Глава 8

Краткое введение в потоки выполнения

Многопоточное программирование заметно отличается от однопоточного, поскольку потоки выполнения нередко проявляют неожиданное и непредвиденное поведение. Следовательно, было бы безответственно, обсудив фоновые потоки выполнения в пользовательских интерфейсах, не пояснить, каким образом запускается, останавливается и отслеживается выполнение длительных заданий. В этой главе рассматриваются новые трудные задачи и вопросы, возникающие в связи с применением многих потоков выполнения, будь то в форме конкретных объектов типа `Thread` или посредством каркасов для выполнения задач. Надеемся, что, прочитав данную главу, вы заинтересуетесь этой весьма любопытной и всячески достойной практического освоения темы, обратившись к дополнительной литературе.

↰ 7.10
↰ 7.10.2
📖 148

8.1. Основы: параллельное выполнение кода

Пользователи современных приложений ожидают от них выполнение многих действий одновременно: воспроизведения анимации, выполнения обширных вычислений, получения доступа к сети и многого другого. Управление этими действиями намного упрощается, если запрограммировать каждую задачу как единственную, а платформа Java возьмет на себя заботу за одновременное (или хотя бы виртуально одновременное) их выполнение. Именно потоки выполнения и позволяют реализовать такую модель программирования.

> Заключайте фрагменты исполняемого кода в оболочку объектов типа `Runnable`, чтобы запускать их на параллельное выполнение.

Допустим, требуется загружать файлы и одновременно выполнять сложные вычисления. С этой целью исполняемый код заключается в оболочку объектов типа `Runnable` следующим образом:

↰ 1.8.6

threads.MinimalThreads

```
class DownloadWorker implements Runnable {
    public void run() {
        ... выполнить работу как единственную задачу
    }
}
class ComputationWorker implements Runnable {
    public void run() {
        ... выполнить работу как единственную задачу
    }
}
```

Затем платформе Java дается команда одновременно выполнить оба фрагмента код в теле методов `run()`, построив объекты типа `Thread` и запустив соответствующие потоки на исполнение, как показано ниже.

threads.MinimalThreads.startActivities

```
new Thread(new DownloadWorker()).start();
new Thread(new ComputationWorker()).start();
```

📖 148

В приведенном выше фрагменте кода демонстрируется вполне приемлемый способ создания потоков выполнения. В принципе прикладной программный интерфейс API допускает переопределение метода `run()` из класса `Thread` непосредственно перед запуском потока выполнения, но преимущество приведенного выше решения заключается в том, что инфраструктура языка отделяется от прикладных объектов.

> Планировщик потоков выполнения случайным образом решает, какие именно потоки следует выполнять.

Результирующее поведение во время выполнения наглядно показано на рис. 8.1. Код задачи загрузки файлов (D) и задачи сложных вычислений (C) исполняется в течение коротких периодов времени, а затем он вытесняется и на исполнение запускается код в другом потоке. Поскольку переход между потоками выполнения осуществляется довольно часто, то с точки зрения пользователя обе задачи выполняются одновременно.

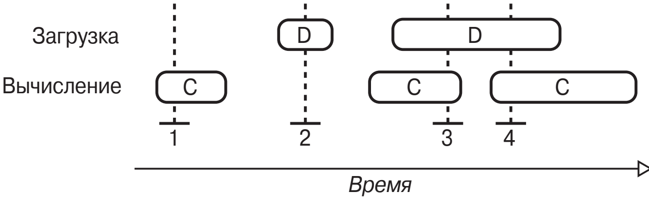


Рис. 8.1. Поочередное исполнение кода в двух потоках

Здесь уместно пояснить связь между параллельным и одновременным выполнением. Планировщик потоков выполнения обеспечивает одновременное выполнение задач, и поэтому можно сказать, что они выполняются *одновременно* и, по существу, *распараллелено*, а также вести речь о *многозадачном* или *параллельном программировании* и т.д. Современные центральные процессоры состоят из нескольких ядер, и поэтому потоки зачастую исполняются действительно *параллельно*. Вряд ли стоит допускать, что во время выполнения одного потока остановятся другие потоки, поэтому рассматриваемую здесь связь можно кратко охарактеризовать следующим образом: распараллеливание — это потенциальный параллелизм.

Рассмотрим пример, приведенный на рис. 8.1, более подробно. В моменты 1 и 2 оба потока выполняются по отдельности, но их исполнение может перекрываться, как это и происходит в моменты 3 и 4, если в системе имеется несколько центральных процессоров или один многоядерный центральный процессор. Вопрос о том, какой именно поток выполняется в конкретный момент времени, решается центральным *планировщиком* потоков выполнения, который обычно находится на уровне операционной системы. Его решения зависят от того, сколько потоков ожидают выполнения, от отличий в приоритетах потоков, попытки прочитать в потоке еще недоступные данные, какой-нибудь блокировки, ожидаемой в отдельном потоке, и многих других факторов. С точки зрения самого приложения решения, принимаемые планировщиком потоков выполнения, кажутся совершенно произвольными и недетерминированными, и поэтому следует всегда допускать, что поток может выполняться в любой момент, как бы активно этому ни препятствовать, например, с помощью блокировок.

⏩ 8.2

Потоки выполнения ставят совершенно новые задачи программирования.

В остальной части этой главы обсуждаются решения следующих основных задач, которые возникают в связи с применением потоков выполнения (рис. 8.2).

1. Когда два потока выполнения с непредсказуемым поведением, показанным на рис. 8.1, пытаются получить доступ к одним и тем же данным, состояние структуры данных также становится непредсказуемым, поскольку неясно, какая именно операция записи произойдет до операции чтения. В связи с тем что неясно, какая именно операция будет выполнена первой, такое затруднение называется *состоянием гонки*. В качестве выхода из этого затруднения рекомендуется ограничить недетерминированное поведение, чтобы писать программы, которые выполняются правильно и надежно.

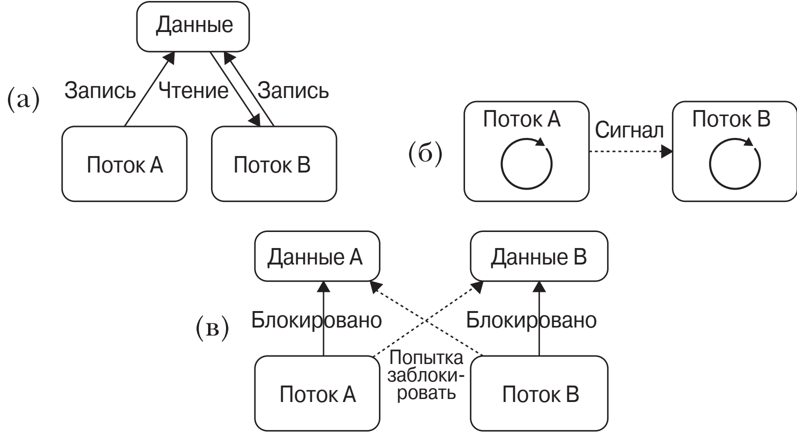


Рис. 8.2. Новые задачи программирования, возникающие в связи с применением потоков выполнения

2. Иногда потоки должны обмениваться данными — главным образом, о наступающих событиях. Однако в каждом потоке выполняется свой код, и поэтому в одном потоке выполнения нельзя вызвать методы из другого потока. Следовательно, требуется какой-то механизм для обмена сигналами между потоками выполнения и для ожидания этих сигналов в каждом потоке.
3. Первое затруднение разрешается блокировкой данных. Перед доступом к данным поток выполнения должен получить *блокировку* по этим данным. Поскольку оба потока выполнения не могут получить блокировку одновременно, то порядок операций записи и чтения должен стать в какой-то степени более предсказуемым. Но при этом возникает новое затруднение. Так, если потоки выполнения пытаются получить доступ к нескольким фрагментам данных, то вполне возможно, что каждый из них приобретет одну блокировку и перейдет в состояние бесконечного ожидания, поскольку другой поток удерживает иную блокировку. Такое явление называется *взаимной блокировкой* и неприятно тем, что его трудно воспроизвести, а следовательно, выявить и устранить.

> Объекты, действие которых ограничивается отдельными потоками выполнения, не требуют особого внимания.

Прежде чем вдаваться в подробности, сформулируем вкратце самую распространенную методику борьбы с подобными явлениями. Так, если ссылка на объект удерживается в одном потоке выполнения, он может оперировать этим объектом, как будто многопоточная обработка вообще отсутствует, поскольку другие потоки выполнения не смогут вмешаться в операции с данным объектом. Такая методика называется *привязкой к потоку*, при которой

интересующий объект вообще не “исчезает” из конкретного потока выполнения. Это гарантируется соблюдением строгой дисциплины владения, исключающей утечку ссылок на объекты в тех местах, где они могут быть доступны из других потоков выполнения. Привязка к потоку служит более глубокой причиной для внедрения специально выделяемого потока диспетчеризации событий в пользовательских интерфейсах и применения других потоков выполнения в качестве только “поставщиков” данных, которые они передают потоку диспетчеризации событий через метод `asyncExec()`. К числу наиболее эффективных стратегий копирования в потоках выполнения относится выявление как можно большего количества структур объектов для управления в одном потоке выполнения. Остальные стратегии рассматриваются далее.

📖 148 (§2.3)
↰ 2.2.1
↰ 7.10.1

8.2. Правильность кода в присутствии потоков выполнения

Чтобы перевести обсуждение в конкретное русло, вернемся к первоначальному побуждению применять фоновые потоки в пользовательских интерфейсах. На рис. 8.3 приведен шаблон, наблюдаемый во многих случаях. В пользовательском интерфейсе порождаются два потока выполнения, A и B, причем в обоих так или иначе накапливаются данные и соответственно изменяются структуры данных приложения, а пользовательский интерфейс зеркально отображает эти данные на экране. Но поскольку в отображении данных уже задействован поток диспетчеризации событий, то те же самые затруднения возникнут и с единственным фоновым потоком A. Ниже перечислены лишь некоторые примеры возникновения подобных ситуаций.

↰ 7.10
⏩ 9.1
↰ 7.10.1

- Приложением служит веб-браузер, а в потоках одновременно выполняются разные операции загрузки веб-страниц.

📖 237

- Приложением служит сервер, а каждый поток выполнения отвечает за сообщение с единственным клиентом.

📖 235

- Приложением служит интегрированная среда разработки Eclipse, а в потоке построителя проектов модель EMF преобразуется в исходный код Java, сохраняемый в файле, к которому в настоящий момент обращается редактор.

Потоки выполнения затрудняют аргументирование правильности кода.

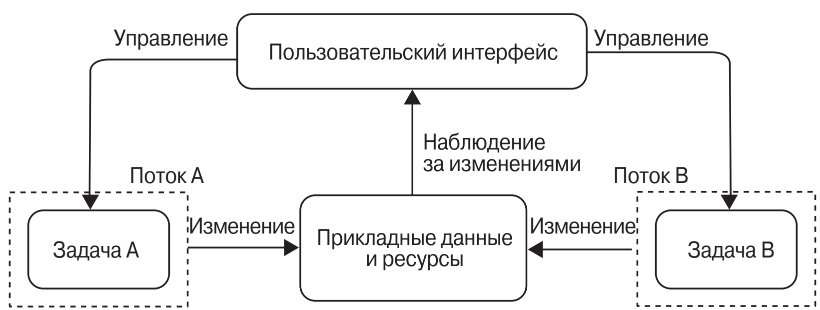


Рис. 8.3. Трудности многопоточной обработки

Главная трудность, возникающая в ситуации, приведенной на рис. 8.3, состоит в том, что доступ к *общим ресурсам* вроде структур данных должен координироваться во избежание хаоса в представлении ресурсов и опасности нарушения правильности программного обеспечения. Ведь вся аргументация правильности кода с помощью контрактов, по существу, основывается на понятии утверждений, закрепляющих знания о поведении программы в формулировках, аналогичных следующей: “Всякий раз, когда исполнение достигает данного места в коде, известно, что ...”.

↰ 4
↰ 4.1

Потоки выполнения вносят проблему *взаимного вмешательства.* Это означает, что если в одном потоке выполнения достигается утверждение, то в следующую микросекунду становится известно, что оно удовлетворяется, тогда как во втором потоке выполнения видоизменяется область памяти, описываемая в утверждении, а следовательно, оно больше не удовлетворяется. Иными словами, пользуясь потоками выполнения, следует иметь в виду, что неожиданные модификации оперативной памяти могут произойти в любой момент, если только не исключить их явным образом.

🕮 6, 15, 204

Особая опасность кроется в нарушении изощренных зачастую инвариантов объектов. Такой инвариант описывает внешний вид “правильного” объекта и должен удовлетворяться всякий раз, когда выполняется код этого

объекта. Но если объект в настоящий момент “работает”, то он временно нарушает свой инвариант. В итоге в разных потоках выполнения, получающих доступ к одному и тому же объекту, уже нельзя допустить, что его инвариант удовлетворяется.

↤ 4.1

Чтобы продемонстрировать проблему взаимного вмешательства, попробуем нарушить инвариант класса `GapTextStore`. Проведенный ранее анализ убедил нас, что код этого класса правильный. Тем не менее выполнение приведенного ниже кода приведет к исключению. В частности, при попытке одновременного доступа к объекту этого класса в строке 7 данного кода из двух потоков выполнения, создаваемых в строках 11 и 12, генерируется соответствующее исключение. Из-за сложных арифметических операций над индексами, вероятнее всего, возникнет исключение типа `ArrayIndexOutOfBoundsException`.

↤ 1.3.1, ↤ 4.1

swt.threads.BreakRepresentation.main

```
1 final GapTextStore shared = new GapTextStore();
2 class Worker implements Runnable {
3    public void run() {
4        for (int i = 0; i != 10000; i++) {
5            String text = String.format("%03d ",
6            rand.nextInt(1000));
7            shared.replace(shared.getLength(), 0, text);
8        }
9    }
10 }
11 new Thread(new Worker()).start();
12 new Thread(new Worker()).start();
```

Минимальное требование к потокам выполнения состоит во взаимном исключении.

Взаимное влияние может возникнуть в любой момент, если не предотвратить его явным образом. В частности, *взаимное исключение* означает отказ всем, кроме одного, потокам выполнения в доступе к общему ресурсу на некоторый период времени, чтобы этот поток смог действовать независимо от других потоков выполнения в системе. Взаимное исключение является минимальным требованием к потокам выполнения, поскольку без него недетерминированное планирование этих потоков приведет к непредсказуемому состоянию общего ресурса.

Взаимное исключение обычно достигается с помощью *блокировок*, как демонстрируется в приведенном ниже фрагменте кода. Сначала в нем создается объект блокировки типа `Lock`. Для установки и снятия блокировок имеются два соответствующих метода: `lock()` и `unlock()`. Когда в потоке выполнения вызывается метод `lock()`, этот поток *приобретает* блокировку и *удерживает* ее до тех пор, пока не вызовет метод `unlock()`. Внутренние механизмы блокировок гарантируют от того, чтобы два каких-нибудь потока выполнения вообще могли удерживать одну и ту же блокировку одновременно. Если в одном потоке выполнения вызывается метод `lock()` в то время, как другой поток уже удерживает блокировку, то первый поток должен ожидать до тех пор, пока блокировка не станет доступной снова.

swt.threads.PreserveRepresentation.main

```
final Lock lock = new ReentrantLock();
```

В интерфейсе `Lock` определяется лишь самое элементарное поведение блокировок, которое может иметь свои отклонения. Одно из таких типичных отклонений в поведении блокировок реализуется в классе `ReentrantLock`, где потоку выполнения, уже удерживающему блокировку, разрешается установить ее снова, вызвав метод `lock()`. При блокировке ведется подсчет количества вызовов методов `lock()` и `unlock()`, и лишь тогда, когда этот подсчет достигает нуля, блокировка снимается. Такое поведение блокировок удобно, например, в том случае, если в одном открытом методе вызывается другой открытый метод, и они оба начинают свое выполнение с блокировки всего объекта, как демонстрируется в следующем далее примере кода.

Применяя блокировки, мы можем теперь защитить класс `GapTextStore` от нарушения его инварианта, заменив проблематичную строку 6 в упомянутом выше примере кода на приведенный ниже код блокировки. В строке 1 данного кода поток, исполняющий этот код, получает блокировку или ожидает до тех пор, пока она не станет доступной. Но самое главное, что выполнение данного кода может достигнуть строки 3 лишь в одном потоке выполнения, а именно в потоке, удерживающем блокировку `lock`. В итоге модификация промежуточного хранилища текста может быть произведена только в одном потоке выполнения, а следовательно, аргументация правильности кода, реализующего это хранилище, не нарушается вследствие взаимного влияния и никакого исключения вообще не возникает. Блокировка защищает код от взаимного влияния в промежутках между вызовами методов `lock()` и `unlock()`.

swt.threads.PreserveRepresentation.main

```
1 lock.lock();
2 try {
3    shared.replace(shared.getLength(), 0, text);
4 } finally {
5    lock.unlock();
6 }
```

Всегда снимайте блокировку, которую вы приобрели в своем коде, ведь иначе общий ресурс окажется недоступным снова. В приведенном выше примере кода демонстрируется методика применения для этой цели блока операторов `try`/`finally`. Блокировка снимается независимо от того, каким образом оставляется защищенный от взаимного влияния промежуток между вызовами методов `lock()` и `unlock()`.

🕮 148
↰ 1.5.6

Зачастую блокировки концептуально связаны с конкретным объектом или группой объектов. Код, в котором требуется оперировать объектом, должен заранее приобрести блокировку и освободить ее впоследствии. В подобных случаях можно сказать, что один поток выполнения удерживает блокировку *по* объекту, даже если связь с ним осуществляется только концептуально, а не технически.

В языке Java предоставляется встроенный механизм блокировок, реализуемый с помощью ключевого слова `synchronized`. В этом случае с каждым объектом связана блокировка, которая приобретается и освобождается по ключевому слову `synchronized`, — как правило, для текущего объекта по ссылке `this`. Если указать ключевое слово `synchronized` до всех методов класса, то защищены будут все его внутренние инварианты. Только один поток выполнения может одновременно получить доступ к коду этого класса, а следовательно, правильность такого кода подтверждается обычными аргументами. Такие объекты иначе называются *мониторами*. (К сожалению, этот термин употребляется в спецификации языка Java в разном смысле.) По сравнению с объектами блокировок типа **`Lock`**, механизм мониторов намного менее гибкий. Его негибкость обсуждается далее в соответствующих местах.

🕮 148
↰ 4.1

Многопоточная обработка на современных процессорах выходит далеко за рамки недетерминированного выполнения того или иного потока. В этом случае потоки могут фактически выполняться параллельно на разных центральных процессорах или в разных ядрах одного и того же центрального процессора (рис. 8.4). Но тогда иерархическая структура памяти процессора вносит новую проблему, помимо взаимного влияния: когда в одном потоке выполнения перезаписывается область памяти, доступная из другого потока, то в другом потоке об этом может быть вообще ничего неизвестно. Допустим, что один поток выполняется на центральный процессор А и в нем записываются какие-нибудь данные в общий объект. Ради повышения эффективности эти данные не записываются в основную (т.е. оперативную) память, а сохраняются в локальном кеше этого центрального процессора до тех пор, пока не будут затребованы из кеша где-нибудь еще. Когда выполняется второй поток на центральном процессора В, то данные читаются в нем только из локального кеша этого процессора и ему ничего неизвестно о новых данных в локальном кеше центрального процессора А.

🕮 115

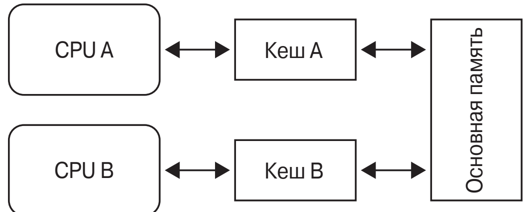


Рис. 8.4. Потоки выполнения и кеширование

К механизмам взаимного исключения, применяемым в Java, относится также *барьер памяти*, синхронизирующий работу разных кешей и основной памяти на аппаратном уровне. Благодаря этому во всех потоках выполнения доступны текущие данные, хранящиеся в объекте. Барьер памяти устанавливается также с помощью ключевого слова `volatile`, но в то же время не предоставляется блокировка. Более подробно этот механизм описывается в документации на *модель памяти Java*. Сама спецификация этого механизма обескураживает. По существу, она гарантирует, что если применяется блокировка или ключевое слово `volatile`, то операции чтения и записи данных в область памяти всегда происходят в определенном порядке в том смысле, что чтение доступно или не доступно после записи в зависимости от решений, принимаемых планировщиком потоков выполнения, но состояние неопределенности в промежутке между этими операциями исключается.

📖 111, 166

Включайте защиту от взаимного влияния в инкапсуляцию.

Защита промежуточного хранилища текста в рассматриваемом здесь примере довольно ненадежна, поскольку все клиенты должны подчиняться протоколу приобретения блокировки перед доступом к этому хранилищу. Лучшее решение состоит в том, чтобы создавать объекты таким образом, чтобы они защищали свои структуры данных и инварианты собственными блокировками. Такие объекты называются *потокобезопасными*.

↰ 2.4

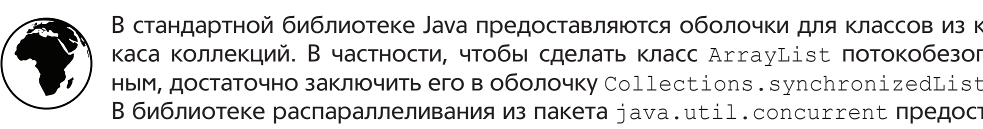
В рассматриваемом здесь примере можно было бы заключить класс `GapTextStore` в оболочку класса, приобретающего блокировку перед доступом к заключенному в него промежуточному хранилищу текста, как показано в приведенном ниже фрагменте кода. Если каждый открытый метод продолжает выполняться таким же образом, как и метод `replace()`, то инварианты промежуточного хранилища текста всегда будут в безопасности, а его клиенты не смогут обойти протокол блокировки.

swt.threads.ThreadSafeTextStore

```
class ThreadSafeTextStore {
    private Lock lock = new ReentrantLock();
```

```
    private GapTextStore rep = new GapTextStore();
    public final void replace(int offset, int length, String text) {
        lock.lock();
        try {
            rep.replace(offset, length, text);
        } finally {
            lock.unlock();
        }
    }
      ... аналогично для методов get() и getLength()
}
```

В подобных небольших примерах издержки на программирование для приобретения и освобождения блокировок не кажутся слишком большими. Однако в реальном коде блокировки защищают более длинные последовательности операций, и поэтому потребность в блоке операторов `try`/`finally` в данном случае не кажется такой докучливой.



В стандартной библиотеке Java предоставляются оболочки для классов из каркаса коллекций. В частности, чтобы сделать класс `ArrayList` потокобезопасным, достаточно заключить его в оболочку `Collections.synchronizedList()`. В библиотеке распараллеливания из пакета `java.util.concurrent` предоставляются также специальные структуры данных (например, в классе `ConcurrentHashMap`), которые являются потокобезопасными, хотя и допускают еще большее распараллеливание, в частности, для обхода коллекций с помощью итераторов. Для хранения одиночных значений потокобезопасным способом служат также классы `AtomicInteger`, `AtomicIntegerArray` и прочие из пакета `java.util.concurrent.atomic`.

Новые объекты промежуточного хранилища текста оказываются потокобезопасными, и поэтому их экземплярами можно совместно пользоваться в разных потоках выполнения:

swt.threads.ThreadSafeRepresentation.main

```
final ThreadSafeTextStore shared = new ThreadSafeTextStore();
```

Для проблематичного доступа к промежуточному хранилищу текста в рассматриваемом здесь примере больше не требуется явная защита.

swt.threads.ThreadSafeRepresentation.main

```
shared.replace(shared.getLength(), 0, text);
```

Вообще говоря, потокобезопасные объекты инкапсулируют не только свои структуры данных, но и их защиту. Их объекты блокировки типа `Lock` становятся частью их внутреннего представления, и это дает возможность сформулировать утверждения об удержании блокировки в какой-то определенный момент. Главное преимущество такого подхода заключается в том,

что клиентам больше не нужно уяснять необходимость блокировки, поскольку ее подробности инкапсулированы и скрыты от них.

Избегайте многоступенчатых операций и межобъектных инвариантов.

Потокобезопасность не является окончательным решением проблем многопоточной обработки и удовлетворяет лишь самым минимальным требованиям. Когда инварианты объектов нарушаются, получить правильное программное обеспечение вообще нельзя. Не следует, однако, забывать, что даже потокобезопасный объект может быть потенциально модифицирован в *промежутке между* двумя последовательными вызовами блокирующих методов. Следовательно, утверждения о состоянии объекта могут быть нарушены.

В качестве наглядного примера рассмотрим самый простой случай, когда символьная строка `text` сначала вводится в потокобезопасное промежуточное хранилище текста в строках 1-2 приведенного ниже фрагмента кода, а затем текст из этой символьной строки сразу же читается обратно в строке 3. Если введенный текст был неожиданно изменен, то в строках 4–7 приведенного ниже кода выводится сообщение об ошибке.

swt.threads.MultiStepBroken.main

```
1 int insertAt = shared.getLength();
2 shared.replace(insertAt, 0, text);
3 String retrieved = shared.get(insertAt, text.length());
4 if (!retrieved.equals(text)) {
5    System.err.println("Unexpected text read back at position "
6              + insertAt + " and step " + i);
7 }
```

При однопоточной обработке состояние промежуточного хранилища текста `shared` можно подробно отслеживать с помощью утверждений. После строки 2 приведенного выше фрагмента кода промежуточное хранилище текста `shared` содержит символьную строку `text` на позиции `insertAt`, и поэтому извлекаемая символьная строка `retrieved` должна быть такой же, как и символьная строка `text`. Но при многопоточной обработке именно это зачастую и не происходит. Если ради эксперимента выполнить приведенный выше фрагмент кода в цикле 10 тысяч раз в двух потоках, то в конечном итоге обнаружится от 20 до 60 нарушений целостности текстовой информации.

↰ 4.7

Блокируйте объекты, упоминаемые в утверждениях.

Как показывают упомянутые выше эксперименты, для потоков выполнения не существует понятия “сразу же после”, а значит, следует всегда допускать, что другой поток может выполняться в самый неподходящий момент. Такой подход очень похож на рекомендацию с подозрением относиться к правильности программного обеспечения.

↰ 4.1
↰ 4.1

Таким образом, всякий раз, когда составляется утверждение о состоянии объекта в контексте многопоточной обработки, необходимо в то же время проверить, удерживается ли блокировка по этому объекту. Ведь утверждения без блокировок всегда нарушаются в какой-то момент из-за неудачно сложившихся взаимодействий потоков выполнения.

В предыдущем примере кода ответственные операции с промежуточным хранилищем текста `shared` выполняются в строка 1–3 данного кода. Именно здесь и делаются допущения о состоянии данного хранилища. В строках 4–7 данного кода, напротив, обрабатываются локальные переменные, которые недоступны из другого потока выполнения. Следовательно, минимально требующаяся блокировка защищает самые ответственные и проблематичные стадии выполнения операций, как показано ниже.

swt.threads.MultiStepCorrect.main

```
int insertAt;
String retrieved;
shared.lock();
try {
    insertAt = shared.getLength();
    shared.replace(insertAt, 0, text);
    retrieved = shared.get(insertAt, text.length());
} finally {
    shared.unlock();
}
```

Обратите внимание на то, что для встраивания текста в промежуточное хранилище `shared` приходится пользоваться блокировкой, иначе другие клиенты данного объектного ресурса могут одновременно модифицировать его. Таким образом, приведенное ниже потокобезопасное хранилище текста типа `ThreadSafeTextStore` экспортирует свою блокировку.

swt.threads.ThreadSafeTextStore

```
public void lock() {
    lock.lock();
}
public void unlock(){
    lock.unlock();
}
```

Итак, мы рассмотрели первое преимущество явных блокировок над оператором `synchronized`. Этот оператор всегда освобождает блокировку в том же самом методе, где и получается блокировка, т.е. оператор `synchronized` поддерживает блочно-структурированную блокировку. Явные объекты блокировки типа `Lock`, напротив, могут удерживаться в течение произвольных периодов времени независимо от блочной структуры программы.

Отсутствие взаимного исключения приводит к трудно выявляемым программным ошибкам.

Следует иметь в виду, что отсутствие блокировки может проявиться не сразу, а оставаться необнаруженным долгое время. Это особенно справедливо, если доступ к общей структуре данных осуществляется нечасто. В подобных случаях программные ошибки могут оставаться не выявленными целыми месяцами, а то и годами. Когда такая ошибка все же проявится, то и тогда ее нелегко отследить и практически невозможно воспроизвести. Поэтому к удерживанию всех необходимых блокировок следует относиться с особым подозрением.

8.3. Обмен уведомлениями между потоками выполнения

Ранее мы рассмотрели широкое применение и важность уведомлений по шаблону “Наблюдатель”. Этот замысел обобщается в событийно-ориентированном программном обеспечении таким образом, чтобы поведение объекта можно было понимать как реакцию на события. Поэтому необходимо исследовать связь уведомлений с потоками выполнения. В частности, что произойдет, если при исполнении одного потока обнаружится изменение какого-то состояния или событие, о наступлении которого следует уведомить другой поток выполнения, даже если он и ожидает такое событие? В отличие от обычного шаблона, нельзя просто вызвать метод в потоке выполнения и передать ему некоторый событийный объект. Вместо этого программное обеспечение должно включать в себя механизмы синхронизации работы потоков выполнения в отдельные моменты времени.

↰ 2.1

Обратимся к примеру простого приложения, чтобы стал понятнее механизм обмена уведомлениями между потоками выполнения (рис. 8.5, *а*). В этом приложении пользователь вводит число, которое обрабатывается в фоновом потоке выполнения, а результат его обработки появляется в списке, расположенном ниже поля ввода. На рис. 8.5, *б*, схематически показана последовательность необходимых шагов, которые следует предпринять в данном приложении для обработки введенного числа. В данном примере приложения задействованы два потока выполнения: поток диспетчериза-

ции событий SWT и фоновый рабочий поток. Новая трудность возникает в связи с синхронизацией на шаге 2 (см. рис. 8.5, *б*), где введенное число передается через границу между потоками выполнения.

↰ 7.10.1

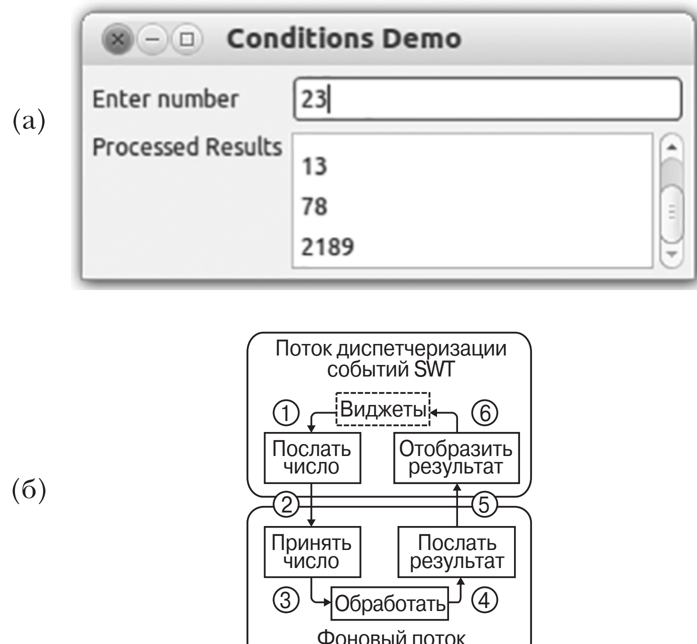


Рис. 8.5. Демонстрация механизма условий

Но давайте сначала проанализируем другие шаги, чтобы составить полное и ясное представление о процессе многопоточной обработки в данном примере. На шаге 1, когда пользователь нажимает клавишу <Enter> в поле ввода текста, приемник событий направляет введенное число на обработку, как показано ниже.

swt.threads.ConditionsDemo.ConditionsDemo

```
number.addSelectionListener(new SelectionAdapter() {
    public void widgetDefaultSelected(SelectionEvent e) {
        int input = Integer.parseInt(number.getText());
        number.setText("");
        sendNumber(input);
    }
});
```

В приведенном выше примере кода обработка введенного числа завершается на уровне пользовательского интерфейса перед его отправкой на дальнейшую обработку. Если оператор с вызовом метода `number.setText()` оставить последним в данном коде, результат может быть получен еще до очистки первого числа. Хотя в данном конкретном примере это и не имеет особого значения, все же лучше соблюсти ясную последовательность действий, завершив извлечение введенного числа из пользовательского интерфейса, прежде чем приступить к его обработке.

↰ 8.1

На шаге 3 (см. рис. 8.5, *б*) вызывается метод `processNumber()`, который не выполняет никакой операции, а только передает введенные данные далее для дальнейшей обработки на шаге 4:

swt.threads.ConditionsDemo

```
public void processNumber(Integer number) {
    Integer result = number; // псевдообработка
    sendResult(result);
}
```

На шаге 5 из рабочего потока выполнения доступ к пользовательскому интерфейсу, как обычно, осуществляется через метод `asyncExec()`. Шаг 6 реализован в следующем коде:

↰ 7.10.1

swt.threads.ConditionsDemo.sendResult

```
display.asyncExec(new Runnable() {
    public void run() {
        if (!results.isDisposed()) {
            results.add(result.toString());
        }
    }
});
```

На тот случай, если у вас возникнет вопрос по поводу асимметрии, напомним, что на шаге 5 вторая очередь задействована очень похожими на рассматриваемые далее способами программирования. Это очередь событий SWT, которая остается скрытой в реализации метода `asyncExec()`.

↰ 7.10.1

Итак, осталось решить задачу завершения шага 2 (см. рис. 8.5, *б*). Общий подход к ее решению схематически показан на рис. 8.6. Оба потока выполнения разделяют общую блокировку `dataLock`, *переменную условия* `dataAvailable`, а также простую несинхронизированную очередь `postbox` для

буферизации данных. Короче говоря, переменные условия позволяют посылать сигналы через границы потоков выполнения. При этом происходит следующее.

1. Рабочий поток выполнения продолжает ожидать сигнал о доступности новых данных.
2. Когда пользователь вводит новое число, поток диспетчеризации событий SWT приобретает блокировку для синхронизации.
3. В этом потоке данные размещаются в очереди.
4. Ожидающему рабочему потоку выполнения посылается сигнал о доступности данных.
5. Рабочий поток выполнения приобретает блокировку и далее извлекает вновь поступившие данные из очереди.

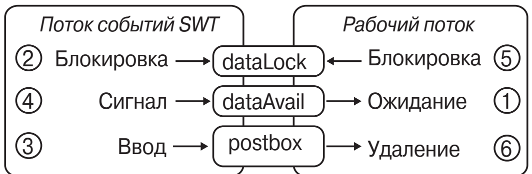


Рис. 8.6. Применение переменных условия для синхронизации потоков выполнения

Итак, мы рассмотрели концептуальную сторону организации многопоточной обработки в данном примере. Ее техническая сторона и способы программной реализации оказываются несколько более сложными вследствие взаимодействий между блокировкой и переменными условия.

> Переменные условия служат для взаимной сигнализации потоков выполнения об изменениях состояния.

В данном примере мы придерживаемся стандартного подхода к шаблону "Поставщик–потребитель" (см. рис. 7.22). В частности, поставщик размещает данные в очереди, откуда потребитель может их извлечь впоследствии. В приведенном ниже фрагменте кода реализуется синхронизация потоков по схеме, приведенной на рис. 8.6. Конкретные данные содержатся в очереди `postbox`, для организации которой выбран простой связный список. Для защиты доступа к этой очереди потребуется блокировка `dataLock`, поскольку данные будут обрабатываться в разных потоках выполнения.

swt.threads.ConditionsDemo

```
private Queue<Integer> postbox = new LinkedList<Integer>();
private Lock dataLock = new ReentrantLock();
private Condition dataAvailable = dataLock.newCondition();
```

Новым элементом в данном примере является переменная условия `dataAvailable`. Эта переменная позволяет приостановить исполнение в одном потоке до тех пор, пока не будет удовлетворено конкретное условие, связанное с состоянием программы, о чем другой поток выполнения сигнализирует через переменную условия после того, как в нем изменится состояние программы. Условие должно быть непременно удовлетворено, поскольку состояние может измениться снова в промежутке между отправкой и обработкой уведомления. Кроме того, в переменной условия не проверяется, насколько обоснован посылаемый сигнал. Переменные условия всегда связаны с блокировками, ведь иначе сами уведомления могут быть подвержены взаимному влиянию и состоянию гонки.

↰ 8.2

В рассматриваемом здесь примере переменная условия `dataAvailable` служит для сигнализации о конкретном состоянии, свидетельствующем о том, что в очереди `postbox` содержатся данные. Когда пользователь нажимает клавишу <Enter> в поле ввода чисел, приемник событий `widgetDefaultSelected()` выполняет приведенный ниже код, где очередь `postbox` блокируется, как обычно, чтобы гарантировать исключительный доступ, а затем новое число размещается в очереди (см. строку 3 приведенного ниже кода). И, наконец, в строке 4 данного кода используется переменная условия `dataAvailable` для отправки любому ожидающему потоку выполнения сигнала о том, что поступили новые данные.

swt.threads.ConditionsDemo.sendNumber

```
1 dataLock.lock();
2 try {
3    postbox.add(number);
4    dataAvailable.signalAll();
5 } finally {
6     dataLock.unlock();
7 }
```

Подобно блокировке, переменная условия формально не связана с конкретной структурой данных или ее состоянием. Выбранное для нее имя отражает принципиальное положение о необходимости поддерживать удобочитаемость исходного кода.

📖 148

В результате вызова метода `signalAll()` активируются все потоки выполнения, ожидающие по заданному условию. Этот метод принято вызывать даже в том случае, если в состоянии ожидания находится единственный поток выполнения, поскольку поступающие сигналы могут легко потеряться, если имеется несколько потоков и если тот, который активируется, не в состоянии обработать поступающий сиг-

нал. Дополнительные меры против потери сигналов способны только усложнить код, а во время выполнения они могут и не принести никаких существенных выгод. Поэтому при программировании многопоточной обработки следует проявлять умеренность.

Обработка чисел, извлекаемых из очереди postbox, продолжается в фоновом потоке выполнения циклически, как показано в приведенном ниже фрагменте кода. Всякий раз, когда этот поток готов обработать следующие введенные данные, он обращается к очереди postbox. Если очередное число еще не поступило, ему придется ждать. Условия позволяют реализовать в этом потоке выполнения именно такое поведение. Поскольку фоновому потоку требуется получить доступ к очереди postbox, то он приобретает соответствующую блокировку (см. строки 1-2, 8–10 приведенного ниже кода). Аналог сигнализации из потока диспетчеризации событий реализован в строках 3-4 того же самого кода, где в фоновом потоке проверяется, содержит ли очередь postbox данные, и если они отсутствуют, фоновый поток переходит в состояние ожидания до тех пор, пока поток диспетчеризации событий не известит его о появлении новых данных. Как только новые данные появятся, они будут обработаны, как обычно, в строках 5–7 приведенного ниже кода.

swt.threads.ConditionsDemo.run

```
1  dataLock.lock();
2  try {
3     while (postbox.isEmpty())
4         dataAvailable.await();
5     do {
6         processNumber(postbox.remove());
7      } while (!postbox.isEmpty());
8  } finally {
9      dataLock.unlock();
10 }
```

Приведенному выше коду присущи три особенности, о которых требуется упомянуть особо. Прежде всего, при вызове метода await() для переменной условия снимается также связанная с ней блокировка. Так, в строке 4 приведенного выше кода снимается блокировка, приобретенная в строке 1. Это необходимо для того, чтобы дать другим потокам выполнения возможность модифицировать общую структуру данных (в данном случае очередь postbox), поскольку для этого им также требуется блокировка. Когда сигнал получен и все готово к возврату из метода await(), блокировка приобретается снова. Так, если метод await() вызывается из нескольких потоков выполнения, исполнение продолжает первый поток, приобретший блокировку, а остальные продолжают ожидать до тех пор, пока блокировка не станет снова доступной.

С этим связана вторая особенность, которая состоит в том, что весь код в строках 1–9 рассматриваемого здесь кода защищен блокировкой с точки зрения потока выполнения. Так, если в результате проверки в строке 3 дан-

ного кода обнаруживается, что новые данные появились и доступны, такое утверждение не может быть нарушено последующим взаимным влиянием, и поэтому строка 6 данного кода вполне обоснована, исходя из того предположения, что вызов метода `remove()` не завершится неудачно. Для демонстрации именно этой особенности и был выбран цикл `do-while`.

↰ 8.2

И, наконец, требует пояснения цикл `while` в строке 3 рассматриваемого здесь кода. Возможно, вместо него было бы достаточно употребить условный оператор `if`. Ведь интуиция подсказывает следующее рассуждение: "Ожидать, *если* не требуется обрабатывать данные". Но в таком рассуждении предполагается, что возврат из метода `await()` происходит лишь тогда, когда данные доступны для обработки. Тем не менее существует ряд причин, по которым этого может и не произойти. Хотя не все эти причины проявляются в текущем контексте, они все же поясняют общий замысел воспользоваться циклом `while`. Во-первых, несколько потоков выполнения могут ожидать одновременно, и обработать все данные сможет только тот поток, который первым снова приобретет блокировку после возврата из метода `await()`. Остальные потоки, активировавшись, обнаружат, что данные отсутствуют, и поэтому им не удастся обработать их в строке 6 данного кода. Во-вторых, условие доступности данных может наступить неумышленно, особенно если переменная условия используется многими клиентами или даже является открытой. И, наконец, на платформе Java не гарантируется полностью возврат из метода `await()` только тогда, когда вызван соответствующий метод `signal()`. *Ложные активации* потоков выполнения могут непроизвольно привести к возврату из метода `await()`. Поскольку при программировании многопоточной обработки рекомендуется проявлять умеренность, то для ожидания требуемого условия следует организовать цикл.

Таким образом, в рассматриваемом здесь примере воплощен основной механизм обмена уведомлениями между потоками выполнения, когда один из них пользуется условиями для отправки конкретного сигнала и задействует структуры данных для хранения событийных объектов или других данных, связанных с уведомлениями. В этом отношении условия можно рассматривать в качестве шаблона для предопределенного межпотокового наблюдателя.

> Пользуйтесь предопределенными стандартными блоками для синхронизации везде, где это только возможно.

Такая методика отправки и получения уведомлений весьма распространена. В стандартной библиотеке Java представляется ряд *блокирующих очередей*, инкапсулирующих всю требующуюся блокировку и сигнализацию. Эти очереди называются так потому, что они блокируют исполнение читающих потоков, когда данные отсутствуют, а также исполнение записывающих по-

токов, когда емкость очереди исчерпана. Блокирующие очереди предоставляют достаточные функциональные возможности для надежного обмена элементами данных или событиями между потоками выполнения.

↰ 8.2

Таким образом, блокирующие очереди значительно упрощают программирование многопоточной обработки. В частности, простую очередь `postbox` из предыдущего примера можно заменить синхронизированной очередью:

swt.threads.BlockingQueueDemo

```
private BlockingQueue<Integer> postbox =
        new LinkedBlockingQueue<Integer>();
```

Всякий раз, когда пользователь вводит данные, в потоке диспетчеризации событий они размещаются в очереди. Все вопросы организации сообщения между потоками разрешаются внутренним механизмом блокирующей очереди.

swt.threads.BlockingQueueDemo.sendNumber

```
postbox.put(Integer.parseInt(number.getText()));
number.setText("");
```

Аналогично в фоновом потоке выполнения из очереди просто извлекается следующий элемент данных. Выполнение приведенного ниже метода `take()` блокируется до тех пор, пока данные не будут доступны для обработки.

swt.threads.BlockingQueueDemo.run

```
processData(postbox.take());
```

↰ 7.10.1, 📖 148

Блокирующие очереди являются основными составляющими для реализации отношений "поставщик–потребитель" между потоками выполнения, когда один поток поставляет данные, размещая их в очереди, а другой поток извлекает данные из очереди, обрабатывая их. В подобных случаях следует принимать во внимание еще одну особенность синхронизации: скорость обработки данных. Если данные поставляются намного быстрее, чем их способен обработать потребитель, то в конечном итоге оперативная память переполняется элементами данных, хранящимися в очереди. Именно по этой причине блокирующие очереди из стандартной библиотеки Java имеют фиксированную *емкость*. Когда поставщик вызывает метод `put()`, а очередь уже заполнена, этот вызов блокируется, и такое поведение оказывается симметричным вызову метода `take()` для пустой очереди. В итоге скорость обработки данных поставщиком снижается до скорости обработки данных потребителем, и переполнения оперативной памяти не происходит.

В приведенном выше примере очередь типа `LinkedBlockingQueue` имела неограниченную емкость, зеркально отражая поведение предыдущей очереди `postbox` и тем самым гарантируя, что вызов метода `put()` не заблокирует поток диспетчеризации событий, который, в свою очередь, затормозит работу пользовательского интерфейса, хотя это и особый случай. При обычных отношениях "поставщик–потребитель" предпочтение следует отдавать очередям с ограниченной емкостью.

В стандартной библиотеке Java поддерживает решение и других типичных задач синхронизации. Так, класс `CountDownLatch` организует ожидание в одном потоке выполнения до тех пор, пока в других потоках не будет достигнут какой-то определенный момент, например, начало или завершение выполнения разных частей вычисления во всех остальных потоках. Класс `CyclicBarrier` позволяет потокам выполнения неоднократно ожидать друг друга в конкретные моменты, когда, например, требуется произвести обмен данными. Настоятельно рекомендуется просмотреть данную библиотеку, прежде чем пытаться реализовать собственный механизм синхронизации потоков выполнения.

📖 148

Останавливайте потоки выполнения, используя метод `interrupt()`.

Последнее уведомление, которым обмениваются потоки выполнения, имеет непосредственное отношение к их завершению. Допустим, что в пользовательском интерфейсе запущен фоновый поток выполнения, но ожидаемые результаты оказались излишними вследствие дополнительных действий пользователя. Как уведомить фоновый поток о таком событии? На платформе Eclipse это осуществляется с помощью мониторов текущего состояния. На уровне самих потоков выполнения этой цели служат *прерывания*.

↰ 7.10.2

В рассматриваемом здесь примере обработки вводимых чисел фоновый поток должен быть остановлен, когда закрывается диалоговое окно. С этой целью к диалоговому окну присоединяется приемник событий типа `DisposeListener`, а исполнение фонового потока прерывается (см. строку 5 приведенного ниже фрагмента кода).

swt.threads.ConditionsDemo.ConditionsDemo

```
1 workerThread = new Thread(worker);
2 workerThread.start();
3 addDisposeListener(new DisposeListener() {
4    public void widgetDisposed(DisposeEvent e) {
5        workerThread.interrupt();
```

```
6    }
7 });
```

Прерываемый поток выполнения может получить прерывание двумя возможными способами. Когда он заблокирован в ожидании какого-нибудь сигнала или условия, например, через вызов метода `await()` для переменной типа `Condition` или метода `take()` для объекта типа `BlockingQueue`, возврат из вызванного метода происходит с исключением типа `Interrupted Exception` (возможно, после ожидания повторного приобретения любых снятых блокировок). В противном случае во встроенном в поток *признаке состояния прерывания* будет установлено логическое значение `true`.

В приведенном ниже цикле обработки из рассматриваемого здесь примера демонстрируется типичная методика обработки прерываний. Прежде чем начать какую-нибудь другую работу, в потоке прерывания проверяется, не был ли он прерван (см. строку 2 приведенного ниже кода). В противном случае поток ожидает дополнительные данные и обрабатывает их. Кроме того, во время ожидания новых данных в нем предполагается генерирование исключения типа `InterruptedException`. В строках приведенного ниже кода демонстрируется методика преобразования исключения в устанавливаемый признак состояния прерывания, сочетающая в себе два упомянутых выше способа получения прерывания. В данном примере прерывается сначала цикл, а затем поток выполнения.

📖 148

swt.threads.ConditionsDemo

```
1 public void run() {
2    while (!Thread.currentThread().isInterrupted()) {
3       try {
4             ...
5       } catch (InterruptedException exc) {
6          // преобразовать в признак
7          Thread.currentThread().interrupt();
8       }
9    }
10 }
```

Таким образом, поток выполнения прерывается сам, а не извне. Прерывание, посылаемое другим потоком, следует рассматривать как своего рода запрос. В частности, получатель прерывания должен быть устойчивым к возможным исключениям типа `InterruptedException`. Это означает, что он должен закрыть все открытые ресурсы и оставить невредимыми инварианты структур данных.

↰ 1.5.6

Не игнорируйте исключения типа `InterruptedException`. При написании кода такие исключения нередко оказываются лишь досадной помехой, устранение которой откладывается на потом, а для избавления от них вводится пустой блок оператора `catch`. Следует, однако, иметь в виду, что правильное завершение потока выполнения является такой же важной составляющей его жизненного цикла и поведения, как и сама обработка, которая в нем выполняется. Надлежащая обработка прерываний типа `InterruptedException` является частью требований, предъявляемых к многопоточной обработке.

Непременно выясните, какие из блокирующих методов являются прерываемыми. Например, метод `Lock.lock()` блокирует поток выполнения до тех пор, пока не станет доступной блокировка, но он *не* завершается исключением типа `InterruptedException`, если между тем поток выполнения получает прерывание. Дело в том, что длительные блокировки обычно не предполагаются, и поэтому такое поведение удобно как стандартное. Если же в приложении допускается более длительное ожидание, то вместо этого, возможно, имеет смысл воспользоваться методом `lockInterruptibly()`.

Способность к блокировке вместе с приемом прерываний и способность к блокировке с заданным временем ожидания являются еще двумя особенностями, благодаря которым объекты типа `Lock` оказываются намного более гибкими, чем встроенная команда `synchronized`.

Никогда не вызывайте метод `stop()` для объектов типа `Thread`. Он сразу же прерывает вычисление, не давая возможности произвести очистку в потоке выполнения. В этом случае *не* выполняется даже блок оператора `finally`, который обычно служит для защиты от непредвиденного завершения.

↰ 1.5.6

Непременно сделайте прерываемыми все запущенные потоки выполнения.

Потоки выполнения, которые больше не нужны, должны быть прерваны, чтобы освободить их ресурсы. Еще одной тому причиной служит то обстоятельство, что виртуальная машина Java не завершит свою работу до тех пор, пока не завершатся все запущенные потоки выполнения. Для пользовательских интерфейсов это означает, что недостаточно будет даже закрыть их оболочки и завершить поток диспетчеризации их событий, поскольку нужно будет также ввести приемник событий, наступающих в связи с удалением виджетов, как было показано ранее. Требование останавливать потоки выполнения на самом деле ничем не отличается от общей рекомендации по поводу организации жизненного цикла объекта. Она состоит, в частности, в том, чтобы снимать объект с регистрации во всех субъектах, за которыми он наблюдает, а также освобождать любые ресурсы, которые он может удерживать.

↰ 2.1.2
↰ 7.4.1

Виртуальная машина Java предоставляет потоковые демоны, которые завершаются автоматически, когда завершается последний обычный поток выполнения, не являющийся демоном. Однако потоковые демоны завершаются внезапно без всякого уведомления других потоков, и поэтому метод `Thread.stop()` не следует вызывать и в данном случае.

8.4. Асинхронные сообщения

Как правило, сообщение между объектами состоит в том, что они вызывают методы друг друга. Поскольку вызовы методов всегда связаны с техническими подробностями передачи параметров и организацией стека вызовов, то проще воспользоваться следующей абстракцией: объекты сообщаются друг с другом, обмениваясь сообщениями, даже если отправка сообщения в конечном итоге превращается в вызов метода. При такой абстракции объекты становятся более независимыми друг от друга, когда один из них посылает сообщение, а другой соответственно реагирует на него.

↰ 1.4.1, ↰ 1.1

В контексте потоков выполнения такая независимость может быть упрочена, поскольку объект, получающий сообщение, не должен реагировать на него немедленно, но может отложить его обработку до более удобного момента. Такие сообщения называются *асинхронными*. Как показано на рис. 8.7, объект *A* существует в потоке 1, где и выполняется его код, тогда как объект *B* – в потоке 2. Как правило, объект *B* будет выполнять цикл в ожидании сообщений и вызывать для себя соответствующие методы. Если объекту *A* требуется послать сообщение объекту *B*, он размещает это сообщение в синхронизированной очереди и продолжает свою обработку дальше. В какой-то последующий момент объект *B* примет отправленное ему сообщение и отреагирует на него. Отправителю сообщения не нужно ждать возвращаемого значения, и поэтому любые результаты, как правило, снова передаются асинхронно через вторую очередь, обозначенную пунктирными линиями на рис. 8.7.

↰ 8.3

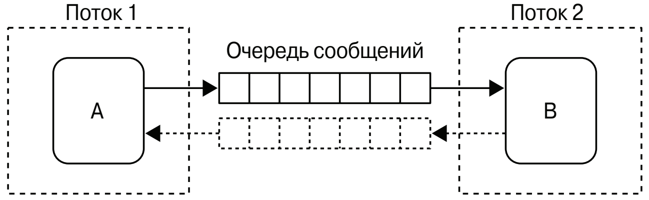


Рис. 8.7. Асинхронные сообщения

В качестве конкретного примера рассмотрим пользовательский интерфейс для загрузки веб-страниц в фоновом режиме. Объект *A* на рис. 8.7 является частью пользовательского интерфейса и существует в потоке диспетчеризации событий. Он вызывает метод `download()` всякий раз, когда требуется загрузить веб-страницу. В строках 2–4 приведенного ниже кода составляется запрос на загрузку, который отображается в таблице. Это требуется для того, чтобы завершить асинхронный обмен сообщениями. Когда поступает ответ на запрос, его результат должен быть отображен в верной строке таблицы.

↰ 7.10.1

swt.threads.AsyncMessages

```
1 public void download(String url) throws InterruptedException {
2    DownloadRequest req = new DownloadRequest(url);
3    TableItem item = new TableItem(table, SWT.NONE);
4    item.setText(0, url);
5    openRequests.put(req, item);
6    requests.put(req);
7 }
```

Результаты, посылаемые через очередь, обозначенную пунктирными линиями на рис. 8.7, получаются в отдельном потоке, где выполняется приведенный ниже цикл, а результаты отображаются в верной строке таблицы (переход к потоку диспетчеризации событий осуществляется, как обычно).

↰ 7.10.1

swt.threads.AsyncMessages.run

```
while (true) {
    final DownloadResult res = results.take();
    final TableItem origin = openRequests.remove(res.request);
    display.asyncExec(new Runnable() {
        public void run() {
            origin.setText(1, res.message);
        }
    });
}
```

📖 218

Такой подход обобщается в шаблоне "Признак асинхронного завершения" (Asynchronous Completion Token) до обмена сообщениями между процессами по сети. В этом случае запрос, к которому относится ответ, не может быть выдан по простой ссылке на объект, как это сделано в предыдущем примере, по-

скольку асинхронное сообщение покидает адресное пространство виртуальной машины Java. Вместо этого каждый запрос содержит однозначный идентификатор (т.е. признак завершения), который получатель запроса пересылает обратно вместе с ответом. Отправитель ведет таблицу открытых запросов, чтобы правильно присваивать им поступающие ответы.

📖 16, 📖 212, 📖 113

В языках функционального программирования, особенно Erlang, ML и Scala, уже давно было обнаружено, что отправка асинхронных сообщений через очереди сообщений сама по себе вполне жизнеспособная модель для параллельного выполнения. Вместо того чтобы вести речь о самих потоках, исполняющих методы объектов, можно рассуждать непосредственно об объектах, иногда называемых в данном контексте *исполнителями* и обменивающимися сообщениями по отдельным *каналам*. Тот факт, что асинхронные сообщения способны поддерживать целые модели программирования, ясно показывает их практическую применимость.

Асинхронные сообщения вносят дополнительную сложность, поскольку не приходится, как обычно, ожидать значения, возвращаемого из вызова метода. Но иногда такую сложность можно скрыть. Общее представление об этом дает шаблон “Активный объект” (Active Object), как поясняется ниже.

📖 218

Шаблон проектирования: Активный объект

Шаблон “Активный объект” предоставляет абстракцию над обработкой асинхронных сообщений (рис. 8.8), инкапсулируя подробности организации очередей и многопоточной обработки. В активном объекте имеется *служащий*, предоставляющий реальные функциональные возможности, а снаружи он действует как *заместитель* этого служащего. Он запускает поток *планировщика*, постоянно извлекающего вызовы методов из *очереди активации*, и вызывает соответствующий метод из *служащего*.

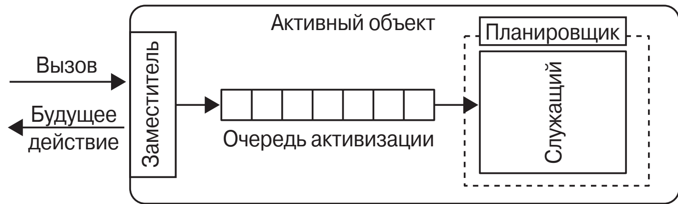


Рис. 8.8. Шаблон “Активный объект”

Клиенты активных объектов просто вызывают методы для его заместителя, зная, что вызовы будут обработаны асинхронно в фоновом режиме. Активные объекты могут поддерживать такую абстракцию вызовов методов для возвращаемых значений, применяя так называемые *будущие действия*,

которые служат контейнерами для вычисляемых в дальнейшем значений. Методы активного объекта немедленно возвращают будущее действие, а клиент может в дальнейшем получить результат будущего действия, вызвав для него метод `get()`. Если обработка еще не завершена, вызов блокируется. В состав стандартной библиотеки Java входит определенный ряд реализаций будущих действий, предназначенных для применения вместе с каркасом исполнителей.

📖 148

8.5. Открытые вызовы для уведомления

Пользовательский интерфейс в частности и событийно-ориентированное программное обеспечение вообще в значительной степени опирается на уведомления и, в частности, на шаблон “Наблюдатель”. В связи с потоками выполнения возникает основной вопрос: следует ли посылать уведомления при удержании блокировки или после его снятия? Уведомления, посылаемые без удержания блокировки, называются *открытыми вызовами*. Рассмотрим последствия такого ослабления блокировки и его отличия от обычного шаблона “Наблюдатель”.

↰ 2.1
📖 148

На рис. 8.9 приведена типичная ситуация, когда прикладные данные хранятся в каком-нибудь потокобезопасном объекте, чтобы их можно было обрабатывать в фоновых потоках выполнения, вызывая доступные методы для выполнения соответствующих операций. В пользовательском интерфейсе предпринимается попытка вовремя отразить состояние данных на экране. С этой целью в нем регистрируется наблюдатель, а обновление отображения происходит в соответствии с сообщаемыми изменениями.

↰ 8.2
⏩ 9.1

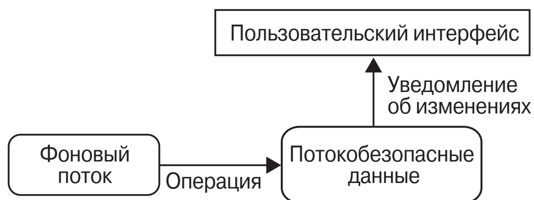


Рис. 8.9. Отражение потокобезопасных данных в пользовательском интерфейсе

В качестве конкретного примера рассмотрим приведенный ниже класс ThreadSafeData, в котором поддерживается единственное значение data, представляющее более крупную структуру данных. Самый первый и очевидный способ реализовать шаблон (в данном случае — "Наблюдатель") состоит в том, чтобы защитить блокировкой как саму модификацию, так и уведомление о ней. Благодаря этому весь блок операций будет вести себя таким же образом, как и при однопоточной обработке.

swt.threads.ThreadSafeData

```
1 public void setValueSync(int value) {
2    lock.lock();
3    try {
4        int oldData = this.value;
5        this.value = value;
6        changes.firePropertyChange("data", oldData, value);
7    } finally {
8        lock.unlock();
9    }
10 }
```

Если обратиться к пользовательскому интерфейсу, то можно обнаружить, что такая строгая дисциплина блокировки может в действительности оказаться ненужной. Ведь в пользовательском интерфейсе регистрируется приемник событий вместе со структурой данных, осуществляется переход к потоку диспетчеризации событий через метод asyncExec() и обновляется текстовое поле value новым содержимым структуры данных (см. строки 6-7 приведенного ниже кода).

swt.threads.ReflectThreadSafeData.setThreadSafeData

```
1 public void propertyChange(final PropertyChangeEvent evt) {
2     display.asyncExec(new Runnable() {
3         public void run() {
4             if (isDisposed())
5                 return;
6             int curValue = data.getValue();
7             value.setText(String.format("%03d", curValue));
8             int reportedValue = (Integer) evt.getNewValue();
9             state.setText(reportedValue == curValue ? "==" : "!=");
10        }
11    });
12 }
```

В строках 8-9 приведенного выше кода раскрывается главная особенность рассматриваемого здесь примера. В них проверяется, осталось ли значение data таким же, как и то, о котором сообщалось в событии об изменении, т.е. то значение, для которого и было первоначально отправлено событие. Но этого обычно не происходит, когда модификация общей структуры данных продолжается в фоновых потоках выполнения. В строке 2 приведенного

выше кода выполнение заданного исполняемого кода планируется в потоке диспетчеризации событий в какой-то последующий момент времени, поэтому код в строке 4 начинает выполняться не сразу, а до тех пор в этом потоке имеется немало времени для модификации данных (см. также рис. 7.21).

Уведомления с блокировками предоставляют строгие гарантии.

Теперь допустим, что к общей структуре данных присоединен еще один приемник событий, как показано ниже. Он не зависит от пользовательского интерфейса и служит лишь для проверки неизменности данных при получении обратного вызова. В действительности это имеет место всегда, поскольку блокировка в методе `setValueSync()` предотвращает изменения в промежутке между выполнением кода в строках 5- 6 данного метода.

swt.threads.DifferencingObserver

```
public void propertyChange(PropertyChangeEvent evt) {
    int reportedValue = (Integer) evt.getNewValue();
    int actualValue = data.getValue();
    if (reportedValue != actualValue)
        System.out.format("detected difference %03d != %03d\n",
               reportedValue, actualValue);
}
```

С более общей точки зрения отправка уведомлений при удержании блокировки гарантирует, что наблюдатели заметят именно то состояние, которое установилось в результате единственной модификации. Это такое же поведение, как и в однопоточной среде. Но столь строгая гарантия приобретается за счет того, что ни один из других потоков выполнения не сможет обратиться к общей структуре данных до тех пор, пока не завершат свою работу все наблюдатели.

Открытые вызовы способны снизить соперничество за блокировки.

148

Рассматриваемый здесь пример пользовательского интерфейса показывает также, что во многих случаях наблюдатели не слишком полагаются на то, что состояние окажется точно таким же, как и после изменения. Это означает, что они все равно будут обращаться к структуре данных, чтобы извлечь самые последние значения, и поэтому совершенно не важно, изменились ли они тем временем. Следовательно, метод установки в классе `ThreadSafeData` можно переделать таким образом, чтобы перед отправкой уведомлений в строке 10 приведенного ниже кода блокировка снималась в строке 8, а данные можно было обрабатывать в других потоках выполнения.

swt.threads.ThreadSafeData

```
1 public void setDataOpen(int value) {
2     lock.lock();
3     int oldValue;
4     try {
5         oldValue = this.value;
6         this.value = value;
7     } finally {
8         lock.unlock();
9     }
10    changes.firePropertyChange("data", oldValue, value);
11 }
```

В общем, *открытые вызовы* являются уведомлениями, которые посылаются, будь то для извещения наблюдателей или обращения к произвольным взаимодействующим объектам, без удержания блокировки по завершении конкретной операции. Если структура данных занимает центральное положение в приложении и к ней часто обращаются из разных потоков выполнения, то замена уведомлений на открытые вызовы может повысить эффективность приложения, поскольку сокращается число потоков, которым приходится ожидать блокировки, а следовательно, снижается соперничество за блокировки. Открытые вызовы помогают также предотвратить *взаимные блокировки* — явление, которое поясняется в следующем разделе.

⏩ 8.6

Открытые вызовы способны поставить под угрозу правильность кода.

Допустим, что структура данных представляет собой список и по шаблону “Наблюдатель” выбрана модель *передачи*, когда субъект посылает подробные сведения об изменениях, например, “введен элемент на позиции 5”. Но когда наблюдатель получит наконец открытый вызов, этот элемент может быть уже перемещен в какое-нибудь другое место или же индекс 5 может оказаться недействительным, поскольку список тем временем опорожнился.

↰ 2.1.3
↰ 8.2, ↰ 4.1

По существу, такое поведение снова отражает нарушение утверждений вследствие взаимного влияния. В конце операции со списком индекс был действительным, поскольку утверждение о том, что индекс находится в пределах списка, было защищено блокировкой. Но как только блокировка была снята, это утверждение могло быть нарушено.

Пользуйтесь открытыми вызовами лишь в том случае, если наблюдатель не собирается полагаться на более строгие гарантии.

Открытые вызовы рассматриваются здесь, главным образом, для того, чтобы указать на трудности их применения, а также обсудить решение о необходимости удерживать или освобождать блокировку для уведомлений. Открытые вызовы могут оказаться удобными в интенсивно используемых структурах данных, но они требуют, чтобы всем возможным их получателям было известно, что принятое ими сообщение может оказаться уже неактуальным.

Открытые вызовы очень похожи на асинхронные сообщения в том отношении, что прием сообщения может быть также задержан до такой степени, что данные, которых оно касается, тем временем изменятся. Как свидетельствует огромный успех управляемых сообщениями систем вроде исполнителей в языке Erlang, открытые вызовы все-таки происходят в приложениях, критичных к производительности.

↰ 8.4
📖 16

8.6. Взаимные блокировки

В связи с потоками выполнения возникает еще одно затруднение, которое требуется обсудить. Речь в разделе пойдет о *взаимных блокировках*. Обычно этим термином обозначают ситуации, когда один или несколько потоков не могут выполняться дальше, поскольку условие, которого они ожидают, может вообще не наступить. В частности, взаимные блокировки происходят в том случае, когда одни потоки выполнения не могут получить требующиеся им блокировки, поскольку другие потоки удерживают эти блокировки и сами не в состоянии выполняться дальше.

Взаимные блокировки легко возникают в операциях, где задействованы многие объекты.

Проиллюстрируем явление взаимные блокировки в контексте многоступенчатых операций. В таких операциях нередко приходится неоднократно получать блокировки по разным объектам из набора, общего для всех потоков выполнения, поскольку они могут исключить взаимное влияние только в том случае, если им удастся удерживать все необходимые блокировки на все время выполнения операции.

↰ 8.2

Итак, продолжим обсуждение явления взаимные блокировки на простом примере, где выполняются операции, которые просто копируют отдельные фрагменты текста из одного потокобезопасного хранилища типа `ThreadSafeTextStore` в другое. По существу, они выполняют следующие действия (см. строки 4–6 приведенного ниже кода): выявление копируемого фрагмента кода, получение отдельных символов текста из исходного хранилища `src` и их запись в целевое хранилище `dst`. Оба хранилища совместно используются в разных одновременно выполняемых операциях. Следовательно, утверждения о диапазоне символов, установленные в строке 4 приведенного ниже кода, подтверждаются в строках 5-6 только в том случае, если вся последовательность символов защищена блокировками.

↰ 8.2

swt.threads.DeadlocksBroken.main

```
1 src.lock();
2 dst.lock();
3 try {
4      ... определить позиции и длину текста
5    String text = src.get(srcOffset, srcLength);
6    dst.replace(dstOffset, 0, text);
} finally {
8     src.unlock();
9     dst.unlock();
10 }
```

Весь рассмотренный выше код выглядит нормально, и ни одно из утверждений не может быть даже нарушено вследствие взаимного влияния. Однако вполне возможно, что операции будут мешать друг другу в получении необходимых блокировок в строках 1-2 данного кода. Допустим, что в ходе двух операций текст копируется из одного буфера в другой в следующей последовательности: в одной операции текст копируется из буфера `a` в буфер `b`, а в другой — в обратном направлении. (В потоке выполнения `Worker` это действие выполняется неоднократно с помощью метода `run()`.)

swt.threads.DeadlocksBroken.main

```
new Thread(new Worker(a, b)).start();
new Thread(new Worker(b, a)).start();
```

Далее между потоками выполнения происходит новое взаимодействие. Как и предполагалось, оно начинается с блокировки как исходного, так и целевого буфера следующим образом:

swt.threads.DeadlocksBroken.main

```
1 src.lock();
2 dst.lock();
```

Однако оба потока выполнения блокируют буфера в разном порядке. В частности, первый поток сначала блокирует буфер `a`, а затем буфер `b`, тогда как второй поток сначала блокирует буфер `b`, а затем буфер `a`. В итоге каждый поток может приобрести в строке 1 приведенного выше фрагмента кода именно ту блокировку, которую другой поток пытается приобрести в строке 2. По существу, эти потоки выполнения препятствуют друг другу продолжить выполнение кода дальше строки 2, и поэтому работа приложения или, по крайней мере, какого-нибудь из его фоновых потоков полностью стопорится.

Как и прежде, подобное осложнение может произойти, а может и нет, в зависимости от точности синхронного выполнения кода в обоих потоках. Так, если в одном потоке удастся продвинуться дальше строк 1-2 приведенного выше фрагмента кода, тогда как в другом потоке выполняется что-нибудь другое, то все будет в порядке. Подобные программные ошибки очень трудно выявлять и устранять, поскольку их проявление зависит от точности синхронного выполнения кода и решений, принимаемых планировщиком потоков выполнения.

↤ 8.1

Избегайте взаимных блокировок, упорядочивая ресурсы.

Сущность взаимных блокировок кроется в разном порядке приобретения блокировок в потоках выполнения. Если бы оба упомянутых выше потока выполнения блокировали сначала буфер `a`, а затем буфер `b`, то ничего плохого бы не произошло, какой бы из этих буферов ни считался исходным или целевым в каждом из потоков. Ведь тот поток, который первым заблокирует буфер `a`, перейдет далее к блокировке буфера `b`, поскольку другой поток по-прежнему ожидает своей блокировки сначала буфера `a`. Такой простой на первый взгляд способ упорядочения блокируемых ресурсов пригоден и для обращения не только с двумя, но и с б*о*льшим количеством объектов и называется *упорядочением ресурсов*.

📖 148

Чтобы стала понятнее суть упорядочения ресурсов, допустим, что обращение к одной коллекции объектов происходит из многих потоков выполнения, как показано на рис. 8.10, *а*. Каждый объект в этой коллекции защищен отдельной блокировкой, которую поток должен приобрести, прежде чем работать с объектом. Пунктирными линиями на рис. 8.10, *а*, обозначен порядок, в котором потоку требуется приобрести нужные блокировки. В данном примере во всех потоках выполнения в какой-то момент потребуется объект, находящийся в центре рис. 8.10, *а*, а иначе им будут доступны

разные объекты, и лишь некоторыми из этих объектов они смогут пользоваться совместно с другими потоками выполнения.

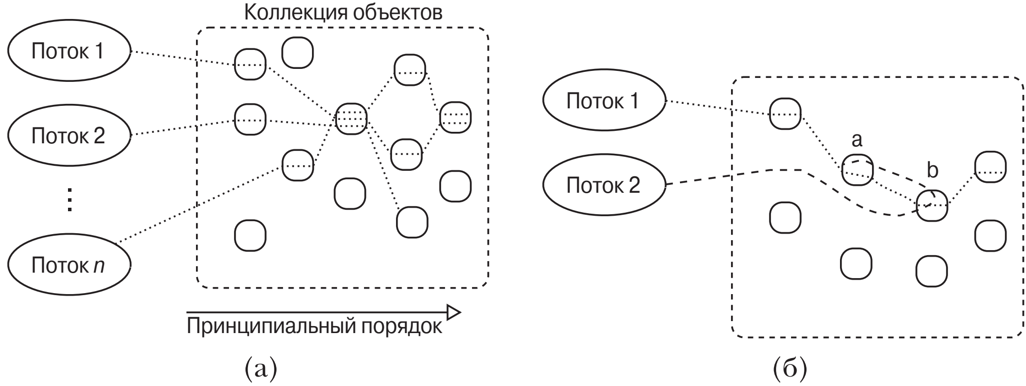


Рис. 8.10. Схематическое представление упорядочения ресурсов

Как следует из рис. 8.10, взаимные блокировки исключаются, если потоки выполнения всегда блокируют объекты слева направо. Для возникновения взаимные блокировки требуется ситуация, аналогичная приведенной на рис. 8.10, *б*, где поток 1 блокирует объект a, а поток 2 — объект b, после чего поток 2 меняет порядок блокировки справа налево, чтобы заблокировать объект a. Но этого не произойдет, если блокировка продолжится слева направо.

Графическое представление порядка блокировки слева направо можно уточнить, приведя объекты к некоторому принципиальному порядку, если выразить формулировку "a слева от b" отношением $a < b$. На рис. 8.10 предполагается, что *линейный порядок* (например, натуральных чисел) совсем не обязательно выбирать для упорядочения объектов строго слева направо. Вполне допустимо, чтобы два объекта занимали одно и то же положение по горизонтали, что приводит к *частичному порядку*, где отношения $a \leq b$ и $b \leq a$ подразумевают, что $a = b$, т.е. нельзя сравнивать два разных объекта в одном и том же положении по горизонтали.

Как же установить такой порядок для объектов? Для этой цели подойдет любой частичный порядок, и поэтому в нашем распоряжении фактически имеются следующие варианты выбора.

- Воспользоваться каким-нибудь внутренним атрибутом заблокированного объекта. Например, при перемещении денежных средств с одного банковского счета на другой можно упорядочить банковские счета по их номерам.

🕮 148 (§2.4.5), ↰ 2.2.1

- Прибегнуть к *блокировке иерархической вложенности*, воздействующей на владение структуры объектов, всегда блокируя владельцев пре-

жде принадлежащих им частей. Такая стратегия особенно удобна, если блокировка частей защищает также их инварианты, поскольку в этом случае вызов метода блокируется в правильном порядке от владельца к его части.

🕮 148 (§2.2.6)

- Воспользоваться значением, которое возвращается методом `System.identityHashCode()` и обычно основывается на адресе объекта. Хотя однозначность такого значения не гарантируется, тем не менее, неудачный исход маловероятен.

Каким бы в конечном итоге ни оказался порядок блокировки, он гарантирует предотвращение взаимных блокировок во всех возможных случаях таким же образом, как и верное рассуждение о контрактах — правильность работы программного обеспечения при всех возможных обстоятельствах.

↰ 4.7

Предусмотреть все объекты заранее нелегко.

Глядя на рис 8.10, *а*, можно, прежде всего, косвенно допустить, что имеется вполне определенная совокупность объектов. Например, в самом общем случае иерархической блокировки известен владелец и его части. Кроме того, в составных операциях владельца учитывается, какие именно части придется блокировать. Однако иногда порядок блокировки объектов в многоступенчатой операции становится очевидным лишь во время выполнения.

↰ 2.1

Одним из примеров тому служит ситуация, когда наблюдатели принимают уведомления об изменениях состояния и должны реагировать на них. Субъекту, посылающему уведомления, будут известны не все наблюдатели, да и вряд ли разные наблюдатели будут известны друг другу. Если субъект удерживает любые блокировки, посылая в то же время уведомления, это, скорее всего, приведет к взаимной блокировке, поскольку наблюдатели обычно возвращаются к тем частям субъекта или взаимодействующим с ним объектам, которые могут быть тем временем заблокированы в других потоках выполнения. В этом случае может помочь установление определенного порядка доступа к ресурсам для этих конкретных объектов, если все наблюдатели способны полагаться на него и подчиняться ему. В других случаях придется предпринять уклончивые действия.

↰ 8.5

Открытые вызовы способны предотвратить взаимные блокировки, поскольку в данном случае субъект уведомляет наблюдателей только после снятия всех своих блокировок. Хотя наблюдатели должны снова запрашивать текущее состояние субъекта после получения своих блокировок, этот недостаток кажется незначительным в данном контексте по сравнению с возможностью избежать взаимных блокировок.

↰ 148

Другой, пессимистический подход состоит в том, чтобы предположить возникновение взаимных блокировок в любом случае, когда происходят обратные вызовы, по существу, произвольного кода, который может находиться в разных модулях крупной системы. Каждый из участвующих в подобном сценарии развития ситуации может подготовиться к худшему, всегда вызывая метод `lock()` с блокировкой по времени и допуская, что взаимная блокировка возникнет по истечении времени ожидания. Тогда отдельный участник освободит свои блокировки, перейдя в состояние ожидания на произвольно короткое время, чтобы дать возможность другим участникам завершить свои операции, прежде чем пытаться снова получить все блокировки.

Умеренно пользуйтесь блокировками.

Зачастую можно выбрать один из следующих вариантов блокировки: связать каждую блокировку с отдельным объектом в группе или же единую блокировку со всей группой. Первый вариант предпочтителен из соображений производительности, поскольку вполне возможно, что несколько потоков выполнения могут параллельно обрабатывать разные подмножества объектов. Второй вариант, очевидно, будет предпочтительным потому, что он полностью исключает взаимные блокировки.

↰ 1.1

В конечном итоге возникает следующий вопрос оптимизации: стоит ли идти на издержки разработки и навлекать риск возникновения взаимных блокировок ради потенциальных выгод от повышения производительности? Как и при любой другой оптимизации, следует делать “самое простое, что только может работать”, до тех пор, пока не станет очевидной потребность в более сложном решении. Многие годы структуры данных ядра FreeBSD были защищены единственной глобальной “гигантской блокировкой”, и тем не менее, эта операционная система прославилась своей высочайшей производительностью.