Многопоточное программирование в Java

Тимур Машнин

Тимур Машнин

Многопоточное программирование в Java

«Издательские решения»

Машнин Т.

Многопоточное программирование в Java / Т. Машнин — «Издательские решения»,

ISBN 978-5-00-531464-2

В многопроцессорных системах многопоточность решает проблему параллельного выполнения кода с наименьшими затратами.Поэтому многопоточность используется в большинстве реальных приложений.И Java, как и большинство языков программирования, поддерживает многопоточность.Познакомьтесь с реализацией процессов и потоков в Java, с управлением и синхронизацией потоков.Узнайте о пуле потоков, потокобезопасных коллекциях, синхронизаторах и параллельных потоках Stream.

ISBN 978-5-00-531464-2

Содержание

Многопоточное программирование в Java

Тимур Машнин

ISBN 978-5-0053-1464-2
Создано в интеллектуальной издательской системе Ridero

Процессы и потоки

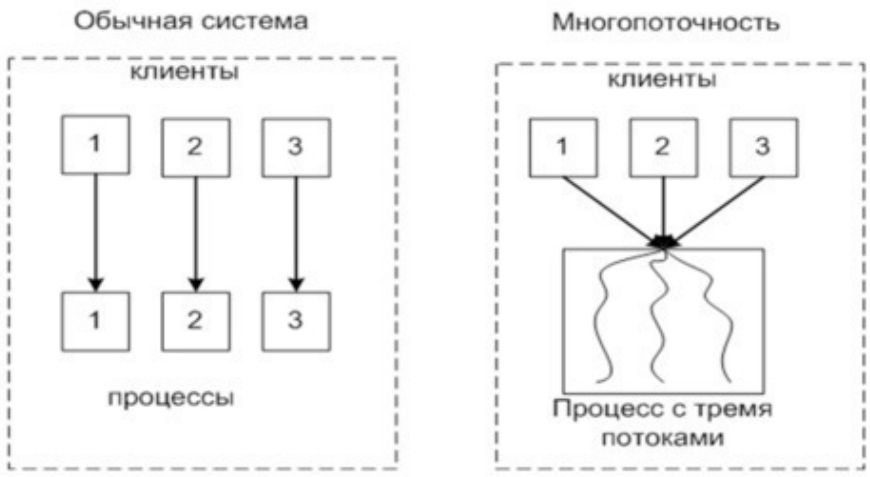
Tech Solutions

Многопоточное программирование в Java

Процессы и потоки

Чтобы начать работу с основами многопоточного программирования, давайте начнем с изучения потоков.

Каждая операционная система поддерживает потоки в той или иной форме.



Вначале, все усилия по повышению производительности процессоров были направлены на наращивание тактовой частоты, но со все большим увеличением частоты, наращивать её стало тяжелее, так как это требовало увеличения охлаждения процессоров.

Поэтому инженеры стали добавлять ядра в процессор, так и возникли многоядерные процессоры.

Принцип увеличения производительности процессора за счёт нескольких ядер, заключается в разделении выполнения потоков или различных задач на несколько ядер.

На самом деле, можно сказать, что практически каждый процесс, запущенный у вас в системе, имеет несколько потоков.

Операционная система может виртуально создать для себя множество потоков и выполнять их все как бы одновременно, даже если физически процессор и одноядерный.

Например, Windows – это многозадачная операционная система, то есть она может одновременно выполнять две и более программ или процессов.

И Windows – это также и многопоточная операционная система.

Это означает, что в действительности программы состоят из ряда более простых потоков выполнения.

Выполнение этих потоков планируется так же, как и выполнение процессов.

Если процессор одноядерный, и так как несколько потока выполняются у нас одновременно, то нужно создать для пользователя, эту самую одновременность выполнения.

Операционная система, делает это хитро, за счет переключения между выполнением этих потоков (эти переключения мгновенны и время идет в миллисекундах).

То есть, система некоторое время выполняет один поток, затем резко переключается на выполнение другого потока, и так далее по кругу.

Таким образом, создается впечатление одновременного выполнения нескольких задач.

Но при этом теряется производительность.

Если процессор многоядерный, тогда переключения может не потребоваться.

Система будет посылать каждый поток на отдельное ядро.

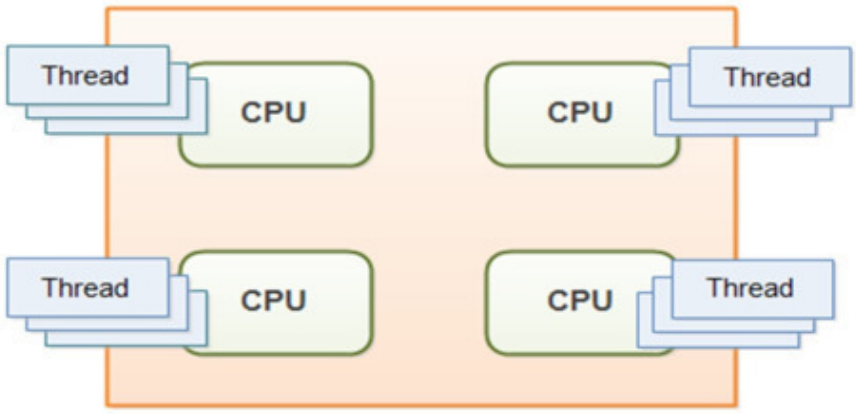
Несколько потоков могут выполняться одновременно, каждый на своем ядре.

Но тут есть проблема.

Для использования преимуществ многоядерности, код программы должен быть оптимизирован для выполнения на многоядерных процессорах.

Это означает, что программа, или процесс, должна быть максимально распараллелена в коде по отдельным задачам.

Если у вас есть многоядерный процессор, и у нас есть два ядра или два процессора P0 и P1, у вас будет возможность создать единицы выполнения, называемые потоками, T1, T2, T3.

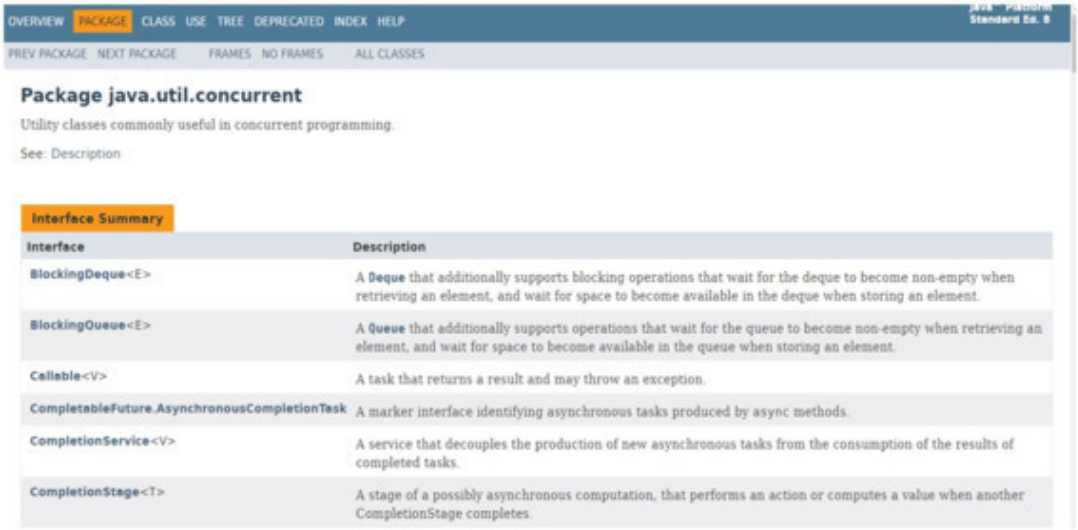


И операционная система сама позаботится о планировании этих потоков на процессорах по мере их доступности.

Таким образом вы получаете многопоточное выполнение.

Платформа Java обеспечивает поддержку многопоточности с помощью пакета java.util.concurrent.

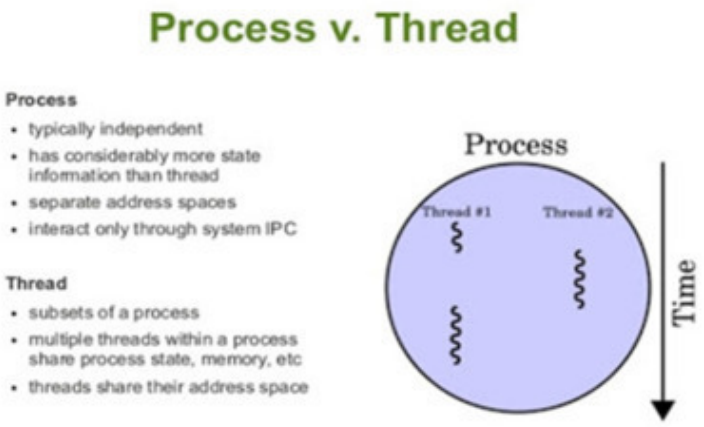
В многопоточном программировании существуют две основные единицы исполнения – это процессы и потоки.



И многопоточное программирование на Java в основном касается потоков.

Чем отличается поток от процесса?

Процесс имеет автономную среду исполнения.



Обычно процесс имеет полный, приватный набор базовых ресурсов среды выполнения, например, каждый процесс имеет собственное выделенное пространство памяти.

Процессы часто ассоциируются с приложением.

Однако то, что пользователь видит, как одно приложение, может быть на самом деле набором взаимодействующих процессов.

Для облегчения взаимодействия между процессами большинство операционных систем поддерживают Inter Process Communication (IPC).

IPC используется не только для связи между процессами в одной и той же системе, но и процессов в разных системах.

Java поддерживает IPC с помощью сокетов, библиотек RMI и CORBA.

Каждый экземпляр работающей виртуальной машины Java представляет собой один процесс.

Приложение Java может создавать дополнительные процессы с помощью объекта ProcessBuilder.

Потоки существуют в процессе – каждый процесс имеет хотя бы один поток.

Потоки используют общие ресурсы процесса, включая память и открытые файлы.

Это обеспечивает эффективное, но потенциально проблематичное взаимодействие между процессами.

Каждый поток имеет свой собственный стек вызовов, но может обращаться к общим данным других потоков в одном и том же процессе.

Каждый поток имеет свой собственный кеш памяти.

Если поток читает общие данные, он сохраняет эти данные в своем собственном кеше памяти.

Несколько потоков создаются в приложении для обеспечения параллельной или скорее независимой обработки или асинхронного поведения.

Многопоточность обещает быстрее выполнить определенную задачу, поскольку эти задачи можно разделить на подзадачи, и эти подзадачи могут выполняться параллельно или независимо.

При этом ускорение программы с помощью многопоточных вычислений на нескольких процессорах ограничено размером последовательной части программы. Это так называемый закон Амдала.

Этот закон гласит следующее – В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента.

Согласно этому закону, ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей, ограничено временем, необходимым для выполнения её последовательных инструкций.

Потоки имеют собственный стек вызовов, но также могут обращаться к общим данным. Поэтому у вас есть две основные проблемы, проблемы с видимостью и доступом.

Проблема видимости возникает, если поток A читает общие данные, которые позже изменяются потоком B, а поток A не знает об этом изменении.

Проблема доступа может возникнуть, если несколько потоков получают доступ и изменяют одновременно одни и те же общие данные.

Проблема видимости и доступа может привести к сбою в работе – программа перестанет реагировать и войдет в ступор или взаимную блокировку из-за одновременного доступа к данным, или может быть сбой безопасности – программа создаст неверные данные.

Как решаются эти проблемы мы обсудим позже.

Таким образом, каждое приложение имеет хотя бы один поток – или несколько, если учитывать «системные» потоки, которые выполняют такие функции, как управление памятью и обработка событий.

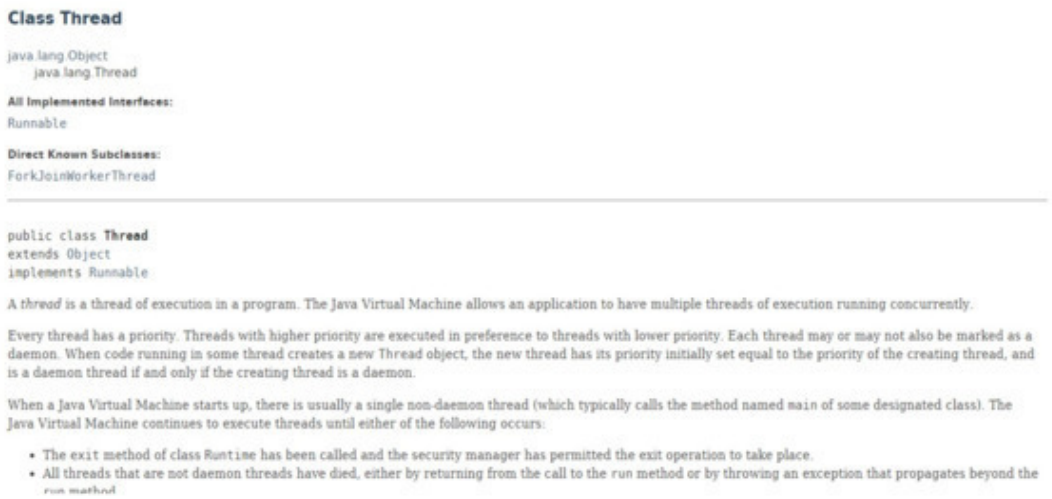
Но с точки зрения программиста, вы начинаете с одного потока, называемого основным потоком.

Этот поток имеет возможность создавать дополнительные потоки.

Вопрос в том, как мы можем создать, запустить и выполнить поток?

В Java каждый поток представлен экземпляром класса Thread.

Создать поток, или экземпляр Thread, можно двумя способами.



Class Thread

java.lang.Object
  java.lang.Thread

All Implemented Interfaces:
Runnable

Direct Known Subclasses:
ForkJoinWorkerThread

---

public class **Thread**
extends Object
implements Runnable

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

Первый способ, это сначала создать объект Runnable.

```
public class HelloRunnable implements
Runnable {

public void run() {
System.out.println("Hello from a thread!");
  }

 public static void main(String args[]) {
 (new Thread(new HelloRunnable())).start();
  }

}
```

```
public class HelloThread extends Thread {

public void run() {
System.out.println("Hello from a thread!");
  }

  public static void main(String args[]) {
    (new HelloThread()).start();
  }

}
```

Интерфейс Runnable определяет один метод run, предназначенный для того, чтобы содержать код, выполняемый в потоке.

После создания, объект Runnable передается конструктору класса Thread.

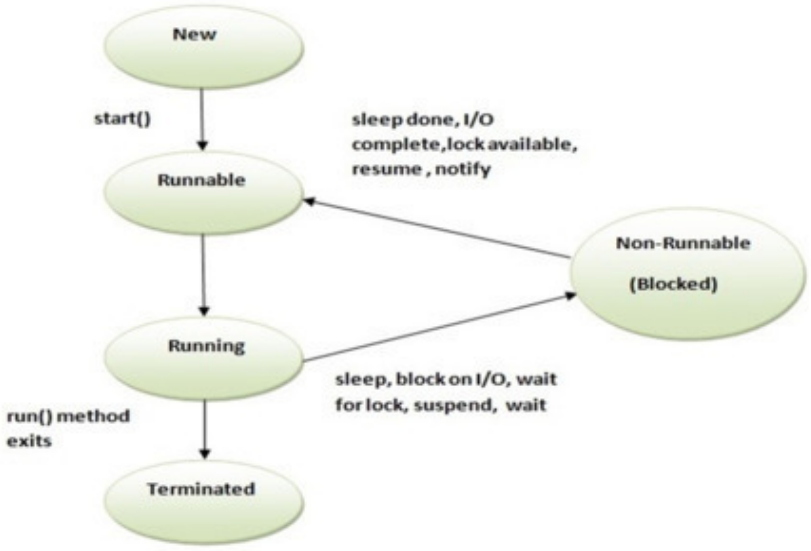
И поток запускается методом start.

Второй способ, это создать подкласс класса Thread.

Сам класс Thread реализует интерфейс Runnable, и при этом его метод run пустой.

Поэтому нужно создать подкласс класса Thread и предоставить собственную реализацию метода run.

Таким образом, первая ключевая операция – это создание потоков.



Но ключевой момент здесь – вам нужно указать вычисление, которое должно быть выполнено в потоке.

Затем после создания потока, он фактически не начинает выполнение.

Поэтому, следующее, что вам нужно сделать, это вызвать метод start.

Теперь, ваша основная программа сама по себе является потоком.

И у нас есть основной поток, который создает и запускает другой поток.

В другом потоке выполняется свой код.

Теперь основной поток после запуска другого потока может выполнить свой код.

В этом случае у нас параллельно выполняются два куска кода на двух разных ядрах.

Класс Thread содержит метод join.

Метод join может быть использован для того, чтобы приостановить выполнение текущего потока до тех пор, пока другой поток не закончит свое выполнение.

join

```
public final void join()
                throws InterruptedException
```

Waits for this thread to die.

An invocation of this method behaves in exactly the same way as the invocation

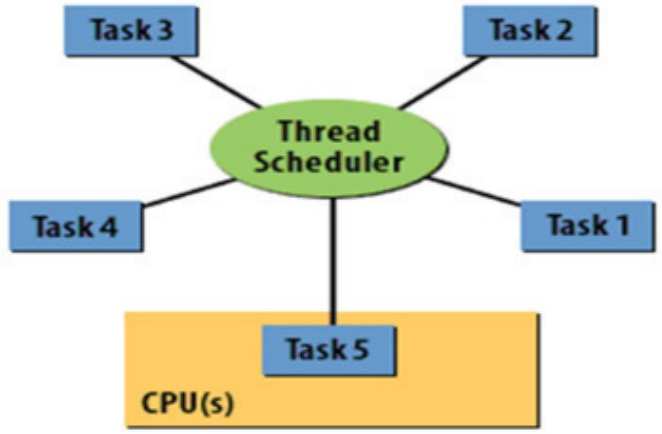
```
join(0)
```

Throws:

`InterruptedException` - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

Как правило, мы используем более одного потока.

В этом случае, планировщик потоков планирует потоки, что не гарантирует порядок выполнения потоков.



В идеальном мире все потоки всех программ работают на отдельных процессорах.

Но в реальности, потоки должны разделяться между одним или несколькими процессорами.

Либо JVM, либо операционная система базовой платформы определяют, как распределять ресурс процессора среди потоков – задача, известная как планирование потоков.

Эта часть JVM или операционной системы, которая выполняет планирование потоков, является планировщиком потоков.

Java не заставляет виртуальную машину планировать потоки определенным образом, поэтому планирование потоков зависит от конкретной платформы.

Предположим, у нас есть два потока t1 и t2.

Несмотря на то, что мы запустили потоки последовательно, планировщик потоков не запускает и не завершает их в указанном порядке.

```
public class JoinExample {
  public static void main(String[] args) {
    Thread t1 = new Thread(new MyClass(), "t1");
    Thread t2 = new Thread(new MyClass(), "t2");

    t1.start();
    t2.start();
}
}
```

```
class MyClass implements Runnable{
  @Override
  public void run() {
          Thread t = Thread.currentThread();
    System.out.println("Thread started: "+t.getName());
    try {
      Thread.sleep(4000);
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
    System.out.println("Thread ended: "+t.getName());
  }
}
```

Каждый раз, когда вы запускаете этот код, вы можете получить разные результаты.

А если поток t1 должен использовать вычисления потока t2, что нам делать?

Решить эту проблему мы можем с помощью метода join ().

Этот код запустит второй поток t2, только после завершения первого потока t1, так как метод join приостанавливает выполнение главного потока до тех пор, пока не завершится поток t1.

```
public class JoinExample {
  public static void main(String[] args) {
    Thread t1 = new Thread(new MyClass(), "t1");
    Thread t2 = new Thread(new MyClass(), "t2");

    t1.start();

    try {
      t1.join();
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }

    t2.start();

}
}
```

```
class MyClass implements Runnable{
  @Override
  public void run() {
          Thread t = Thread.currentThread();
    System.out.println("Thread started: "+t.getName());
    try {
      Thread.sleep(4000);
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
    System.out.println("Thread ended: "+t.getName());
  }
}
```

Если поток прерывается, бросается исключение InterruptedException.

Теперь, предположим, что мы передали в метод run класса MyClass основной поток и применили к нему метод join.

Тогда первый поток будет ждать, когда завершится основной поток, а основной поток будет ждать, когда завершится первый поток.

Возникнет дедлок deadlock или взаимная блокировка потоков.

Для отладки долгоиграющих операций, например, сетевых запросов, часто используется статический метод sleep класса Thread.

Вызов этого метода ставит выполнение текущего потока на паузу, при этом нужно указать количество миллисекунд паузы.

Здесь также нужно обрабатывать исключение InterruptedException.

Это исключение, которое метод бросает, когда другой поток прерывает текущий поток, при работающем методе.

Теперь, вы можете столкнуться с ситуацией, когда вам нужно выполнить некоторые длительные задачи в отдельных потоках.

И возможно, вам нужно будет завершить работу какой-либо задачи еще до того, как задача будет полностью выполнена, с помощью остановки соответствующего потока.

Например, при закрытии приложения, которое может использовать несколько потоков, и они могут быть не завершены в момент закрытия приложения.

Как запросить задачу, выполняемую в отдельном потоке, закончиться раньше?

Как заставить задачу реагировать на такой запрос?

В этом примере создается задача, которая печатает числа от 0 до 9 в консоли.

```
public static void main(String[] args) throws InterruptedException {
   Thread taskThread = new Thread(taskThatFinishesEarlyOnInterruption());
   taskThread.start();
   Thread.sleep(3000);
   taskThread.interrupt();
   taskThread.join(1000);
}

private static Runnable taskThatFinishesEarlyOnInterruption() {
   return () -> {
      for (int i = 0; i < 10; i++) {
         System.out.print(i);
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            break;
         }
      }
   };
}
```

После печати числа, задача должна подождать 1 секунду перед печатью следующего числа.

Задача выполняется в отдельном потоке, отличном от основного потока приложения.

После запуска задачи основной поток должен подождать 3 секунды и затем завершить работу.

При завершении работы приложение должно запросить завершение выполняемой задачи.

Перед тем, как полностью закрыть приложение, приложение должно максимально ждать 1 сек для завершения задачи.

Задача должна ответить на запрос завершения, немедленно останавливаясь.

Общее выполнение задачи занимает не менее 9 секунд.

Поэтому задача не сможет распечатать все десять чисел от 0 до 9.

Для запроса на прерывание потока, основной поток вызывает метод прерывания interrupt.

В Java один поток не может просто остановить другой поток.

Поток может только запросить остановку другого потока.

И запрос выполняется в виде вызова метода interrupt.

Вызов метода interrupt в экземпляре Thread устанавливает флаг прерывания как true.

Если этот поток заблокирован вызовом методов wait, join или sleep, то его статус прерывания будет очищен, и он выбросит исключение InterruptedException.

Таким образом, как только taskThread прерывается основным потоком, Thread.sleep (1000) отвечает на прерывание, выбрасывая исключение.

Исключение InterruptedException обрабатывается, прерывая цикл и тем самым заканчивая задачу раньше.

Таким образом, чтобы задача немедленно реагировала на запрос прерывания, можно использовать обработку исключения InterruptedException.

После очистки статуса прерывания, подтвердить этот статус можно самопрерыванием с помощью вызова Thread.currentThread().interrupt ().

```
Thread.currentThread().interrupt();

Thread.isInterrupted()
```

И без использования обработки InterruptedException, прервать цикл задачи можно, проверяя статус прерывания с помощью вызова Thread.isInterrupted ().

Синхронизация потоков

Теперь, когда мы рассмотрели потоки, давайте разберем ключевую концепцию в многопоточном программировании, которая идет рука об руку с потоками, и это блокировки.

Потоки взаимодействуют между собой, главным образом, путем совместного доступа к полям объектов.

Это взаимодействие делает возможными два вида ошибок: интерференция потоков и ошибки согласованности памяти.

Предположим, что у нас есть очень простой метод объекта, который принимает число и увеличивает его на единицу.

```
class Counter {
  private int c = 0;

  public void increment() {
    c++;
  }

  public void decrement() {
    c--;
  }

  public int value() {
    return c;
  }

}
```

Другой метод этого объекта уменьшает это число на единицу.

Предположим, есть два потока T1 и T2, и один поток хочет увеличить число, а другой поток хочет уменьшить число.

Эти два потока могут быть запланированы на двух разных ядрах, и они могут читать и записывать поле объекта в одно и то же время, и результат будет непредсказуемым.

При одновременной записи возникнет интерференция потоков.

А при одновременной записи и чтении возникнет ошибка согласованности памяти.

Как нам избежать ситуации, когда два потока хотят получить доступ к одному и тому же объекту одновременно?

Для этого используется блокировка.

Java обеспечивает блокировки для защиты определенных частей кода, которые будут выполняться несколькими потоками одновременно.

Самый простой способ блокировки определенного метода – это определить метод с ключевым словом synchronized.

Ключевое слово synchronized в Java обеспечивает:

```
public class SynchronizedCounter {
  private int c = 0;

  public synchronized void increment() {
    c++;
  }

  public synchronized void decrement() {
    c--;
  }

  public synchronized int value() {
    return c;
  }
}
```

Что только один поток может одновременно выполнять блок кода

Что каждый поток, входящий в синхронизированный блок кода, видит результаты всех предыдущих модификаций, которые были защищены одной и той же блокировкой.

Синхронизация необходима для взаимоисключающего доступа к блокам и для надежной связи между потоками.

Синхронизация метода обеспечивает, что, когда один поток выполняет синхронизированный метод объекта, все другие потоки, которые вызывают синхронизированные методы этого объекта приостанавливают выполнение до тех пор, пока первый поток не закончит свою работу с объектом.

Когда синхронизированный метод завершится, он автоматически установит причинно-следственную связь для последующего вызова синхронизированного метода этого объекта.

Это гарантирует, что изменения состояния объекта будут видны для всех потоков.

Когда поток вызывает синхронизированный метод, он автоматически получает внутреннюю блокировку для объекта этого метода и освобождает его при возврате метода.

Освобождение блокировки происходит, даже если возврат метода был вызван неперехваченным исключением.

Другими словами, каждый объект в Java имеет ассоциированный с ним монитор.

Монитор представляет своего рода инструмент для управления доступа к объекту.

Когда выполнение кода доходит до оператора synchronized, монитор объекта захватывается владельцем, и на это время монопольный доступ к синхронизированному коду имеет только один поток, который является владельцем монитора.

После окончания работы блока кода, монитор объекта освобождается и становится доступным для других потоков.

Когда вызывается статический синхронизированный метод, так как статический метод связан с классом, а не с объектом, в этом случае поток получает блокировку для объекта Class, связанного с классом и представляющего класс в среде выполнения.

Таким образом, доступ к синхронизированным статическим полям класса контролируется блокировкой, отличной от блокировки для любого экземпляра класса. Поэтому статические синхронизированные методы и нестатические синхронизированные методы никогда не заблокируют друг друга.

Конструкторы не могут быть синхронизированы – использование ключевого слова synchronized для конструктора является синтаксической ошибкой.

Синхронизация конструктора не имеет смысла, потому что только поток, который создает объект, имеет доступ к нему во время его создания.

Еще раз, если два нестатических метода класса объявлены как synchronized, то в каждый момент времени из разных потоков на одном объекте может быть вызван только один из них.

Поток, который вызывает метод первым, захватит монитор, и второму потоку придется ждать.

Это верно только для разных потоков.

Один и тот же поток может вызвать синхронизированный метод, внутри него – другой синхронизированный метод на том же экземпляре. И это будет повторная блокировка.

Поскольку этот поток владеет монитором, проблем второй вызов не создаст.

Это верно только для вызовов методов одного экземпляра.

У разных экземпляров разные мониторы, поэтому одновременный вызов нестатических методов проблем не создаст.

Другой способ создания синхронизированного кода – синхронизированные блоки.

В отличие от синхронизированных методов, синхронизированные блоки должны указывать объект, который обеспечивает внутреннюю блокировку.

```
synchronized(object) {

. . . code

}
```

Когда один поток заходит внутрь блока кода, помеченного словом synchronized, то Java-машина тут же блокирует монитор объекта, который указан в круглых скобках после слова synchronized.

Больше ни один поток не сможет зайти в этот блок, пока наш поток его не покинет.

Как только наш поток выйдет из блока, помеченного synchronized, то монитор тут же автоматически освобождается и будет свободен для захвата другим потоком.

Для нестатических методов, синхронизация метода эквивалентна синхронизации тела метода с объектом this.

```
public synchronized void someMethod(){
   // code
}
```

```
public void someMethod(){
   synchronized(this){
      // code
   }
}
```

```
public class SomeClass{

   public static synchronized void someMethod(){
      //code
   }

}
```

```
public class SomeClass{

   public static void someMethod(){
      synchronized(SomeClass.class){
         //code
      }
   }

}
```

Для статических методов, синхронизация метода эквивалентна синхронизации тела метода с объектом Class.

Предположим, что класс имеет два поля экземпляра: c1 и c2, которые никогда не используются вместе.

```
public class MsLunch {
   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc1() {
      synchronized(lock1) {
         c1++;
      }
   }

   public void inc2() {
      synchronized(lock2) {
         c2++;
      }
   }
}
```

Все обновления этих полей должны быть синхронизированы, но нет никаких причин препятствовать тому, чтобы обновление c1 чередовалось с обновлением c2, чтобы не создавать ненужную блокировку.

Вместо использования синхронизированных методов или использования блокировки this, мы создаем два объекта исключительно для обеспечения блокировок.

Таким образом синхронизация блоков может дать возможность из разных потоков на одном объекте вызывать разные синхронизированные блоки.

Атомарный доступ и volatile

В программировании атомарное действие – это действие, которое происходит за один раз.

Атомарное действие не может остановиться посередине: оно либо происходит полностью, либо вообще не происходит.

Никакие промежуточные результаты атомарного действия не видны, пока действие не будет завершено.

Например, оператор инкремента ++, не является атомарным действием.

Он состоит из следующих действий:

– Получить текущее значение.

– Увеличить полученное значение на 1.

– Сохранить увеличенное значение.

Поэтому очень простые выражения могут определять сложные действия, которые могут разлагаться на другие действия.

Но есть действия, которые являются атомарными:

Это чтение и запись всех переменных, ссылочных на объекты и примитивных переменных, за исключением переменных типа long и double.

Так как в Java 64-битные long и double значения рассматриваются как два 32-битных значения.

Это означает, что 64-разрядная операция записи выполняется как две отдельные 32-разрядные операции.

И это значит, что действия с long и double переменными не являются потокобезопасными.

Когда несколько потоков получают доступ к long или double значению без синхронизации, это может вызвать проблемы.

Чтобы обеспечить атомарность действий с long и double значениями можно использовать ключевое слово volatile.

Если переменная объявлена как volatile, это означает, что она может изменяться разными потоками.

```
volatile int i = 0;

public synchronized void incrementI(){
  i++;
}
```

Среда выполнения JRE неявно обеспечивает синхронизацию при доступе к volatile-переменным, но с очень большой оговоркой: чтение volatile-переменной и запись в volatile-переменную синхронизированы, а неатомарные операции, такие как операция инкремента или декремента — нет.

Атомарные действия не могут перемешиваться, поэтому их можно использовать, не опасаясь интерференции потоков.

Однако это не устраняет необходимости синхронизации атомарных действий, так как возможны ошибки согласованности памяти.

В целях повышения производительности среда выполнения JRE сохраняет локальные копии переменных для каждого потока, который на них ссылается.

Такие «локальные» копии переменных работают как кэш и помогают потоку избежать обращения к главной памяти каждый раз, когда требуется получить значение переменной.

Поэтому если один поток изменит значение переменной, то другой поток может не увидеть это изменение, так как будет считывать значение из своего кэша. Это и будет ошибкой согласованности памяти

Однако если переменная помечена как volatile, то, когда бы поток не считывал значение, он будет считывать ее текущее значение.

Использование volatile переменных, не только для long и double, снижает риск ошибок согласованности памяти, поскольку любая запись в volatile переменную устанавливает связь между событиями и последующими чтениями этой же переменной.

Это означает, что изменения в volatile переменной всегда видны для других потоков.

Опять же речь идет только об операциях чтения и записи.

Резюмируя, объявление блока кода синхронным обеспечивает для кода атомарность и видимость.

Атомарность значит, что только один поток одновременно может выполнять код, защищенный данным объектом-монитором (блокировкой), позволяя предотвратить многочисленные потоки от столкновений друг с другом во время обновления общего состояния.

Видимость связана с особенностями кэширования памяти и оптимизацией программы в процессе компилирования.

Обычно потоки могут свободно кэшировать значения для переменных так, чтобы они не обязательно сразу же были бы видны другим потокам, но, если разработчик использовал синхронизацию, во время выполнения будет проверяться, что обновления переменных, выполненные одним потоком до выхода из синхронизированного блока, сразу же будут видны другому потоку, когда он будет входить в синхронизированный блок, защищенный тем же монитором (блокировкой).

Подобное правило видимости существует и для переменных volatile.

Живучесть Liveness

Теперь, давайте рассмотрим фундаментальное свойство корректности многопоточных программ, которое называется LIVENESS или живучесть.

Когда вы пишете не многопоточную, а последовательную программу, и в ней есть ошибка, вы запускаете ее, и на экране ничего не появляется.

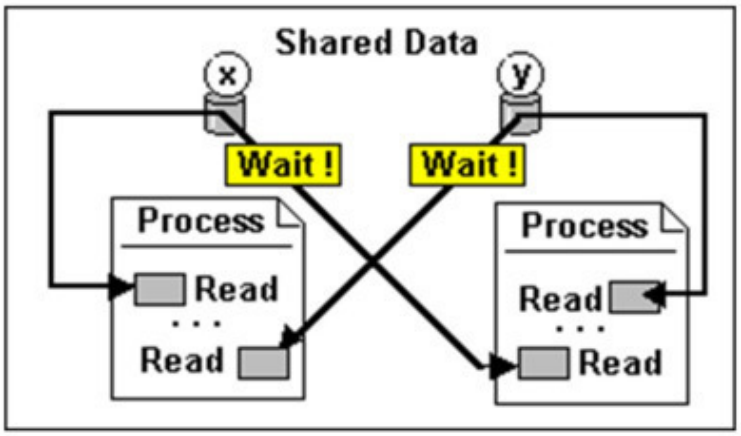
Вы ждете, а затем выясняете, например, что у вас в коде есть бесконечный цикл.

И тогда вы исправляете ошибку.

К сожалению, в многопоточных программах есть много других способов получить этот эффект пустого экрана.

Один из них – это DEADLOCK или взаимная блокировка.

Это мы уже видели в случае операции join.



Если два потока соединяются друг с другом, они создают цикл взаимоблокировки, и по этой причине программа не завершается.

Это не бесконечный цикл в обычном понимании, это два потока, заблокированных друг от друга на неопределенный срок.

Существуют и другие способы получения взаимоблокировки.

Например, если поток T1 выполняет синхронизированную операцию на объекте A и вложенную синхронизированную операцию на объекте B, а поток T2 выполняет синхронизированную операцию на объекте B и вложенную синхронизированную операцию на объекте A, мы получаем другую форму взаимоблокировки.

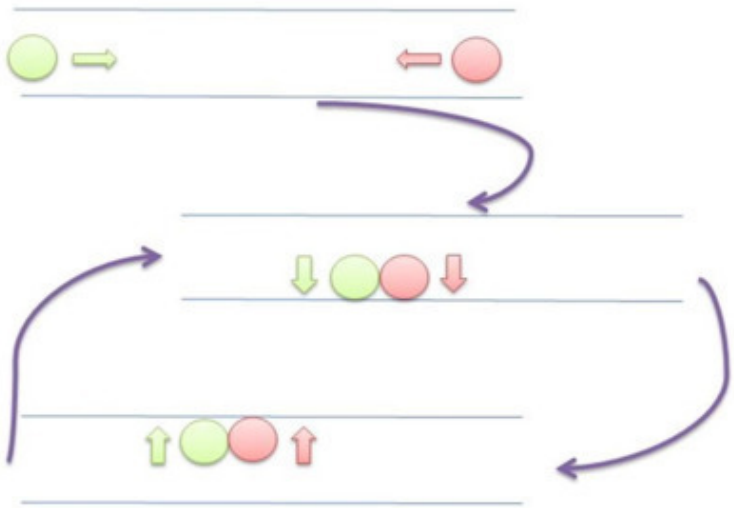
```
public static Object cacheLock = new Object();
 public static Object tableLock = new Object();
 ...
 public void oneMethod(){
  synchronized (cacheLock) {
    synchronized (tableLock) {
     doSomething();
    }
  }
 }
 public void anotherMethod(){
  synchronized (tableLock) {
    synchronized (cacheLock) {
     doSomethingElse();
    }
  }
 }
```

Поток T1 может получить монитор объекта A одновременно с тем, что поток T2 получит монитор объекта B, а затем каждый поток будет ожидать монитора B и A соответственно неопределенный срок.

Одним из лучших способов предотвращения взаимоблокировки – это избегать одновременного получения более одного монитора.

Еще одно нарушение живучести, это LIVELOCK или динамическая взаимоблокировка.

В livelock потоки не блокируются, но они находятся в режиме, в котором их выполнение не продвигается дальше, это похоже на пат в шахматной игре.



Например, если у нас есть объект, скажем, изменяемая целочисленная переменная x, и у нас есть два потока.

Поток T1 в цикле увеличивает x, затем читает значение x и продолжает делать это, пока x меньше 2.

А поток T2 в цикле уменьшает значение x, затем читает значение x и продолжает делать это, пока x больше -2.

Возможна ситуация, при которой поток T1 получает x = 1, но прежде чем он получит шанс увеличить x и достичь x = 2, поток T2 уменьшает x, противодействуя тому, что делает T1.

И делает x = -1.

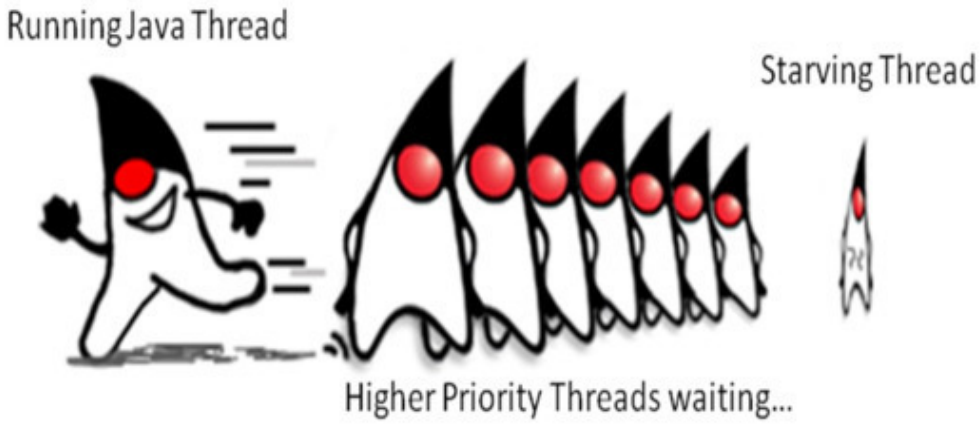
Но до того, как поток T2 получит шанс уменьшить x до -2, поток T1 может снова увеличить x до 1.

И так до бесконечности.

Таким образом, значение x может двигаться вперед и назад, как непрерывный бесконечный пинг-понг.

Теперь третий вид проблемы с живучестью, называется STARVATION или голодание.

Starvation возникает, когда какой-либо поток не может получить доступ к общим ресурсам и не может быть выполнен в результате, например, синхронизированного доступа к этому ресурсу другими потоками, выполнение которых занимает долгое время.



В результате этот поток голодает.
Паттерн защищенный блок Guarded Block

Многопоточное программирование в Java

Паттерн Guarded Block

Предположим у нас есть задача написать приложение Producer-Consumer.

Это приложение состоит из двух потоков – производителя, который создает данные, и потребителя, который что-то делает с этими данными.

Конец ознакомительного фрагмента.