Параллельное программирование на C++ в действии

Энтони Уильямс

Практика разработки многопоточных программ

MANNING

УДК 004.438С++11
ББК 32.973.26-018.2
У36

У36 Энтони Уильямс

Параллельное программирование на С++ в действии. Практика разработки многопоточных программ. Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2012. – 672с.: ил.

ISBN 978-5-94074-448-1

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

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

УДК 004.438С++11
ББК 32.973.26-018.2

Original English language edition published by Manning Publications Co., Rights and Contracts Special Sales Department, 20 Baldwin Road, PO Box 261, Shelter Island, NY 11964. ©2012 by Manning Publications Co.. Russian-language edition copyright © 2012 by DMK Press. All rights reserved.

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.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-93398-877-1 (англ.)
ISBN 978-5-94074-448-1

©2012 by Manning Publications Co.
© Оформление, перевод на русский язык
ДМК Пресс, 2012

ОГЛАВЛЕНИЕ

Предисловие 13
Благодарности 15
Об этой книге 17
Об иллюстрации на обложке 21
ГЛАВА 1. Здравствуй, параллельный мир! 22
1.1. Что такое параллелизм? 23
1.1.1. Параллелизм в вычислительных системах 23
1.1.2. Подходы к организации параллелизма 26
1.2. Зачем нужен параллелизм? 29
1.2.1. Применение параллелизма для разделения обязанностей 29
1.2.2. Применение параллелизма для повышения производительности 30
1.2.3. Когда параллелизм вреден? 32
1.3. Параллелизм и многопоточность в C++ 33
1.3.1. История многопоточности в C++ 34
1.3.2. Поддержка параллелизма в новом стандарте 35
1.3.3. Эффективность библиотеки многопоточности для C++ 36
1.3.4. Платформенно-зависимые средства 37
1.4. В начале пути 38
1.4.1. Здравствуй, параллельный мир 38
1.5. Резюме 40
ГЛАВА 2. Управление потоками 41
2.1. Базовые операции управления потоками 41
2.1.1. Запуск потока 42
2.1.2. Ожидание завершения потока 45
2.1.3. Ожидание в случае исключения 46
2.1.4. Запуск потоков в фоновом режиме 48
2.2. Передача аргументов функции потока 51
2.3. Передача владения потоком 54
2.4. Задание количества потоков во время выполнения 58
2.5. Идентификация потоков 61
2.6. Резюме 64

ГЛАВА 3. Разделение данных между потоками 65
3.1. Проблемы разделения данных между потоками 66
3.1.1. Гонки 68
3.1.2. Устранение проблематичных состояний гонки 69
3.2. Защита разделяемых данных с помощью мьютексов 70
3.2.1. Использование мьютексов в C++ 71
3.2.2. Структурирование кода для защиты разделяемых данных 73
3.2.3. Выявление состояний гонки, внутренне присущих интерфейсам 74
3.2.4. Взаимоблокировка: проблема и решение 83
3.2.5. Дополнительные рекомендации, как избежать взаимоблокировок 86
3.2.6. Гибкая блокировка с помощью std::unique_lock 94
3.2.7. Передача владения мьютексом между контекстами.......... 95
3.2.8. Выбор правильной гранулярности блокировки 97
3.3. Другие средства защиты разделяемых данных 100
3.3.1. Защита разделяемых данных во время инициализации..... 100
3.3.2. Защита редко обновляемых структур данных.................. 105
3.3.3. Рекурсивная блокировка.. 107
3.4. Резюме 108

ГЛАВА 4. Синхронизация параллельных операций 110
4.1. Ожидание события или иного условия 111
4.1.1. Ожидание условия с помощью условных переменных..... 112
4.1.2. Потокобезопасная очередь на базе условных переменных 115
4.2. Ожидание одноразовых событий с помощью механизма будущих результатов 121
4.2.1. Возврат значения из фоновой задачи 122
4.2.2. Ассоциирование задачи с будущим результатом 125
Передача задач между потоками.. 127
4.2.3. Использование std::promise 129
4.2.4. Сохранение исключения в будущем результате 131
4.2.5. Ожидание в нескольких потоках.................................... 133
4.3. Ожидание с ограничением по времени 136
4.3.1. Часы 137
4.3.2. Временные интервалы... 138
4.3.3. Моменты времени 140
4.3.4. Функции, принимающие таймаут 142
4.4. Применение синхронизации операций для упрощения кода.. 144

4.4.1. Функциональное программирование с применением будущих результатов 145
4.5. Резюме 156

ГЛАВА 5. Модель памяти C++ и атомарные операции 158
5.1. Основы модели памяти 159
5.1.1. Объекты и ячейки памяти 159
5.1.2. Объекты, ячейки памяти и параллелизм 161
5.1.3. Порядок модификации 162
5.2. Атомарные операции и типы в C++ 163
5.2.1. Стандартные атомарные типы 163
5.2.2. Операции над std::atomic_flag 167
5.2.3. Операции над std::atomic<bool> 170
5.2.4. Операции над std::atomic<T*>: арифметика указателей 173
5.2.5. Операции над стандартными атомарными целочисленными типами 175
5.2.6. Основной шаблон класса std::atomic<> 175
5.2.7. Свободные функции для атомарных операций 177
5.3. Синхронизация операций и принудительное упорядочение 180
5.3.1. Отношение синхронизируется-с 182
5.3.2. Отношение происходит-раньше 183
5.3.3. Упорядочение доступа к памяти для атомарных операций 185
5.3.4. Последовательности освобождений и отношение синхронизируется-с 208
5.3.5. Барьеры 212
5.3.6. Упорядочение неатомарных операций с помощью атомарных 214
5.4. Резюме 216

ГЛАВА 6. Проектирование параллельных структур данных с блокировками 218
6.1. Что понимается под проектированием структур данных, рассчитанных на параллельный доступ? 219
6.1.1. Рекомендации по проектированию структур данных для параллельного доступа 220
6.2. Параллельные структуры данных с блокировками 222
6.2.1. Потокобезопасный стек с блокировками 222
6.2.2. Потокобезопасная очередь с блокировками и условными переменными 226
6.2.3. Потокобезопасная очередь с мелкогранулярными блокировками и условными переменными 231

6.3. Проектирование более сложных структур данных с блокировками 245
6.3.1. Разработка потокобезопасной справочной таблицы с блокировками 246
6.3.2. Потокобезопасный список с блокировками 253
6.4. Резюме 258

ГЛАВА 7. Проектирование параллельных структур данных без блокировок 260
7.1. Определения и следствия из них 261
7.1.1. Типы неблокирующих структур данных 262
7.1.2. Структуры данных, свободные от блокировок 262
7.1.3. Структуры данных, свободные от ожидания 263
7.1.4. Плюсы и минусы структур данных, свободных от блокировок 264
7.2. Примеры структур данных, свободных от блокировок 266
7.2.1. Потокобезопасный стек без блокировок 266
7.2.2. Устранение утечек: управление памятью в структурах данных без блокировок 271
7.2.3. Обнаружение узлов, не подлежащих освобождению, с помощью указателей опасности 277
7.2.4. Нахождение используемых узлов с помощью подсчета ссылок 287
7.2.5. Применение модели памяти к свободному от блокировок стеку 293
7.2.6. Потокобезопасная очередь без блокировок 299
7.3. Рекомендации по написанию структур данных без блокировок 313
7.3.1. Используйте std::memory_order_seq_cst для создания прототипа 314
7.3.2. Используйте подходящую схему освобождения памяти 314
7.3.3. Помните о проблеме ABA 315
7.3.4. Выявляйте циклы активного ожидания и помогайте другим потокам 316
7.4. Резюме 317

ГЛАВА 8. Проектирование параллельных программ 318
8.1. Методы распределения работы между потоками 319
8.1.1. Распределение данных между потоками до начала обработки 320
8.1.2. Рекурсивное распределение данных 322
8.1.3. Распределение работы по типам задач 327

8.2. Факторы, влияющие на производительность параллельного кода........ 330
8.2.1. Сколько процессоров?........ 331
8.2.2. Конкуренция за данные и перебрасывание кэша........ 333
8.2.3. Ложное разделение........ 335
8.2.4. Насколько близки ваши данные?........ 336
8.2.5. Превышение лимита и чрезмерное контекстное переключение........ 337
8.3. Проектирование структур данных для повышения производительности многопоточной программы........ 338
8.3.1. Распределение элементов массива для сложных операций........ 339
8.3.2. Порядок доступа к другим структурам данных........ 342
8.4. Дополнительные соображения при проектировании параллельных программ........ 344
8.4.1. Безопасность относительно исключений в параллельных алгоритмах........ 344
8.4.2. Масштабируемость и закон Амдала........ 353
8.4.3. Сокрытие латентности с помощью нескольких потоков... 355
8.4.4. Повышение быстроты реакции за счет распараллеливания........ 356
8.5. Проектирование параллельного кода на практике........ 359
8.5.1. Параллельная реализация std::for_each........ 359
8.5.2. Параллельная реализация std::find........ 362
8.5.3. Параллельная реализация std::partial_sum........ 369
8.6. Резюме........ 380

ГЛАВА 9. Продвинутое управление потоками ... 382

9.1. Пулы потоков........ 383
9.1.1. Простейший пул потоков........ 383
9.1.2. Ожидание задачи, переданной пулу потоков........ 386
9.1.3. Задачи, ожидающие других задач........ 391
9.1.4. Предотвращение конкуренции за очередь работ........ 394
9.1.5. Занимание работ........ 396
9.2. Прерывание потоков........ 401
9.2.1. Запуск и прерывание другого потока........ 402
9.2.2. Обнаружение факта прерывания потока........ 404
9.2.3. Прерывание ожидания условной переменной........ 405
9.2.4. Прерывание ожидания std::condition_variable_any........ 409
9.2.5. Прерывание других блокирующих вызовов........ 411
9.2.6. Обработка прерываний........ 412
9.2.7. Прерывание фоновых потоков при выходе из приложения..... 413
9.3. Резюме........ 415

ГЛАВА 10. Тестирование и отладка многопоточных приложений 416

10.1. Типы ошибок, связанных с параллелизмом 417
10.1.1. Нежелательное блокирование 417
10.1.2. Состояния гонки 418
10.2. Методы поиска ошибок, связанных с параллелизмом ... 420
10.2.1. Анализ кода на предмет выявления потенциальных ошибок.. 420
10.2.2. Поиск связанных с параллелизмом ошибок путем тестирования 423
10.2.3. Проектирование с учетом тестопригодности 425
10.2.4. Приемы тестирования многопоточного кода................ 427
10.2.5. Структурирование многопоточного тестового кода....... 431
10.2.6. Тестирование производительности многопоточного кода 435
10.3. Резюме 436

ПРИЛОЖЕНИЕ A. Краткий справочник по некоторым конструкциям языка C++ 437

A.1. Ссылки на *r*-значения... 437
A.1.1. Семантика перемещения.. 439
A.1.2. Ссылки на *r*-значения и шаблоны функций..................... 442
A.2. Удаленные функции 442
A.3. Умалчиваемые функции 445
A.4. constexpr-функции 449
A.4.1. constexpr и определенные пользователем типы............. 450
A.4.2. constexpr-объекты 454
A.4.3.Требования к constexpr-функциям 454
A.4.4. constexpr и шаблоны... 455
A.5. Лямбда-функции 456
A.5.1. Лямбда-функции, ссылающиеся на локальные переменные ... 458
A.6. Шаблоны с переменным числом параметров................. 461
A.6.1. Расширение пакета параметров 463
A.7. Автоматическое выведение типа переменной................ 466
A.8. Поточно-локальные переменные 467
A.9. Резюме... 469

ПРИЛОЖЕНИЕ B. Краткое сравнение библиотек для написания параллельных программ 470

ПРИЛОЖЕНИЕ C. Каркас передачи сообщений и полный пример программы банкомата 472

ПРИЛОЖЕНИЕ D. Справочник по библиотеке C++ Thread Library.................................... 492
D.1. Заголовок <chrono> 492
D.1.1. Шаблон класса std::chrono::duration 493
D.1.2. Шаблон класса std::chrono::time_point 503
D.1.3. Класс std::chrono::system_clock 506
D.1.4. Класс std::chrono::steady_clock..................................... 508
D.1.5. Псевдоним типа std::chrono::high_resolution_clock.......... 510
D.2. Заголовок <condition_variable> 511
D.2.1. Класс std::condition_variable 511
D.2.2. Класс std::condition_variable_any.................................... 521
D.3. Заголовок <atomic>.. 530
D.3.1. std::atomic_xxx, псевдонимы типов 531
D.3.2. ATOMIC_xxx_LOCK_FREE, макросы 532
D.3.3. ATOMIC_VAR_INIT, макрос 533
D.3.4. std::memory_order, перечисление 533
D.3.5. std::atomic_thread_fence, функция 534
D.3.6. std::atomic_signal_fence, функция 535
D.3.7. std::atomic_flag, класс.. 535
D.3.8. Шаблон класса std::atomic... 539
D.3.9. Специализации шаблона std::atomic 552
D.3.10. Специализации std::atomic<integral-type> 552
D.4. Заголовок <future> 571
D.4.1. Шаблон класса std::future.. 572
D.4.2. Шаблон класса std::shared_future 578
D.4.3. Шаблон класса std::packaged_task 585
D.4.4. Шаблон класса std::promise... 592
D.4.5. Шаблон функции std::async 598
D.5. Заголовок <mutex>.. 600
D.5.1. Класс std::mutex 601
D.5.2. Класс std::recursive_mutex 603
D.5.3. Класс std::timed_mutex 606
D.5.4. Класс std::recursive_timed_mutex 611
D.5.5. Шаблон класса std::lock_guard 615
D.5.6. Шаблон класса std::unique_lock..................................... 617
D.5.7. Шаблон функции std::lock.. 628
D.5.8. Шаблон функции std::try_lock 629
D.5.9. Класс std::once_flag... 630
D.5.10. Шаблон функции std::call_once.................................... 630
D.6. Заголовок <ratio> 631
D.6.1. Шаблон класса std::ratio 632
D.6.2. Псевдоним шаблона std::ratio_add................................. 633
D.6.3. Псевдоним шаблона std::ratio_subtract 634
D.6.4. Псевдоним шаблона std::ratio_multiply........................... 635

D.6.5. Псевдоним шаблона std::ratio_divide 635
D.6.6. Шаблон класса std::ratio_equal 636
D.6.7. Шаблон класса std::ratio_not_equal 636
D.6.8. Шаблон класса std::ratio_less 637
D.6.9. Шаблон класса std::ratio_greater 637
D.6.10. Шаблон класса std::ratio_less_equal 638
D.6.11. Шаблон класса std::ratio_greater_equal 638
D.7. Заголовок <thread> 638
D.7.1. Класс std::thread 639
D.7.2. Пространство имен this_thread 649

РЕСУРСЫ 652
Печатные ресурсы 652
Сетевые ресурсы 653

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 654

ГЛАВА 1. Здравствуй, параллельный мир!

В этой главе:

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

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

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

стремясь повысить производительность приложений, все чаще посматривают в сторону параллелизма вообще и многопоточного программирования в особенности.

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

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

1.1. Что такое параллелизм?

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

1.1.1. Параллелизм в вычислительных системах

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

одновременного выполнения задач, а действительно исполняют их параллельно.

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

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

На рис. 1.1 показан идеализированный случай: компьютер, исполняющий ровно две задачи, каждая из которых разбита на десять одинаковых этапов. На двухъядерной машине каждая задача может исполняться в своем ядре. На одноядерной машине с переключением задач этапы той и другой задачи чередуются. Однако между ними существует крохотный промежуток времени (на рисунке эти промежутки изображены в виде серых полосок, разделяющих более широкие этапы выполнения) – чтобы обеспечить чередование, система должна произвести *контекстное переключение* при каждом переходе от одной задачи к другой, а на это требуется время. Чтобы переключить контекст, ОС должна сохранить состояние процессора и счетчик ко-

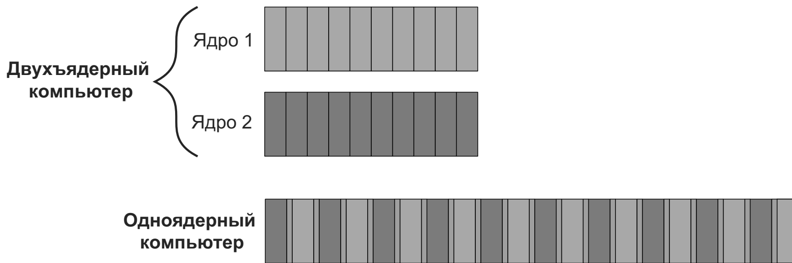

Рис. 1.1. Два подхода к параллелизму: параллельное выполнение на двухъядерном компьютере и переключение задач на одноядерном

манд для текущей задачи, определить, какая задача будет выполняться следующей, и загрузить в процессор состояние новой задачи. Не исключено, что затем процессору потребуется загрузить команды и данные новой задачи в кэш-память; в течение этой операции никакие команды не выполняются, что вносит дополнительные задержки.

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

Все рассматриваемые в этой книге приемы, функции и классы применимы вне зависимости от того, исполняется приложение на ма-

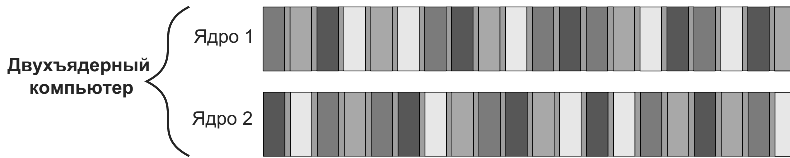

Рис. 1.2. Переключение задач на двухъядерном компьютере

шине с одноядерным процессором или с несколькими многоядерными процессорами. Не имеет значения, как реализован параллелизм: с помощью переключения задач или аппаратно. Однако же понятно, что способ использования параллелизма в приложении вполне может зависеть от располагаемого оборудования. Эта тема обсуждается в главе 8 при рассмотрении вопросов проектирования параллельного кода на C++.

1.1.2. Подходы к организации параллелизма

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

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

Эти два способа организации труда разработчиков иллюстрируют два основных подхода к параллелизму. Разработчик – это модель потока, а кабинет – модель процесса. В первом случае имеется несколько однопоточных процессов (у каждого разработчика свой кабинет), во втором – несколько потоков в одном процессе (два разработчика в одном кабинете). Разумеется, возможны произвольные комбинации:

может быть несколько процессов, многопоточных и однопоточных, но принцип остается неизменным. А теперь поговорим немного о том, как эти два подхода к параллелизму применяются в приложениях.

Параллелизм за счет нескольких процессов

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

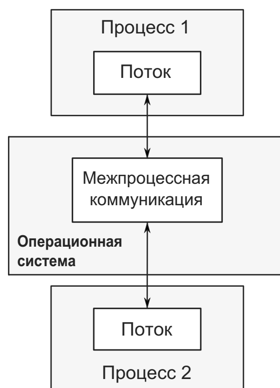

Рис. 1.3. Коммуникация между двумя параллельно работающими процессами

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

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

Параллелизм за счет нескольких потоков

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

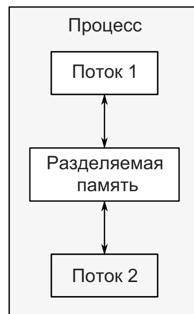

Рис. 1.4. Коммуникация между двумя параллельно исполняемыми потоками в одном процессе

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

Низкие накладные расходы на запуск потоков внутри процесса и коммуникацию между ними стали причиной популярности этого подхода во всех распространенных языках программирования, включая C++, даже несмотря на потенциальные проблемы, связанные с разделением памяти. Кроме того, в стандарте C++ не оговаривается встроенная поддержка межпроцессной коммуникации, а, значит, при-

ложения, основанные на применении нескольких процессов, вынуждены полагаться на платформенно-зависимые API. Поэтому в этой книге мы будем заниматься исключительно параллелизмом на основе многопоточности, и в дальнейшем всякое упоминание о параллелизме предполагает использование нескольких потоков.

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

1.2. Зачем нужен параллелизм?

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

1.2.1. Применение параллелизма для разделения обязанностей

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

Рассмотрим приложение, имеющее графический интерфейс и выполняющее сложные вычисления, например DVD-проигрыватель на настольном компьютере. У такого приложения два принципиально разных набора обязанностей: во-первых, читать данные с диска, декодировать изображение и звук и своевременно посылать их графическому и звуковому оборудованию, чтобы при просмотре фильма не было заминок, а, во-вторых, реагировать на действия пользователя, например, на нажатие кнопок «Пауза», «Возврат в меню» и даже «Выход». Если бы приложение было однопоточным, то должно

было бы периодически проверять, не было ли каких-то действий пользователя, поэтому код воспроизведения DVD перемежался бы кодом, относящимся к пользовательскому интерфейсу. Если же для разделения этих обязанностей использовать несколько потоков, то код интерфейса и воспроизведения уже не будут так тесно переплетены: один поток может заниматься отслеживанием действий пользователя, а другой – воспроизведением. Конечно, как-то взаимодействовать они все равно должны, например, если пользователь нажимает кнопку «Пауза», но такого рода взаимодействия непосредственно связаны с решаемой задачей.

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

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

1.2.2. Применение параллелизма для повышения производительности

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

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

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

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

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

гаемое повышение производительности иногда открывает новые возможности. Например, если разные области изображения можно обрабатывать параллельно, то можно будет обработать видео более высокого разрешения.

1.2.3. Когда параллелизм вреден?

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

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

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

рано или поздно возникнут проблемы. Для ограничения количества потоков можно воспользоваться пулами потоков (см. главу 9), но и это не панацея – у пулов есть и свои проблемы.

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

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

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

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

1.3. Параллелизм и многопоточность в C++

Стандартизованная поддержка параллелизма за счет многопоточности – вещь новая для C++. Только новый стандарт C++11 позволит писать многопоточный код, не прибегая к платформенно-зависи-

мым расширениям. Чтобы разобраться в подоплёке многочисленных решений, принятых в новой стандартной библиотеке C++ Thread Library, необходимо вспомнить историю.

1.3.1. История многопоточности в C++

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

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

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

Во многих случаях поддержка многопоточности в имеющихся компиляторах C++ вкупе с доступностью платформенно-зависимых API и платформенно-независимых библиотек классов типа Boost и ACE оказывается достаточно прочным основанием, на котором можно писать многопоточные программы. В результате уже написаны многопоточные приложения на C++, содержащие миллионы строк кода. Но коль скоро прямой поддержки в стандарте нет, бывают случаи, когда отсутствие модели памяти, учитывающей многопоточность, приводит к проблемам. Особенно часто с этим сталкиваются разработчики, пытающиеся увеличить производительность за счет использования особенностей конкретного процессора, а также те, кто пишет кросс-платформенный код, который должен работать независимо от различий между компиляторами на разных платформах.

1.3.2. Поддержка параллелизма в новом стандарте

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

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

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

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

1.3.3. Эффективность библиотеки многопоточности для C++

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

Комитет по стандартизации C++ прекрасно понимал это, когда проектировал стандартную библиотеку C++ вообще и стандартную библиотеку многопоточности для C++ в частности. Среди целей проектирования была и такая: выигрыш от использования низкоуровневых средств по сравнению с высокоуровневой оберткой (если такая предоставляется) должен быть ничтожен или отсутствовать вовсе. Поэтому библиотека спроектирована так, чтобы ее можно было эффективно реализовать (с очень небольшой платой за абстрагирование) на большинстве популярных платформ.

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