, 603
Потокобезопасная инициализация, 151
Потокобезопасная очередь, 535
Потокобезопасное создание объекта-одиночки, 299
Потокобезопасный односвязный список, 64

Потокобезопасный стек, 528
Поэтапная оптимизация, 312
Пояснение некоторых терминов, 598
Предикат, 603
Преобразование функции в сопрограмму, 250
Проактор (шаблон), 469
Проблема ABA, 552
Проблемы со временем жизни объектов, 566
Прогнозирование ветвлений, 40
Продолжение фьючерса, 387
Производительность стандартных параллельных алгоритмов, 241
Промежутки времени, 575
Прообраз ожидания, 254
Простая реализация очереди, 539

Р

Разветвление и слияние, 402
Расширенные фьючерсы, 386
Реактор (шаблон), 466

С

Свободные функции над атомарными типами, 73
Сводные данные по алгоритмам с общей переменной, 287
Сводные данные по однопоточным алгоритмам, 280
Свойства алгебраических операций, 603
Семантика захвата и освобождения, 88
Семафоры, 178
Сильная модель памяти, 44
Синхронизированные потоки вывода, 216
Слабая модель памяти, 46
Сложности параллельного программирования, 552
Смесь атомарных и неатомарных переменных (CppMem), 331
Согласованность атомарности (свойство транзакции), 396
Соглашения об именовании, 584
Создание новых фьючерсов, 388
Создание потока, 118
Сопрограммы, 245
Состояние гонки, 569
Состояние сопрограммы, 252
Состояние фьючерса, 206
Спин-блокировкой, 48
Список опасных указателей, 555
Способы управления заданиями, 364
Сравнение и обмен, 60, 598
Сравнение циклической блокировки с мьютексом, 50
Ссылка на помеченное состояние, 555
Стек, 528
Стратегии блокировки, 506
Стратегии ожидания, 585
Структуры данных, 504
Считающий семафор, 178

Т

Типы часов, 579
Тип std::future, 205
Тонкости блокировок, 556
Точка последовательности, 604
Точность и монотонность часов, 579
Транзакционно-безопасная функция, 401
Транзакционно-небезопасная функция, 401
Транзитивность (порядка доступа к памяти), 90
Требования к сопрограммам, 250
Тривиально копируемый объект, 604

У

Унифицированные фьючерсы, 391
Управление изменяемым состоянием, 423
Управление общим доступом, 415
Управление потоками, 117
Устойчивость атомарности (свойство транзакции), 396
Утерянное пробуждение, 164

Утерянные и ложные пробуждения, 164
Утеря пробуждения, 604
Учебные примеры, 274

Ф

Фабрика сопрограмм, 251
Фундаментальный атомарный интерфейс, 55
Функции выполнения, 381
Функции-члены атомарных типов, 71
Функциональный объект, 604
Функция-генератор, 247

Ц

Циклическая блокировка, 48

Ш

Шаблон
  проектирования, 410, 605
  «Реактор», 466
  синхронизации, 415

Э

Эмпирические правила, 473

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Райнер Гримм

Параллельное программирование на современном языке C++

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Винник В. Ю.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 50,05. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«В книге излагаются как простые, так и углубленные аспекты параллельного программирования. В ней вы найдете все, что нужно специалисту в этой области».

Роберт Бадеа, технический руководитель команды

Книга во всех подробностях освещает параллельное программирование на современном C++. Особое внимание уделено опасностям и трудностям параллельного программирования (например, гонке данных и мертвой блокировке) и способам борьбы с ними. Приводятся многочисленные примеры кода, позволяющие читателю легко закрепить теорию на практике.

Среди рассматриваемых тем:

- атомарные переменные и барьеры памяти;
- синхронизация по наступлению событий на основе переменных условий;
- асинхронные задания – удобная альтернатива ручному управлению потоками;
- считающие и двоичные семафоры;
- сопрограммы в стандарте C++ 20;
- новшества, запланированные в стандарт C++ 23.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

www.дмк.рф