Программирование на Java

Лекция 12. Потоки выполнения. Синхронизация

20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

Эта лекция завершает рассмотрение ключевых особенностей Java. Последняя тема раскрывает особенности создания многопоточных приложений – такая возможности присутствует в языке начиная с самых первых версий.

Первый вопрос – как на много- и, самое интересное, однопроцессорных машинах выполняются несколько потоков одновремнно, и для чего они нужны в програме. Затем описываются классы, необходимые для создания, запуска и управления потоками в Java. При одновремнной работе с данными из нескольких мест возникает проблема синхронного доступа, блокировок и, как следствие, взаимных блокировок. Изучаются все механизмы, присутствующие в языке, для корректной организации такой логики работы.

Оглавление

Лекция 12. Потоки выполнения. Синхронизация

Содержание лекции.

1. Введение

До сих пор во всех рассматриваемых примерах подразумевалось, что в один момент времени исполняется лишь одно выражение, или действие. Однако начиная с самых первых версий, виртуальные машины Java поддерживают многопоточность, т.е. поддержку нескольких потоков исполнения (threads) одновременно.

В данной главе сначала рассматриваются преимущества такого подхода, способы реализации и возможные недостатки.

Затем описываются базовые классы Java, которые позволяют запускать и управлять потоками исполнения.

При одновременном обращении нескольких потоков к одним и тем же данным может возникнуть ситуация, когда результат программы будет зависеть от случайных факторов, таких как временное чередование исполнения операций несколькими потоками. В такой ситуации становятся необходимым механизмы синхронизации, обеспечивающие последовательный, или монопольный, доступ. В Java этой цели служит ключевое слово

synchronized. Предварительно будет рассмотрен подход к организации хранения данных в виртуальной машине.

В заключение рассматриваются методы wait(), notify(), notifyAll() класса Object.

2. Многопоточная архитектура

Не претендуя на полноту изложения, рассмотрим общее устройство многопоточной архитектуры, ее достоинства и недостатки.

Реализацию многопоточной архитектуры проще всего представить себе для системы, в которой есть несколько центральных вычислительных процессоров. В этом случае для каждого из них можно выделить задачу, которую он будет выполнять. В результате несколько задач будут обслуживаться одновременно.

Однако возникает вопрос- каким же тогда образом обеспечивается многопоточность в системах с одним центральным процессором, который в принципе выполняет лишь одно вычисление в один момент времени? В таких системах применяется процедура квантования времени (time-slicing). Время разделяется на набольшие интервалы. Перед началом каждого интервала принимается решение, какой именно поток выполнения будет отрабатываться на протяжении этого кванта времени. За счет такого частого переключения между задачами эмулируется многопоточная архитектура.

На самом деле, как правило и для многопроцессорных систем применяется процедура квантования времени. Дело в том, что даже в мощных серверах приложений процессоров не так много (редко бывает больше десяти), а потоков исполнения запускается как правило гораздо больше. Например, операционная система Windows без единого стартованного приложения инициализирует десятки, а то и сотни потоков. Квантование времени позволяет упростить управление выполнением задач на всех процессорах.

Теперь перейдем к вопросу о преимуществах - зачем вообще может потребоваться более одного потока выполнения?

Среди начинающих программистов существует мнение, что многопоточные программы работают быстрее. Рассмотрев способ реализации многопоточности, можно утверждать, что такие программы работают на самом деле медленнее. Действительно, для переключения между задачами на каждом интервале требуется дополнительное время, а ведь они (переключения) происходят довольно часто. Если бы процессор, не отвлекаясь, выполнял задачи последовательно, одну за другой, он закончил бы их заметно быстрее. Стало быть, преимущества заключаются не в этом.

Первый тип приложений, который выигрывает от поддержки многопоточности, предназначен для задач, где действительно требуется выполнять несколько действий одновременно. Например, будет вполне обоснованно ожидать, что сервер общего пользования будет обслуживать несколько клиентов одновременно. Можно легко представить себе пример из сферы обслуживания, когда есть несколько потоков клиентов, и желательно обслуживать их все одновременно.

Другой пример - активные игры, или подобные приложения. Необходимо одновременно опрашивать клавиатуру и другие устройства ввода, чтобы реагировать на действия пользователя. В то же время одновременно необходимо рассчитывать и перерисовывать изменяющееся состояние игрового поля.

Понятно, что в случае отсутствия поддержки многопоточности для реализации подобных приложений потребовалось бы реализовывать квантования времени вручную. Условно говоря - одну секунду проверять состояние клавиатуры, а следующую - пересчитывать и перерисовывать игровое поле. Если сравнить две реализации time-slicing, одну - на низком уровне, выполненную средствами, как правило, операционной системы, другую - выполняемую вручную, на языке высокого уровня, мало подходящего для таких задач, то становится понятным первое и, возможно, главное преимущество многопоточности. Она обеспечивает наиболее эффективную реализацию процедуры квантования времени, существенно облегчая и укорачивая процесс разработки приложения. Код переключения между задачами на Java выглядел бы гораздо более громоздко, чем независимое описание действий для каждого потока независимо.

Следующее преимущество проистекает из того, что компьютер состоит не только из одного или нескольких процессоров. Вычислительное устройство - лишь одно из ресурсов, необходимых для выполнения задач. Всегда есть оперативная память, дисковая подсистема, сетевые подключения, периферия (принтер и т.п.) и другие. Предположим, пользователю требуется распечатать большой документ и скачать большой файл из сети. Очевидно, что обе задачи требуют совсем небольшого участия процессора, а основные необходимые ресурсы, которые будут использоваться на пределе возможностей, у них разные - сетевое подключение и принтер. Значит, если выполнять задачи одновременно, но замедление от организации квантования времени будет незначительным, процессор легко справится с обслуживанием обеих задач. В то же время, если каждая задача по отдельности занимала, скажем, 2 часа, то вполне вероятно, что и при одновременном исполнении потребуется не более тех же 2 часов, а сделано при этом будет гораздо больше.

Если же задачи в основном загружают процессор (например, математические расчеты), то их одновременное исполнение займет в лучшем случае столько же времени, что и последовательное, а скорее всего и больше.

Третье преимущество появляется из-за возможности более гибко управлять выполнением задач. Предположим, пользователь системы, не поддерживающей многопоточность, решил скачать большой файл из сети или произвести сложное вычисление, что занимает, скажем, 2 часа. Запустив задачу на выполнение, он может внезапно обнаружить, что ему нужен не этот, а какой-нибудь другой файл (или вычисление с другими начальными параметрами). Однако если приложение занимается только работой с сетью (вычислениями) и не реагирует на действия пользователя (не обрабатываются данные с устройств ввода, таких как клавиатура или мышь), то он не сможет быстро исправить свою ошибку. Получается, что процессор выполняет большее количество вычислений, но при этом приносит гораздо меньше пользы своему пользователю.

Процедура квантования времени поддерживает приоритеты (priority) задач. В Java приоритет представляется целым числом. Чем больше число, тем выше приоритет. Строгих правил работы с приоритетами нет, каждая реализация может вести себя по-разному на разных платформах. Однако есть общее правило - поток с более высоким приоритетом будет получать большее количество квантов времени на исполнение, и таким образом сможет быстрее выполнять свои действия и реагировать на поступающие данные.

В описанном примере представляется разумным запустить дополнительный поток, отвечающий за взаимодействие с пользователем. Ему можно поставить высокий приоритет, так как в случае бездействия пользователя этот поток практически не будет занимать

ресурсы машины. В случае же активности пользователя необходимо как можно быстрее произвести необходимые действия, чтобы обеспечить максимальную эффективность работы пользователя.

Рассмотрим здесь же еще одно свойство потоков. Раньше, когда рассматривались однопоточные приложения, завершение вычислений однозначно приводило к завершению выполнения программы. Теперь же приложение должно работать видимо до тех пор, пока есть хоть один действующий поток исполнения. В то же время часто бывают нужны обслуживающие потоки, которые не имеют никакого смысла, если они остаются в системе одни. Например, автоматический сборщик мусора в Java запускается в виде фонового (низкоприоритетного) процесса. Его задача - отслеживать объекты, которые уже не используются другими потоками, и затем уничтожать их, освобождая оперативную память. Понятно, что работа одного потока garbage collector'a не имеет никакого смысла.

Такие обслуживающие потоки называют демонами (daemon), это свойство можно установить любому потоку. В итоге приложение выполняется до тех пор, пока есть хотя бы один поток не-демон.

Рассмотрим, как потоки реализованы в Java.

3. Базовые классы для работы с потоками

3.1. Класс Thread

Поток выполнения в Java представляется экземпляром класса Thread. Для того, чтобы написать свой поток исполнения необходимо наследоваться от этого класса и переопределить метод run(). Например,

```
public class MyThread extends Thread {
  public void run() {
   // некоторое долгое действие, вычисление
   long sum=0;
   for (int i=0; i<1000; i++) {
    sum+=i;
   }
   System.out.println(sum);
  }
}
```

Метод run() содержит действия, которые должны исполнятся в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника, и вызвать унаследованный метод start(), который сообщает виртуальной машине, что необходимо запустить новый поток исполнения и начать в нем исполнять метод run().

```
MyThread t = new MyThread();
t.start();
```

В результате чего на консоли появится результат:

```
499500
```

Когда метод run() завершен (в частности, встретилось выражение return) поток выполнения останавливается. Однако, ничто не препятствует записи бесконечного цикла в этом методе. В результате поток не прервет своего исполнения, и будет остановлен только при завершении работы всего приложения.

3.2. Интерфейс Runnable

Описанный подход обладает одним недостатком. Поскольку в Java отсутствует множественное наследование, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, то станет понято, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод - уже знакомый void run(). Запишем пример, приведенный выше, с помощью этого интерфейса:

```
public class MyRunnable implements Runnable {
  public void run() {
   // некоторое долгое действие, вычисление
   long sum=0;
   for (int i=0; i<1000; i++) {
    sum+=i;
   }
   System.out.println(sum);
  }
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), содержащим полезную функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, можно свободно решать в каждом конкретном случае.

Подчеркнем, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

3.3. Работа с приоритетами

Рассмотрим, как в Java потокам можно назначать приоритеты. Для этого в классе Thread существуют методы getPriority() и setPriority(), а также объявлены три константы:

```
MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY
```

Из названия очевидно, что их значения описывают минимальное, максимально и нормальное (по умолчанию) значения приоритета.

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {

 public void run() {
  double calc;
  for (int i=0; i<50000; i++) {
   calc=Math.sin(i*i);
   if (i%10000==0) {
    System.out.println(getName()+" counts " + i/10000);
   }
  }
 }

public String getName() {
  return Thread.currentThread().getName();
}

public static void main(String s[]) {
 // Подготовка потоков
  Thread t[] = new Thread[3];
  for (int i=0; i<t.length; i++) {
   t[i]=new Thread(new ThreadTest(), "Thread "+i);
  }

// Запуск потоков
  for (int i=0; i<t.length; i++) {
   t[i].start();
   System.out.println(t[i].getName()+" started");
  }
 }
}
```

В примере используется несколько новых методов класса Thread:

- getName()

  Обратите внимание, что конструктору класса Thread передается два параметра. К реализации Runnable добавляется строка. Это имя потока, которое используется только для упрощения его идентификации. Имена нескольких потоков могут совпадать. Если его не задать, то Java генерирует простую строку вида "Thread-" и номер потока (вычисляется простым счетчиком) Именно это имя возвращается методом getName(). Его можно сменить с помощью метода setName().

- currentThread()

  Этот статический метод позволяет в любом месте кода получить ссылку на объект класса Thread, представляющий текущий поток исполнения.

Результат работы такой программы будет иметь следующий вид:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 0 counts 0
Thread 1 counts 0
Thread 2 counts 0
Thread 0 counts 1
Thread 1 counts 1
Thread 2 counts 1
Thread 0 counts 2
Thread 2 counts 2
Thread 1 counts 2
Thread 2 counts 3
Thread 0 counts 3
Thread 1 counts 3
Thread 2 counts 4
Thread 0 counts 4
Thread 1 counts 4
```

Можно видеть, что все три потока были запущены один за другим, и начали проводить вычисления. Видно также, что потоки исполняются без определенного порядка, случайным образом. Тем не менее, в среднем они движутся с одной скоростью, никто не отстает и не догоняет.

Введем в программу работу с приоритетам, расставим разные значения для разных потоков и посмотрим, как это скажется на выполнении. Изменяется только метод main()

```
public static void main(String s[]) {
// Подготовка потоков
 Thread t[] = new Thread[3];
 for (int i=0; i<t.length; i++) {
  t[i]=new Thread(new ThreadTest(), "Thread "+i);
  t[i].setPriority(Thread.MIN_PRIORITY +
(Thread.MAX_PRIORITY-Thread.MIN_PRIORITY)/t.length*i);
 }

// Запуск потоков
 for (int i=0; i<t.length; i++) {
  t[i].start();
  System.out.println(t[i].getName()+" started");
 }
}
```

Формула вычисления приоритетов позволяет равномерно распределить все допустимые значения для всех запускаемые потоки. На самом деле, константа минимального приоритета имеет значение 1, максимального - 10, нормального - 5. Так что в простых программах можно явно пользоваться этими величинами и указывать в качестве, например, пониженного приоритета значение 3.

Результатом работы будет:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 2 counts 0
Thread 2 counts 1
Thread 2 counts 2
Thread 2 counts 3
Thread 2 counts 4
Thread 0 counts 0
Thread 1 counts 0
Thread 1 counts 1
Thread 1 counts 2
Thread 1 counts 3
Thread 1 counts 4
Thread 0 counts 1
Thread 0 counts 2
Thread 0 counts 3
Thread 0 counts 4
```

Потоки, как и раньше, стартуют последовательно. Но затем сразу видно, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее весьма показательно, что поток с минимальным приоритетом (Thread 0) тем не менее получил возможность выполнить одно действие раньше, чем отработал поток с более высоким приоритетом (Thread 1). Это говорит о том, что приоритеты не делают систему однопоточной, выполняющей единовременно лишь один поток с наивысшим приоритетом. Напротив, приоритеты позволяют одновременно работать над несколькими задачами с учетом их важности.

Если увеличить параметры метода (выполнять 500.000 вычислений, а не 50.000, и выводить сообщение каждое 1000 вычисление, а не 10.000), то можно будет наглядно увидеть, что все три потока имеют возможность выполнять свои действия одновременно, просто более высокий приоритет позволяет выполнять их чаще.

3.4. Демон-потоки

Демон-потоки позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них. Для работы с этим свойством существуют методы setDaemon() и isDaemon().

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {
```

```
 // Отдельная группа, в которой будут
 // находится все потоки ThreadTest
public final static ThreadGroup GROUP =
 new ThreadGroup("Daemon demo");

 // Стартовое значение,
 // указывается при создании объекта
private int start;

public ThreadTest(int s) {
   start = (s%2==0)? s: s+1;
  new Thread(GROUP, this, "Thread "+start).start();
 }

 public void run() {
  // Начинаем обратный отсчет
  for (int i=start; i>0; i--) {
   try {
    Thread.sleep(300);
   } catch (InterruptedException e) {}

   // По достижению середины порождаем новый
   // поток с половинным начальным значением
   if (start>2 && i==start/2) {
    new ThreadTest(i);
   }
  }
 }

public static void main(String s[]) {
   new ThreadTest(16);
   new DaemonDemo();
 }
}

public class DaemonDemo extends Thread {
 public DaemonDemo() {
  super("Daemon demo thread");
  setDaemon(true);
  start();
 }

 public void run() {
  Thread threads[]=new Thread[10];
  while (true) {
   // Получаем набор всех потоков из
   // тестовой группы
   int count=ThreadTest.GROUP.activeCount();
   if (threads.length<count) threads =
```

```
new Thread[count+10];
   count=ThreadTest.GROUP.enumerate(threads);

   // Распечатываем имя каждого потока
   for (int i=0; i<count; i++) {
    System.out.print(threads[i].getName()+", ");
   }
   System.out.println();
   try {
    Thread.sleep(300);
   } catch (InterruptedException e) {}
  }
 }
}
```

В этом примере происходит следующее. Потоки ThreadTest имеют некоторое стартовое значение, которое передается им при создании. В методе run() это значение последовательно уменьшается. При достижении половины от начальной величины порождается новый поток с вдвое меньшим начальным значением. По исчерпании счетчика поток останавливается. Метод main() порождает первый поток со стартовым значением 16. В ходе программы, значит, будут дополнительно порождены потоки со значениями 8, 4, 2.

За этим процессом наблюдает демон-поток DaemonDemo. Этот поток регулярно получает список всех существующих потоков ThreadTest и распечатывает их имена для удобства наблюдения.

Результатом программы будет:

```
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8, Thread 4,
Thread 16, Thread 8, Thread 4,
Thread 8, Thread 4,
Thread 4, Thread 2,
Thread 2,
```

Несмотря на то, что демон-поток никогда не выходит из метода run(), виртуальная машина прекращает работу как только все не-демон-потоки завершаются.

В примере использовалось несколько дополнительных классов и методов, которые еще не были рассмотрены:

- класс ThreadGroup

  Все потоки находятся в группах, представляемых экземплярами класса ThreadGroup. Группа указывается при создании потока. Если группа не была указана, то поток помещается в ту же группу, где находится поток, породивший его.

  Методы activeCount() и enumerate() возвращают количество и полный список соответственно всех потоков в группе.

- sleep()

  Этот статический метод класса Thread приостанавливает выполнение текущего потока на указанное количество миллисекунд. Обратите внимание, что метод требует обработки исключения InterruptedException. Он связан с возможностью вернуть к работе метод, который приостановил свою работу. Например, если поток находится в выполнении метода sleep(), то есть ничего не исполняет на протяжении указанного периода времени, то его можно вывести из этого состояния, вызвав метод interrupt() из другого потока выполнения. В результате, метод sleep() прервется исключением InterruptedException.

Кроме метода sleep() существует еще один статический метод yield() без параметров. Когда поток вызывает его, он временно приостанавливает свою работу и позволяет отработать другим потокам. Один из методов обязательно должен применяться внутри бесконечных циклов ожидания, иначе есть риск, что такой ничего не делающий поток серьезно затормозит работу остальных потоков.

4. Синхронизация

При многопоточной архитектуре приложения возможны ситуации, когда несколько потоков будут одновременно работать с одними и теми же данными, используя их значения и присваивая новые. В таком случае результат работы программы становится невозможным предопределить, глядя только на исходный код. Финальные значения переменных будут зависеть от случайных факторов, исходя из того, какой поток какое действие успел сделать первым или последним.

Рассмотрим пример:

```
public class ThreadTest  {

 private int a=1, b=2;
 public void one() {
  a=b;
 }
 public void two() {
  b=a;
 }

 public static void main(String s[]) {
  int a11=0, a22=0, a12=0;
```

```
  for (int i=0; i<1000; i++) {
   final ThreadTest o = new ThreadTest();

   // Запускаем первый поток, который
   // вызывает один метод
new Thread() {
    public void run() {
     o.one();
    }
   }.start();

   // Запускаем второй поток, который
   // вызывает второй метод
   new Thread() {
    public void run() {
     o.two();
    }
   }.start();

   // даем время отработать потокам
try {
    Thread.sleep(100);
   } catch (InterruptedException e) {}

   // анализируем финальные значения
if (o.a==1 && o.b==1) a11++;
   if (o.a==2 && o.b==2) a22++;
   if (o.a!=o.b) a12++;
  }
  System.out.println(a11+" "+a22+" "+a12);
 }
}
```

В этом примере два потока исполнения одновременно обращаются к одному и тому же объекту, вызывая у него два разных метода one() и two(). Эти методы пытаются приравнять два поля класса a и b друг другу, но в разном порядке. Учитывая, что исходные значения полей равны 1 и 2 соответственно, можно было бы ожидать, что после того, как потоки завершат свою работу, поля будут иметь одинаковое значение. Однако понять, какое из двух возможных значений они примут, уже невозможно. Посмотрим на результат программы:

```
135 864 1
```

Первое число показывает, сколько раз из тысячи обе переменные приняли значение 1. Второе число соответствует значению 2. Такое сильное преобладание одного из значений обусловлено последовательностью запусков потоков. Если ее изменить, то и количества случаев с 1 и 2 также меняются местами. Третье же число сообщает, что на тысячу случаев произошел один, когда поля вообще обменялись значениями!

При количестве итераций, равном 10.000, были получены следующие данные, которые подтверждают сделанные выводы:

```
494 9498 8
```

А если убрать задержку перед анализом результатов, то получаемые данные радикально меняются:

```
0 3 997
```

Видимо, потоки просто не успевают отработать.

Итак, наглядно показано, сколь сильно и непредсказуемо может меняться результат работы одной и той же программы, применяющей многопоточную архитектуру. Необходимо учитывать, что в приведенном простом примере задержки создавались вручную методом Thread.sleep(). В реальных сложных системах задержки могут возникать в местах проведения сложных операций, их длины непредсказуемы, и оценить их последствия невозможно.

Для более глубокого понимания принципов многопоточной работы в Java рассмотрим организацию памяти в виртуальной машине для нескольких потоков.

4.1. Хранение переменных в памяти

Виртуальная машина поддерживает основное хранилище данных (main storage), в котором сохраняются значения всех переменных и которое используется всеми потоками. Под переменными здесь понимаются поля объектов и классов, а также элементы массивов. Что касается локальных переменных и параметров методов, их значения не могут быть доступы другим потокам, поэтому они не представляют интереса.

Для каждого потока создается его собственная рабочая память (working memory), в которую копируются значения всех переменных перед использованием.

Рассмотрим основные операции, доступные для потоков при работе с памятью:

- use - чтение значения переменной из рабочей памяти потока
- assign - запись значения переменной в рабочую память потока
- read - получение значения переменной из основного хранилища
- load - сохранение значения переменной, прочитанного из основного хранилища, в рабочей памяти
- store - передача значения переменной из рабочей памяти в основное хранилище для будущего сохранения
- write - сохраняет в основном хранилище значение переменной, переданной командой store

Подчеркнем, что перечисленные команды не являются методами каких-либо классов, они не доступны программисту. Сама виртуальная машина использует их для обеспечения корректной работы потоков исполнения.

Поток, работая с переменной, регулярно применяет команды use и assign для использования ее существующего значения и присвоения нового. Кроме этого, должны осуществляться действия по передачи значений из/в основное хранилище. Они выполняются в два этапа.

При получении данных сначала основное хранилище считывает значение командой read, а затем поток сохраняет результат в своей рабочей памяти командой load. Эта пара команд всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую. При отправлении данных сначала поток считывает значение из рабочей памяти командой store, а затем основное хранилище сохраняет его командой write. Эта пара команд также всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую.

Набор этих правил составлялся с тем, чтобы операции с памятью были достаточно строги для точного анализа их результатов, а с другой стороны правила должны оставлять достаточное пространство для различных технологий оптимизаций (регистры, очереди, кэш и т.д.).

Последовательность команд подчиняется следующим правилам:

- все действия, выполняемые одним потоком строго упорядочены, т.е. выполняются одно за другим
- все действия, выполняемые с одной переменной в основном хранилище памяти, строго упорядочены, т.е. следуют одно за другим

За исключением некоторых дополнительных очевидных правил больше никаких ограничений нет. Например, если поток изменил значение сначала одной, а затем другой переменной, то они эти изменения могут быть переданы в основное хранилище в переставленном порядке.

Поток создается с чистой рабочей памятью и должен загрузить все необходимые переменные из основного хранилища перед использованием. Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее использовать.

Таким образом, потоки никогда не взаимодействуют друг с другом на прямую, только через главное хранилище.

4.2. Модификатор volatile

При объявлении полей объектов и классов может быть указан модификатор volatile. Он устанавливает более строгие правила работы со значениями переменных.

Если поток собирается выполнить команду use для volatile переменной, то требуется, чтобы предыдущим действием над этой переменной было обязательно load, и наоборот - операция load может выполняться только перед use. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Аналогично, если поток собирается выполнить команду store для volatile переменной, то требуется, чтобы предыдущим действием над этой переменной было обязательно assign, и наоборот - операция assign может выполняться только если следующей будет store. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Наконец, если проводятся операции над несколькими volatile переменными, то передача соответствующих изменений в основное хранилище должно проводится строго в том же порядке.

При работе с обычными переменными компилятор имеет больше пространства для маневра. Например, при благоприятных обстоятельствах может оказаться возможным предсказать значение переменной, заранее вычислить и сохранить его, а затем в нужный момент использовать уже готовым.

Нужно обратить внимание на два 64-разрядных типа double и long. Поскольку многие платформы поддерживают лишь 32-битную память, величины этих типов рассматриваются как две переменные, и все описанные действия делаются независимо для двух половинок таких значений. Конечно, если производитель виртуальной машины считает возможным, он может обеспечить атомарность операций и над этими типами. Для volatile переменных это является обязательным требованием.

4.3. Блокировки

В основном хранилище для каждого объекта поддерживается блокировка (lock), над которой можно произвести два действия - установить (lock) и снять (unlock). Только один поток в один момент времени может установить блокировку на некоторый объект. Если до того, как этот поток выполнит операцию unlock, другой поток попытается установить блокировку, его выполнение будет приостановлено до тех пор, пока первый поток не отпустит ее.

Операции lock и unlock накладывают жесткое ограничение на работу с переменными в рабочей памяти потока. После успешно выполненного lock, рабочая память очищается, и все переменные необходимо заново считывать из основного хранилища. Аналогично, перед операцией unlock необходимо все переменные сохранить в основном хранилище.

Важно подчеркнуть, что блокировка является чем-то вроде флага. Если блокировка на объект установлена, это не означает, что этим объектом нельзя пользоваться, что его поля и методы становятся недоступными - это не так. Единственное действие, которое становится невозможным - установить эту же блокировку другому потоку до тех пор, пока первый поток не выполнит unlock.

В Java-программе для того, чтобы воспользоваться механизмом блокировок, существует ключевое слово synchronized. Оно может быть применено в двух вариантах - для объявления synchronized-блока и как модификатор метода. В обоих случаях действие его примерно одинаковое.

Synchronized-блок записывается следующим образом:

```
synchronized (ref) {
  ...
}
```

Прежде чем начать выполнять действия, описанные в этом блоке, поток обязан установить блокировку на объект, на который ссылается переменная ref (поэтому она не может быть null). Если другой поток уже установил блокировку на этот объект, то выполнение первого потока приостанавливается до тех пор, пока не удастся выполнить операцию lock.

После этого блок выполняется. В случае успешного либо не успешного завершения исполнения, производится операция unlock, чтобы освободить объект для других потоков.

Рассмотрим пример:

```
public class ThreadTest  implements Runnable {

 private static ThreadTest shared = new ThreadTest();

 public void process() {
  for (int i=0; i<3; i++) {
   System.out.println (Thread.currentThread().
getName()+" "+i);
   Thread.yield();
  }
 }

 public void run() {
  shared.process();
 }

 public static void main(String s[]) {
  for (int i=0; i<3; i++) {
   new Thread(new ThreadTest(),
"Thread-"+i).start();
  }
 }
}
```

В этом простом примере три потока вызывают метод у одного объекта, чтобы тот распечатал три значения. Результатом будет:

```
Thread-0 0
Thread-1 0
Thread-2 0
Thread-0 1
Thread-2 1
Thread-0 2
Thread-1 1
Thread-2 2
Thread-1 2
```

То есть, все потоки одновременно работают с одним методом одного объекта. Заключим обращение к методу в synchronized-блок:

```
  public void run() {
synchronized (shared) {
shared.process();
}
}
```

Теперь результат будет строго упорядочен:

```
Thread-0 0
Thread-0 1
Thread-0 2
Thread-1 0
Thread-1 1
Thread-1 2
Thread-2 0
Thread-2 1
Thread-2 2
```

Synchronized-методы работают аналогичным образом. Прежде чем начать выполнять их, поток пытается заблокировать объект, у которого вызывается метод. После выполнения блокировка снимается. В предыдущем примере аналогичной упорядоченности можно было добиться, если использовать не synchronized-блок, а объявить метод process() синхронизированным.

Также допустимы static synchronized методы. При их вызове блокировка устанавливается на объект класса Class, отвечающего за тип, у которого вызывается этот метод.

При работе с блокировками всегда надо помнить о возможности появления deadlock - взаимных блокировок, которые приводят к зависанию программы. Если один поток заблокировал один ресурс, и пытается заблокировать второй, а другой поток заблокировал второй и пытается заблокировать первый, то такие потоки уже никогда не выйдут из состояния ожидания.

Рассмотрим простейший пример:

```
public class DeadlockDemo {

 // Два объекта-ресурса
 public final static Object one=new Object(), two=new Object();

 public static void main(String s[]) {
  // Создаем два потока, которые будут
  // конкурировать за доступ к объектам one и two
  Thread t1 = new Thread() {
   public void run() {
    // Блокировка первого объекта
    synchronized(one) {
     Thread.yield();
     // Блокировка второго объекта
     synchronized (two) {
      System.out.println("Success!");
     }
    }
   }
  };
  Thread t2 = new Thread() {
```

```
   public void run() {
    // Блокировка второго объекта
    synchronized(two) {
     Thread.yield();
     // Блокировка первого объекта
     synchronized (one) {
      System.out.println("Success!");
     }
    }
   }
  };

  // Запускаем потоки
  t1.start();
  t2.start();
 }
}
```

Если запустить такую программу, то она никогда не закончит свою работу. Обратите внимание, на вызовы метода yield() в каждом потоке. Они гарантируют, что когда один поток выполнил первую блокировку и переходит к следующей, второй поток находится в таком же состоянии. Легко понять, что в результате оба потока "замрут", не смогут продолжить свое выполнение. Первый поток будет ждать освобождение второго объекта, и наоборот. Именно такая ситуация называется "мертвой блокировкой", или deadlock. Если один из потоков успел бы заблокировать оба объекта, то программа успешно бы выполнилась до конца. Однако многопоточная архитектура не дает никаких гарантий, как именно потоки будут выполняться друг относительно друга. Задержки (которые в примере моделируются вызовами yield()) могут возникать из логики программы (необходимость произвести вычисления), действий пользователя (не сразу нажал кнопку "ОК"), занятости ОС (из-за нехватки физической оперативной памяти пришлось воспользоваться виртуальной), значений приоритетов потоков и так далее.

В Java нет никаких средств распознавания или предотвращения ситуаций deadlock. Также нет способа перед вызовом синхронизированного метода узнать, заблокирован ли уже объект другим потоком. Программист сам должен строить работу программы таким образом, чтобы неразрешимые блокировки не возникали. Например, в рассмотренном примере достаточно было организовать блокировки объектов в одном порядке (всегда сначала первый, затем второй), и программа выполнялась бы всегда успешно.

Опасность возникновения взаимных блокировок заставляет с особенным вниманием относиться к работе с потоками. Например, важно помнить, что если у объекта потока был вызван метод sleep(..), то такой поток будет бездействовать определенное время, но при этом все заблокированные ими объекты будут оставаться недоступными для блокировок со стороны других потоков, а это потенциальный deadlock. Такие ситуации крайне сложно выявить тестированием и отладкой, поэтому вопросам синхронизации надо уделять много времени на этапе проектирования.

5. Методы wait(), notify(), notifyAll() класса Object

Наконец, переходим к рассмотрению трех методов класса Object, которое завершает описание механизмов поддержки многопоточности в Java.

Каждый объект в Java имеет не только блокировку для synchronized блоков и методов, но и так называемый wait-set, набор потоков исполнения. Любой поток может вызвать метод wait() любого объекта и таким образом попасть в его wait-set. При этом выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у точно этого же объекта метод notifyAll(), который пробуждает все потоки из wait-set. Метод notify() пробуждает один, случайно выбранный поток из этого набора.

Однако применение этих методов связано с одним важным ограничением. Любой из них может быть вызван потоком у объекта только после установления блокировки на этот объект. То есть, либо внутри synchronized-блока с ссылкой на этот объект в качестве аргумента, либо обращение к методам должны быть в синхронизированных методах класса самого объекта. Рассмотрим пример:

```
public class WaitThread implements Runnable {
  private Object shared;

  public WaitThread(Object o) {
   shared=o;
}

public void run() {
 synchronized (shared) {
  try {
  shared.wait();
    } catch (InterruptedException e) {}
  System.out.println("after wait");
 }
}

public static void main(String s[]) {
 Object o = new Object();
 WaitThread w = new WaitThread(o);
 new Thread(w).start();
 try {
  Thread.sleep(100);
} catch (InterruptedException e) {} System.out.println("before notify");
 synchronized (o) {
o.notifyAll();
   }
}
}
```

Результатом программы будет:

```
before notify
after wait
```

Обратите внимание, что метод wait(), также как и sleep(), требует обработки InterruptedException, то есть его выполнение также можно прервать методом interrupt().

В заключение рассмотрим более сложный пример для трех потоков:

```
public class ThreadTest implements Runnable {
  final static private Object shared=new Object();

  private int type;
  public ThreadTest(int i) {
   type=i;
  }

public void run() {
 if (type==1 || type==2) {
  synchronized (shared) {
   try {
    shared.wait();
   } catch (InterruptedException e) {}
   System.out.println("Thread "+type+
" after wait()");
  }
 } else {
  synchronized (shared) {
   shared.notifyAll();
   System.out.println("Thread "+type+
" after notifyAll()");
  }
   }
}

public static void main(String s[]) {
 ThreadTest w1 = new ThreadTest(1);
 new Thread(w1).start();
 try {
  Thread.sleep(100);
 } catch (InterruptedException e) {}

 ThreadTest w2 = new ThreadTest(2);
 new Thread(w2).start();
 try {
  Thread.sleep(100);
 } catch (InterruptedException e) {}
```

```
 ThreadTest w3 = new ThreadTest(3);
 new Thread(w3).start();

}
}
```

Результатом работы программы будет:

```
Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()
```

Рассмотрим, что происходило. Во-первых, был запущен поток 1, который тут же вызвал метод wait() и приостановил свое выполнение. Затем то же самое произошло с потоком 2. Далее начинает выполняться поток 3.

Сразу обращает на себя внимание следующий факт. Еще поток 1 вошел в synchronized-блок, а стало быть установил блокировку на объект shared. Но судя по результатам видно, что это не помешало и потоку 2 затем зайти в synchronized-блок, а потом и потоку 3. Причем для последнего это просто необходимо, иначе как можно "разбудить" потоки 1 и 2?

Можно сделать вывод, что потоки, прежде чем приостановить выполнение после вызова метода wait(), отпускают все занятые блокировки. Итак, вызывается метод notifyAll(). Как уже было сказано, все потоки из wait-set возобновляют свою работу. Однако чтобы корректно продолжить исполнение необходимо вернуть блокировку на объект, ведь следующая команда также находится внутри synchronized-блока!

Получается, что даже после вызова notifyAll() все потоки не могут сразу возобновить работу. Лишь один из них сможет вернуть себе блокировку и продолжить работу. Когда он покинет свой synchronized-блок и отпустит объект, второй поток возобновит свою работу и так далее. Если по какой-то причине объект так и не будет освобожден, поток так никогда и не выйдет из метода wait(), даже если будет вызван метод notifyAll(). В рассмотренном примере потоки один за другим смогли возобновить свою работу.

Кроме этого определен метод wait() с параметром, который задает период тайм-аута, по истечении которого поток сам попытается возобновить свою работу. Но начать ему придется все равно с повторного получения блокировки.

6. Контрольные вопросы

12-1. Каким образом на однопроцессорной машине исполняются многопоточные приложения?

a.) Операционная система разделяет рабочее время процессора на небольшие интервалы. В начале каждого интервала решается, какая из задач будет выполняться на его протяжении. Поскольку процессор переключается между задачами очень быстро, возникает ощущение, что они выполняются одновременно. Этот прием называется time-slicing.

12-2. Какие преимущества дает многопоточная архитектура?

а.) Основные преимущества:

- Если сама проектируемая система предназначена для выполнения нескольких действий одновременно (например, сервер, обслуживающий нескольких клиентов сразу), то, воспользовавшись встроенной многозадачностью, можно существенно упростить архитектуру системы, повысив ее надежность и быстродействие.
- Если необходимо выполнить несколько действий, которые требуют различных аппаратных и других ресурсов (память, жесткий диск, сеть, принтер, монопольный доступ к БД и т.д.), то одновременное выполнение этих задач потребует меньше времени, чем выполнение их по отдельности.
- Воспользовавшись свойством приоритетов потоков можно настроить систему так, чтобы она была наиболее удобна пользователю. Пусть формально процессор выполнит меньше вычислений, но они будут более полезны клиенту.

12-3. Что такое приоритет потока?

а.) Приоритет – это характеристика, как правило, числовая, которая влияет на распределение интервалов работы процессора между задачами. Предсказать количественный эффект от изменения приоритета в ту или иную сторону нельзя, но задача с большим приоритетом всегда получает больше времени, чем с меньшим приоритетом.

12-4. Что такое демон-поток?

а.) Демон-поток – это обычный поток. Единственно отличие от не-демонов заключается в том, что виртуальная машина закрывается, когда остаются рабочими только демон-потоки.

12-5. Когда закрывается виртуальная машина, выполняющая программу с несколькими потоками исполнения?

а.) Когда остаются рабочими только демон-потоки, либо не остается ни одного потока.

12-6. Для чего служит в Java класс Thread?

а.) Класс Thread служит для запуска потока, изменения его свойств и дальнейшего управления им (приостановка, возобновление работы, остановка и т.д.).

12-7. Поскольку интерфейс Runnable представляет собой альтернативный способ программировать потоки исполнения, можно ли в такой программе обойтись вовсе без класса Thread?

а.) Нет, поскольку для создания и запуска потока всегда необходимо создавать экземпляр класса Thread или его наследника. Также он необходим для дальнейшего управления потоком.

12-8. Что такое локальное и главное хранилища в Java? Чем они различаются?

a.) JVM поддерживает одно главное хранилище, в котором хранятся значения всех переменных, созданных программой. Локальное хранилище создается для каждого потока. После создания оно пустое, и поток должен скопировать те переменные, с которыми работает, в свою локальную память. Взаимодействие между главным и локальными хранилищами строго регламентировано. В частности, любая переменная создается сначала в главной, а затем копируется в локальную память.

12-9. Если один поток начал исполнение synchronized-блока, указав ссылку на некий объект, может ли другой поток обратиться к полю этого объекта? К методу?

a.) К любому полю можно обратиться беспрепятственно, равно как и к не-synchronized методу. Вызов synchronized-метода потребует установки блокировки, а до этого второй поток будет приостановлен.

12-10. Если объявить метод synchronized, то какой эффект будет этим достигнут?

a.) Гарантируется, что в один момент времени только один поток может его выполнять.

12-11. Как работают static synchronized методы?

a.) Аналогично обычным synchronized методам, только блокировка устанавливается не на объект класса, а на объект класса Class, описывающий исходный класс. Таким образом, гарантируется, что в один момент времени только один поток может работать со static synchronized методами класса.

12-12. Почему метод wait требует обработки InterruptedException, а методы notify и notifyAll – нет?

a.) Потому что вызов метода wait приводит к приостановке потока, работу которого можно возобновить, вызвав метод interrupt(), что и приведет к возникновению исключительной ситуации InterruptedException.

Методы notify и notifyAll не приостанавливают работу потока.

12-13. Может ли поток никогда не выйти из метода wait, даже если будет вызван метод notify? notifyAll?

a.) Поскольку метод wait может быть корректно вызван только при наличии блокировки на объект, а после успешного вызова поток эту блокировку освобождает, то для возобновления работы поток должен снова установить блокировку. Если по каким-то причинам она постоянно «занята», то поток никогда не выйдет из метода wait.

12-14. Какой будет результат работы следующего кода?

```
public abstract class Test implements Runnable {
 private Object lock = new Object();

 public void lock() {
  synchronized (lock) {
```

```
    try {
     lock.wait();
     System.out.println("1");
    } catch (InterruptedException e) {
    }
   }
  }

  public void unlock() {
   synchronized (lock) {
    lock.notify();
    System.out.println("2");
   }
  }

  public static void main(String s[]) {
   new Thread(new Test() {
    public void run() {
     lock();
    }
   }).start();
   new Thread(new Test() {
    public void run() {
     unlock();
    }
   }).start();
  }
 }
```

a.) На консоли появится только число 2, а виртуальная машина никогда не завершит работу, поскольку первый поток никогда не выйдет из метода wait. Хотя второй поток и вызывает метод notify, однако потоки работают с разными объектами (у каждого свое поле lock), что и приводит к описанному эффекту.