МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования

«Национальный исследовательский

Нижегородский государственный университет им. Н.И. Лобачевского»

С.П.Никитенкова

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA

Учебно-методическое пособие

Рекомендовано методической комиссией радиофизического факультета
для студентов ННГУ, обучающихся по направлению подготовки
02.03.02 «Фундаментальная информатика и информационные технологии»

Нижний Новгород

2015

УДК 004.4'2
ББК 32.973-018я7
Н 62

Рецензент: д.ф.-м.н., проф. Юнаковский А.Д.

Н 62 Никитенкова С.П. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA: Учебно-методическое пособие – Нижний Новгород : Нижегородский университет, 2015 -90с.

Учебно-методическое пособие представляет собой начальный курс по изучению многопоточного программирования. На практических примерах, сопровождаемых теоретическими сведениями, рассматривается процесс создания и реализации многопоточных приложений в рамках возможностей, предоставляемых языком программирования Java. Особое внимание уделяется организации синхронизации потоков при разработке многопоточных приложений. Рассмотрены основные проблемы, возникающие при разработке многопоточных программ: “гонка” данных, обнаружение взаимоблокировки, бесконечное ожидание и т.д. и их решение средствами языка Java. Даны понятия «потокобезопасность» и «потокобезопасного» класса. Приведены примеры параллельного программирования алгоритмов решения различных типовых задач, возникающих при исследованиях математических моделей физических явлений, такие как параллельное умножение матриц, адаптивная квадратура. Пособие содержит контрольные вопросы и задания, которые могут быть использованы в процессе обучения и при проведении лабораторных работ.

Учебно-методическое пособие подготовлено в соответствии с Программой повышения конкурентоспособности Нижегородского государственного университета им.Н.И. Лобачевского среди ведущих мировых научно-образовательных центров на 2013 – 2020 годы, программой повышения конкурентоспособности ННГУ им. Н.И. Лобачевского Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений», мероприятия 7.1.1. «Разработка образовательных программ подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий и высокопроизводительных вычислений».

Данное пособие предназначено для студентов младших курсов, обучающихся по направлению подготовки 02.03.02 «Фундаментальная информатика и информационные технологии»

Ответственный за выпуск

зам. председателя методической комиссии радиофизического факультета

д.ф.-м.н., проф. Грибова Е.З.

УДК 004.4'2

ББК 32.973-018я7

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время программы по развитию и внедрению суперкомпьютерных технологий и грид-сетей в промышленность входят в число наиболее приоритетных.

Неотъемлемой частью программы развития суперкомпьютеров является разработка соответствующего программного обеспечения, что требует практических знаний параллельного и распределенного программирования. Чтобы программа выигрывала от увеличения вычислительной мощности, ее необходимо проектировать как набор параллельно выполняющихся задач.

Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Параллельное программирование становится в последнее время жизненной необходимостью, которая диктуется темпами развития многоядерных процессоров. Многопоточность обеспечивает возможность параллельного выполнения нескольких видов операций в одной прикладной программе. Параллельные вычисления реализуется на уровне потоков, и программа, оформленная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет параллельного выполнения ее отдельных частей.

Многопоточность представляет собой ключевую модель параллелизма, поддерживаемую современными компьютерами, языками программирования и операционными системами. Многопоточность — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно». Такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Впервые многопоточность была широко использована советскими разработчиками компьютерной аппаратуры и программистами в 1970-е годы. Многопроцессорный вычислительный комплекс "Эльбрус-1", разработанный в 1979 году, поддерживал в аппаратуре и операционной системе эффективную концепцию процесса, которая была близка к современному понятию облегченного процесса.

Концепция многопоточности начала складываться с 1980-х гг. в системе UNIX. Далее, в середине 1990-х гг. была выпущена ОС Windows NT, в которую была также включена многопоточность. Однако в разных операционных системах интерфейс прикладного программирования (application programming interface) для многопоточности существенно

отличался. Поэтому многопоточные программы, в том числе написанные на языках высокого уровня, оказались не переносимыми с одной платформы на другую.

Язык программирования Java является первым массовым языком программирования, который явно включает в свой состав потоки, а не рассматривает их как функцию нижележащей операционной системы. Именно в Java впервые многопоточность была реализована на уровне конструкций языка и базовых библиотек.

В настоящее время Java - это один из наиболее распространённых языков программирования, при помощи которого работают миллиарды устройств. Язык программирования Java объединяет возможности объектно-ориентируемых и параллельных языков, а также является идеальным инструментом, использующим различные сетевые технологии. Язык Java поддерживает как многопоточное, так и распределенное программирование, когда программа выполняется на разных компьютерах, связываемых по сети.

ГЛАВА 1.
МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

Целью многопоточного программирования является эффективная организация программы через взаимодействие отдельных компонент.

Понятие процесса является одним из основополагающих в теории и практике многопоточного программирования. Процесс, это выполняющийся экземпляр программы (в такой форме процессы иногда называют тяжеловесными процессами). Программа может состоять из одного процесса, но может и из нескольких, например, браузер Chrome создает отдельный процесс для каждой вкладки. Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств). Каждый процесс имеет собственное адресное пространство, память и стек данных, а также может использовать другие вспомогательные данные для контроля над выполнением. Операционная система управляет выполнением всех процессов в системе, выделяя каждому процессу процессорное время по определенному принципу. Операционная система также отвечает за то, как виртуальное пространство процесса проецируется на физическую память. В ходе выполнения процесса может происходить ветвление или порождение новых процессов для осуществления других задач, но каждый новый процесс имеет собственную

память, стек данных и т.д. Отдельные процессы не могут иметь доступ к общей информации, если не реализовано межпроцессное взаимодействие (interprocess communication) в той или иной форме.

Потоки (иногда называемые легковесными процессами) подобны процессам, за исключением того, что все они выполняются в пределах одного и того же процесса, следовательно, используют один и тот же контекст. Все потоки, организованные в одном процессе, используют общее пространство данных с основным потоком, поэтому могут обмениваться информацией или взаимодействовать друг с другом с меньшими сложностями по сравнению с отдельными процессами. Потоки, как правило, выполняются параллельно. Именно распараллеливание и совместное использование данных становятся предпосылками обеспечения координации выполнения нескольких задач. Одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система периодически переключается между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В многоядерных процессорах на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения.

Потоки работают в общем адресном пространстве и, тем самым, разделяют данные программы. Следует отметить, что общность данных, с одной стороны, существенно упрощает организацию взаимодействия потоков (результат, вычисленный одним потоком, сразу становится доступным всем остальным потокам программы), но, с другой стороны, требует соблюдения определенных правил использования разделяемых ресурсов (общих данных, файлов, устройств и т. п.). Для организации разделения ресурсов между несколькими потоками необходимо иметь возможность: определения доступности запрашиваемых ресурсов (ресурс свободен и может быть выделен для использования, ресурс уже занят одним из потоков программы и не может использоваться дополнительно каким-либо другим потоком); выделения свободного ресурса одному из потоков, запросивших ресурс для использования; приостановки (блокировки) потоков, выдавших запросы на ресурсы, занятые другими потоками.

1.1. СОЗДАНИЕ ПОТОКА

Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач. Каждая задача выполняется в независимом потоке. Любая программа на языке Java содержит как минимум один поток. «Главный» (main thread) запускается методом main() приложения. Код, исполняемый в методе main() может запустить другие потоки. Когда останавливается «главный» поток, программа завершается.

Потоки представляют собой классы, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существуют два способа создания и запуска потока: расширение класса Thread или реализация интерфейса Runnable.

```
class MyRunnable implements Runnable
{
  public void run() {
    //execution starts here
  }
}
…
Runnable target =  new MyRunnable();
Thread t = new Thread (target, “one”);
t.start();
```

```
class MyThread extends Thread
{
  public void run() {
    //execution starts here
  }
}
…
MyThread t = new MyThread();
t.setName(“two”);
t.start();
```

Рассмотрим более подробно создание потока на основе класса Thread

Пример1.

```
class A extends Thread {
  public void run() {
```

```
      for ( int i = 1; i <= 5; i++ ) {
        System.out.println("Thread A ... i = " + i);      }
      System.out.println("Exit from Thread A : Execution over");   }}
/* определим класс 'B' который наследуется от класса 'Thread'
 * ( создаем 2-ой поток )
 */
class B extends Thread {
  public void run() {
      for ( int i = 11; i <= 15; i++ ) {
        System.out.println("Thread B ... i = " + i);      }
      System.out.println("Exit from Thread B : Execution over");   }
}
public class ThreadDemo {
  public static void main(String args[]) {
      A thread_a = new A();
      B thread_b = new B();
      thread_a.start(); /* стартуем 1-ый поток ( start() – метод класса Thread) */
      thread_b.start(); /* стартуем 2-ый поток */
      /* можно заменить приведенные выше конструкции  на:
        new A().start();
        new B().start();
      */
  }}
```

Результат работы программы, приведенной выше, непредсказуем и зависит от переключений между двумя потоками. Один из возможных результатов может быть:

Thread B ... i = 11

Thread B ... i = 12

Thread A ... i = 1

Thread B ... i = 13

Thread A ... i = 2

Thread B ... i = 14

Thread A ... i = 3

Thread B ... i = 15

Exit from Thread B : Execution over

Thread A ... i = 4

Thread A ... i = 5

Exit from Thread A : Execution over

При повторном запуске программы результат может быть другим.

Java не поддерживает множественное наследование классов, но позволяет реализовывать множество интерфейсов. Это означает, что лучше реализовывать интерфейс Runnable, если предполагается наследование от другого класса. При реализации интерфейса Runnable необходимо определить его единственный абстрактный метод run(). Интерфейс Runnable не имеет метода start(), а только единственный метод run(). Поэтому для запуска потока следует создать объект класса Thread и передать объект потока его конструктору. Однако при прямом вызове метода run() поток не запустится, выполнится только тело самого метода.

```
class A implements Runnable {
  public void run() { //  run() метод интерфейса Runnable
    for ( int i = 1; i <= 5; i++ ) {
      System.out.println("Thread A ... i = " + i);      }
    System.out.println("Exit from Thread A : Execution over");   }
```

```
}
/* класс 'B' также реализует интерфейс 'Runnable'
* ( создаем 2-ой поток )
 */
class B implements Runnable {
  public void run() {
    for ( int i = 6; i <= 10; i++ ) {
      System.out.println("Thread B ... i = " + i);      }
    System.out.println("Exit from Thread B : Execution over");   }
}
public class RunnableDemo {
  public static void main(String args[]) {
    A obj_a = new A();
    B obj_b = new B();
    Thread thread_a = new Thread(obj_a);
    Thread thread_b = new Thread(obj_b);
    thread_a.start(); // старт 1-ого потока
    thread_b.start(); // старт 2-ого потока
    /* можно заменить приведенные выше конструкции  на :-
      new Thread(new A()).start();
      new Thread(new B()).start();
    */
  }       }
```

Метод start() используется для запуска нового потока. Несмотря на то, что start() вызывает метод run() внутри себя, это не то же самое, что просто вызвать run(). Если

вызывать run() как обычный метод, он вызывается в том же потоке и никакой новый поток не запуститься, что происходит, когда вызывается метод start().

1.2. ЖИЗНЕННЫЙ ЦИКЛ ПОТОКА

Поток Java может находиться в одном из следующих состояний в течение периода существования:

Состояние Новый (New). После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.

Состояние Работоспособный (Runnable). Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован. Когда поток находится в этом состоянии, он считается живым.

Состояние Работающий (Running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент. У JVM (Java виртуальная машина) свой планировщик потоков, который не зависит от планировщика операционной системы под которой работает JVM.

Состояние Живой, но не работоспособный (Alive, but not runnable). Поток может быть живым, но не работоспособным по нескольким причинам. Он может быть в состояниях Ожидания, Сна или Блокировки.

Ожидание (Waiting). Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный. Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах, как показано ниже:

Блокировка (Blocked). Поток может перейти в это состояние в ожидании ресурса, такого как ввод/вывод, или из-за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен.

Мёртвый (Dead). Поток считается мёртвым, когда его метод run() полностью выполнен. Мёртвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start().

Получить значение состояния потока можно вызовом метода getState(). Для получения текущего потока нужно воспользоваться Thread.currentThread().

Ниже приведены примеры, иллюстрирующие использование методов sleep()и yield(). Методы sleep() и yield() – статические, действуют на текущий поток. Приостановить работу другого потока нельзя. Метод sleep() приостанавливает выполнение потока на заданное время. Метод yield() отдает управление другим потокам, без указания времени, когда управление должно вернуться к текущему потоку. Метод yield() заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще. Распределение времени особенно важно для программ с потоками, которые работают в цикле или тратят значительное время на математические расчеты без ввода-вывода. В этих случаях простой вызов yield() достаточен для того, чтобы дать возможность работать другим потокам с тем же приоритетом.

```
class A extends Thread {
  public void run() {
    for ( int i = 1; i <= 5; i++ ) {
      System.out.println("Thread A ... i = " + i);
      if ( i == 3 ) {
        try {
          sleep(800); // поток засыпает на 800 ms
        } catch(Exception e) { }
      }     }
    System.out.println("Exit from Thread A : Execution over");
  } }

class B extends Thread {
  public void run() {
```

```
    for ( int i = 11; i <= 15; i++ ) {
      System.out.println("Thread B ... i = " + i);
      if ( i == 12 ) {
        yield(); // передает управление другому потоку
}     }
    System.out.println("Exit from Thread B : Execution over");
  }  }

class C extends Thread {
  public void run() {
    for ( int i = 21; i <= 25; i++ ) {
      System.out.println("Thread C ... i = " + i);
      if ( i == 23 ) {
        stop(); // поток остановлен
      }     }
    System.out.println("Exit from Thread C : Execution over");
  }   }

public class StopBlockDemo {
  public static void main(String args[]) {
    new A().start();
    new B().start();
    new C().start();
  }  }
```

В классе Object есть три метода wait(), notify(), notifyAll(), которые позволяют потоку сообщать информацию о своем состоянии другим, заинтересованным в этой информации, потокам. Таким образом, каждый объект в Java так называемый wait-set набор потоков исполнения и может быть монитором. 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 этого же объекта метод notifyAll(), который пробуждает все потоки. Метод notify() пробуждает один случайно выбранный поток из данного набора. Метод notify() освобождает все блокировки, удерживаемые текущим потоком, тогда как метод sleep() этого не делает. Поток может вызвать методы wait() или notify() для определённого объекта, только если он в данный момент имеет блокировку на этот объект. wait(), notify() и notifyAll() должны вызываться только из синхронизированного кода. 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 обращения к методам должны быть в синхронизированных методах класса самого объекта. Любой объект может служить блокировкой. Снятие блокировки производится автоматически. Синтаксис:

synchronized (o) { // Получение блокировки

...

}// Снятие блокировки

Программы, иллюстрирующие использование методов wait()и notify() приведены ниже:

```
class SumAvg {
  int set[];
  int sum = 0;
  boolean sum_computed = false;
  SumAvg(int arr[]) {
    set = arr;   }
  void sum() {
```

```
    synchronized(this) { // текущий объект приобретает блокировку
      try {
        System.out.println("Entered sum() method");
        for ( int i = 0; i < set.length; i++ ) {
          sum += set[i];          }
        sum_computed = true;
        notify(); /* освобождает поток, ожидающий вычисления суммы */
      } catch(Exception e) { }
    }   }
  void average() {
    synchronized(this) { // блокировка текущего объекта
      try {
        System.out.println("Entered average() method");
        while(!sum_computed) {
          System.out.println("Waiting for sum to be computed ");
          wait(); // вызов метода wait()приостанавливает текущий поток
          System.out.println("Sum has been computed ");
        }
        float avg = (float)sum / set.length;
        System.out.println("Avg = " + avg);
      } catch(Exception e) { }
    }   }   }
/* Этот вспомогательный класс вычисления суммы всех элементов */
class SumHelper extends Thread {
```

```
  SumAvg sa;
  SumHelper(SumAvg obj) {
    sa = obj;   }
  public void run() {
    sa.sum();   }
}
/*  Этот вспомогательный класс вычисления среднего значения всех элементов */
class AvgHelper extends Thread {
  SumAvg sa;
  AvgHelper(SumAvg obj) {
    sa = obj;   }
  public void run() {
    sa.average();   }
}
public class WaitNotifyDemo {
  public static void main(String args[]) {
    int arr[] = { 56, 43, 12, 9, 39 };
    SumAvg sa = new SumAvg(arr);
    /* создаем различные потоки для вычисления суммы и среднего значения*/
    SumHelper sh = new SumHelper(sa); // поток для вычисления суммы
    AvgHelper ah = new AvgHelper(sa); // поток для вычисления среднего
    ah.start();
    sh.start();
  }   }
```

В приведенной выше программе используются два потока, чтобы вычислить сумму элементов массива и среднее значение элементов массива. Для корректного вычисления среднего значения сумма должна быть вычислена первой. Поэтому поток вычисления среднего блокируется, используя метод wait(), и блокировка снимается, используя метод notify(), только когда сумма будет вычислена. Синхронизация потоков выполнена с помощью ‘this', что означает, что любому другому потоку, который использует тот же объект, придется ждать, пока блокировка с объекта не будет снята.

В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.

Метод join() имеет перегруженную версию, которая получает в качестве параметра время ожидания. В этом случае join() возвращает управление либо когда завершится ожидаемый поток, либо когда закончится время ожидания. Подобно методу sleep() метод join() может ждать в течение заданного отрезка.

```
public class MyThread extends Thread {
public void run()   {
      System.out.println("r1 ");
      try{
       Thread.sleep(500);
   }catch(InterruptedException ie){}
            System.out.println("r2 ");  }
public static void main(String[] args){
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
 try{
```

```
t1.join();                    // ожидаем, пока поток t1 завершится
}catch(InterruptedException ie){}
t2.start();  }    }
```

Методы wait(), notify() и notifyAll() никогда не переопределяются и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации IllegalMonitorStateException. Методы sleep() и join() обёрнуты в конструкции try-catch. Это необходимое условие работы этих методов. Вызывающий их код должен перехватывать исключение InterruptedException, которое они выбрасывают при прерывании во время ожидания.

Методы suspend(), resume() и stop() являются deprecated-методами (сомнительными, т.е. использование которых в программе необоснованно) из-за потенциальных угроз взаимной блокировки. Методы запрещены к использованию, так как они не являются в полной мере “потокобезопасными”. “Потокобезопасность” – свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается.

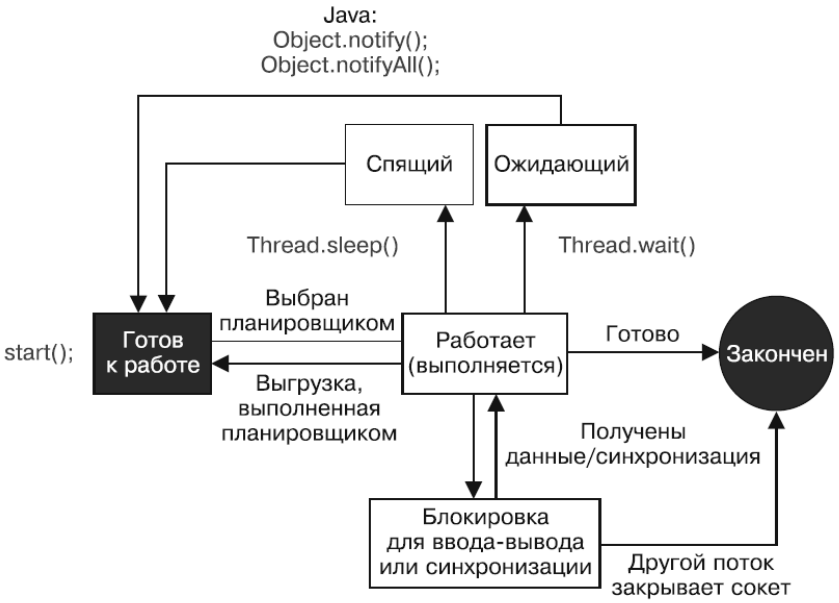


Рис.1. Жизненный цикл потока

На рисунке 1 показано, как протекает жизненный цикл потока — от создания к запуску, возможной приостановке, блокированию на ресурсе или возобновлению работы и, наконец, к завершению.

1.3. УПРАВЛЕНИЕ ПРИОРИТЕТАМИ И ГРУППЫ ПОТОКОВ

Потоку можно назначить приоритет от 1 (константа MIN_PRIORITY) до 10 (MAX_PRIORITY) с помощью метода setPriority(int prior). Получить значение приоритета можно с помощью метода getPriority().

```
public class PriorThread extends Thread {
public PriorThread(String name){
super(name); }
public void run(){
for (int i = 0; i < 71; i++){
System.out.println(getName() + " " + i);
try {
sleep(1);//
} catch (InterruptedException e) {
System.err.print("Error" + e); } } } }
public class PriorityRunner {
public static void main(String[] args) {
PriorThread min = new PriorThread("Min");//1
PriorThread max = new PriorThread("Max");//10
PriorThread norm = new PriorThread("Norm");//5
min.setPriority(Thread.MIN_PRIORITY);
max.setPriority(Thread.MAX_PRIORITY);
```

```
norm.setPriority(Thread.NORM_PRIORITY);

min.start();

norm.start();

max.start(); } }
```

Поток с более высоким приоритетом в данном случае, как правило, монополизирует вывод на консоль.

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

```
ThreadGroup tg = new ThreadGroup("Группа потоков 1");

Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные группой, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод getThreadGroup(). Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения значение его приритета станет равным приоритету группы. Поток же со значением приоритета более низким, чем приоритет группы после включения в группу, значения своего приоритета не изменит.

```
class ThreadGroupDemo {

  public static void main (String [] args)    {

    ThreadGroup tg = new ThreadGroup ("subgroup 1");

    Thread t1 = new Thread (tg, "thread 1");

    Thread t2 = new Thread (tg, "thread 2");

    Thread t3 = new Thread (tg, "thread 3");

    tg = new ThreadGroup ("subgroup 2");

    Thread t4 = new Thread (tg, "my thread");

    tg = Thread.currentThread ().getThreadGroup ();

    int agc = tg.activeGroupCount ();
```

```
System.out.println ("Active thread groups in " + tg.getName () +
                    " thread group: " + agc);
tg.list ();   }  }
```

1.4. ПОТОКИ-ДЕМОНЫ

В Java есть два вида потоков: потоки - демоны (daemon threads) и пользовательские потоки (user threads). JVM завершает выполнение программы тогда, когда все пользовательские потоки завершат свое выполнение.

Демоном называется поток, предоставляющий некоторый сервис, работающий в фоновом режиме во время выполнения программы, но при этом не являющийся ее неотъемлемой частью. Таким образом, когда все потоки не-демоны заканчивают свою деятельность, программа завершается. И наоборот, если существуют работающие потоки не-демоны, программа продолжает выполнение. Существует, например, поток не-демон, выполняющий метод main(). Потоки-демоны не препятствуют завершению работы программы.

```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);      }
    } catch(InterruptedException e) {      print("sleep() interrupted");    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < 10; i++) {
      Thread daemon = new Thread(new SimpleDaemons());
      daemon.setDaemon(true); // Необходимо вызвать перед start()
```

```
    daemon.start();     }
  print("All daemons started");
  TimeUnit.MILLISECONDS.sleep(175);  }  }
```

1.5. ПУЛ ПОТОКОВ (THREAD POOL)

Поток является относительно дорогим ресурсом и его создание может быть сопряжено со значительными накладными расходами в плане времени и ресурсов. При разработке программ часто возникает необходимость в параллельной обработке задач или запросов, при этом создание потока каждый раз, когда это нужно оказывается неэффективной стратегией.

Для решения подобной проблемы можно воспользоваться «пулом потоков». Пул потоков - это компонент, управляющий группой потоков, доступных для выполнения некоторой работы. Вместо создания потока для выполнения какой-то определенной задачи, задание помещается в очередь, а компонент извлекает его оттуда и передает первому свободному потоку для выполнения. Пулы потоков помогают снизить затраты на создание и уничтожение потоков для очень коротких заданий, помогают избежать монополизации ресурсов и перегрузки системы, ограничивая общее количество потоков, используемых приложением, и автоматизируют принятие решения о выборе оптимального количества потоков для решения данной задачи.

Начиная с Java 1.5 Java API предоставляет фреймворк Executor, который позволяет создавать различные пулы потоков, например single thread pool, который обрабатывает только одно задание за единицу времени, fixed thread pool, пул с фиксированным количеством нитей, и cached thread pool, расширяемый пул, подходящий для приложений с множеством недолгих заданий.

```
public class MyRunnable implements Runnable {
  private final long countUntil;
  MyRunnable(long countUntil) {
    this.countUntil = countUntil;  }
    @Override
  public void run() {
    long sum = 0;
    for (long i = 1; i < countUntil; i++) {
      sum += i;    }
    System.out.println(sum);}}
```

Интерфейс ExecutorService является сервисом для запуска потоков, который позволяет управлять коллективным выполнением и окончанием асинхронных задач.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main {
  private static final int NTHREDS = 10;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Runnable worker = new MyRunnable(10000000L + i);
      executor.execute(worker);
    }
    // Завершить все существующие потоки в очереди
     executor.shutdown();
    // Ожидать пока все потоки будут завершены
    executor.awaitTermination();
    System.out.println("Finished all threads");  }}
```

Очень часто необходимо, чтобы поток после выполнения своей работы возвращал некоторое значение, в таких ситуациях можно использовать интерфейс Callable при создании класса. Он похож на Runnable, но имеет несколько отличий. В первую очередь после объявления данного интерфейса необходимо указать тип параметра, который должен вернуть поток. Вместо метода run() необходимо использовать метод call().

```
import java.util.concurrent.Callable;
public class CallableTask implements Callable<String>{
@Override
public String call() throws Exception {
String s="Callable Task Run at "+System.currentTimeMillis();
return s;}}
```

Класс Future используется с Callable и служит удобным средством для ожидания и возвращения результата задачи или отмены задачи перед ее выполнением. Future

вращается методами submit() ExecutorService. Future также является общим интерфейсом, который параметризуется возвращаемым типом. Интерфейс Future обладает блокирующим методом и методом ожидания get() . Если результата пока нет, то метод get() блокируется до тех пор, пока результат не появится. Метод isDone() — позволяет определить, завершены ли вычисления.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CallableClient {
public static void main(String[] args) {
Callable callableTask = new CallableTask();
//  Используем FixedThreadPool executor
ExecutorService executor = Executors.newFixedThreadPool(2);
Future<String> future = executor.submit(callableTask);
boolean listen = true;
while (listen) {
if (future.isDone()) {
String result;
try {
result = future.get();
listen = false;
System.out.println(result);
} catch (InterruptedException | ExecutionException e) {
e.printStackTrace();}   }   }
```

ГЛАВА 2.
ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

Многопоточный режим работы открывает новые возможности для программистов, однако эти возможности сопряжены с усложнением процесса проектирования приложения

и отладки. При написании параллельных или распределенных программ, как правило, необходимо “пройти” следующие три основных этапа.

1. Идентификация естественного параллелизма, который существует в контексте предметной области.

2. Разбиение задачи, стоящей перед программным обеспечением, на несколько подзадач, которые можно выполнять одновременно, чтобы достичь требуемого уровня параллелизма.

3. Координация этих задач, позволяющая обеспечить корректную и эффективную работу программных средств в соответствии с их назначением.

Эти три этапа достигаются при условии параллельного решения следующих проблем: “гонка” данных, обнаружение взаимоблокировки, бесконечное ожидание, отказ средств коммуникации, отсутствие средств централизованного распределения ресурсов.

К самым частым и сложным в обнаружении ошибкам многопоточных программ относится состояние “data race” и “race condition”. Первый означает наличие двух конфликтующих обращений к общему участку памяти из различных потоков; второй – влияние порядка выполнения операций в программе, определяемого извне, на корректность её работы. Оба термина описывают одну и ту же ситуацию – несинхронизированные конкурентные обращения к одним и тем же разделяемым данным из различных потоков. Таким образом, большинство опасностей, связанных с параллелизмом сводятся к своего рода *соперничеству за данные*. Соперничество за данные, или *условие соперничества*, происходит, когда множественные потоки или процессы читают или записывают разделяемый объект данных, и конечный результат зависит от порядка, в котором потоки работают. Листинг 1 показывает пример простого соперничества за данные, в котором программа может выдать либо 0, либо 1, в зависимости от планирования потоков.

Пример 1. Простое соперничество за данные

```
public class DataRace {
  static int a = 0;
  public static void main() {
    new MyThread().start();
    a = 1;  }
  public static class MyThread extends Thread {
    public void run() {
      System.out.println(a);    }  }  }
```

На рис. 2 представлен пример гонки в программе, написанной на языке с моделью разделяемой памяти. Из этого примера видно, что при отсутствии синхронизации между потоками возможны проблемные ситуации, когда потоки не «видят» изменений, сделанных друг другом. В данном случае первый поток увеличил значение переменной i, равное 5, на x, а второй – на y, но в итоге переменная i оказалась равной y+5, а не x+y+5. Таким образом, в программе возникло состояние гонки.

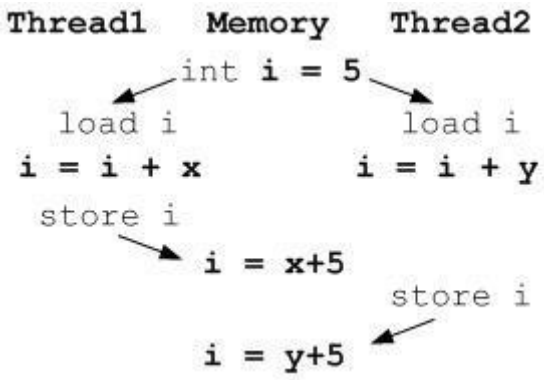


Рис.2. Пример гонки данных в программе.

Состояния гонки могут быть причинами серьезных проблем. Описанная выше ситуация может произойти с банковским счетом, который два разных потока пытаются одновременно увеличить на x и y рублей соответственно. При отсутствии должной синхронизации вместо итогового увеличения на x+y рублей может произойти увеличение лишь на x или на y. Если же речь идет о медицинских системах, то гонки могут привести еще к куда более серьезным последствиям – например, от передозировок, допущенных аппаратом лучевой терапии Therac-25 по причине гонок, скончались, как минимум, шесть человек [5]. Также состояние гонки послужило одной из причин аварии в энергосистеме США и Канады в 2003 году, в результате которой порядка 50 миллионов человек остались без электричества [6]. Эксплуатация состояния гонок используется для взлома программ. Атаки, базирующиеся на классе ошибок, позволяющих непривилегированной программе влиять на работу других программ через возможность изменения общедоступных ресурсов (обычно — временных файлов), в определённые моменты времени, в которые файл по ошибке программиста доступен для записи всем или части пользователей системы. Атакующая программа может разрушить содержимое файла, вызвав аварийное завершение программы-жертвы, или, подменив данные, заставить программу выполнить какое-либо действие на уровне своих привилегий. Именно по этой причине программное

обеспечение с серьёзными требованиями по безопасности, такое, как веб-браузер, использует случайные числа криптографического качества для именования временных файлов. В веб-приложениях класс уязвимостей «race condition» обычно встречается в функциях связанных с переводом денег/очков и т.п.

«Ручное» обнаружение гонок крайне затруднительно в силу следующих причин:

1. Конкретный порядок выполнения потоков специфичен для каждого запуска программы и зависит от порядка выполнения операций в потоках, определяемого планировщиком задач операционной системы. Поэтому состояния гонки трудновоспроизводимы даже при запусках одной и той же программы с одинаковыми входными данными.

2. Как правило, состояния гонки не локализованы во времени и, являясь причиной повреждения глобальных структур данных, не приводят к немедленным заметным ошибочным действиям программы – она будет продолжать выполняться, что впоследствии может привести к труднообъяснимым сбоям. Когда же ошибка в данных будет обнаружена, установить то место в программном коде, которое привело к этой ошибке, будет крайне проблематично.

3. Тестирование приложения (как самими разработчиками, так и тестировщиками) часто осуществляется на аппаратных ресурсах с достаточно малым количеством ядер, но промышленные сервера, на которых происходит эксплуатация приложения, могут содержать десятки или даже сотни ядер. В этом случае код различных потоков будет с гораздо большей вероятностью исполняться на физически различных ядрах, что приведет к проявлению многопоточных эффектов, которые остались незамеченными на этапе тестирования.

Единственным способом гарантировать отсутствие ошибок синхронизации является полное исключение доступа к механизмам возникновения таких ошибок на уровне языка программирования. По этому пути идут некоторые современные языки – Erlang, Jocaml и другие. Однако большинство используемых в индустрии языков, таких как C/C++, Java, C#, основываются на модели разделяемой памяти и не предоставляют подобной защиты.

2.1. МОДЕЛЬ ПАМЯТИ JAVA

Язык Java мультиплатформенный и программы на Java должны корректно выполняться на различных процессорах, под управлением различных операционных

систем. Все эти факторы требуют набора общих правил для корректного взаимодействия виртуальной машины с программной и аппаратной средой выполнения. Java поддерживает многопоточность на уровне языка. Модель памяти описывает отношения между переменными в программе (полями экземпляров, статическими полями и элементами массива) и деталями низкого уровня их хранения и восстановления из памяти в реальной вычислительной системе. Объекты хранятся в памяти, но компилятор, среда выполнения, процессор или кэш могут обращаться на свое усмотрение с установкой временных характеристик для перемещаемых значений в заданные для переменных ячейки памяти или из них. Например, компилятор может выбрать оптимизацию переменной счетчика цикла, сохранив ее в регистре, или же кэш может отложить сброс нового значения переменной в основную память на более подходящее время. Все эти оптимизации направлены на повышение производительности и, в общем случае, прозрачны для пользователя, но в мультипроцессорных системах эти сложности могут иногда проявиться.

Модель памяти Java - это набор правил, описывающий выполнение многопоточных программ. В отсутствии подобных правил в многопоточной программе может возникнуть, например, следующий вопрос: если один поток записал в переменную var1 значение A, всегда ли другой поток, читая значение той же переменной var1, получит значение A? Это может не произойти по ряду причин: значение переменной может сохраняться в регистрах, а не в основной памяти; значение переменной может сохраняться в локальном кэше процессора. Эти изменения не всегда доступны потокам, исполняющимся на других процессорах; при записи в основную память процессор может использовать буфер записи и запись в память из буфера может откладываться. Кроме этого оптимизирующий компилятор может переупорядочить операции, если это не меняет семантики программы. В этом случае без правильной синхронизации с точки зрения потока T1 действия, выполняемые потоком T2, могут происходить в порядке, не соответствующем порядку, записанном в программе, и в результате работы неправильно синхронизированной программы могут быть получены совершенно неожиданные результаты.

Надо отметить, что подобные проблемы могут возникнуть только в многопоточной программе и только если потоки работают с общими данными. Если поток работает с локальными данными, то в этом случае компилятор и процессор могут безопасно использовать весь набор оптимизаций — операции могут выполняться в любом порядке (если это не меняет логику программы).

Модель памяти Java базируется на двух базовых концепциях. Это отношения между блоками кода Synchronizes-With (синхронизируется с) и Happens-Before (происходит до).

Synchronizes-With — означает, что действие будет синхронизировать свое представление об объекте с представлением об объекте в основной памяти и лишь потом сможет продолжить работу.

Happens-Before — указывает, что один блок кода должен полностью завершиться, прежде чем выполнение другого блока кода сможет начаться. Отношение "happens-before" (предшествования) традиционно обозначается $\rightarrow_{hb}$:

- если событие A синхронизировано с событием B, то A $\rightarrow_{hb}$ B;
- если два действия в одном потоке A и B таковы, что B произошло после A, то A $\rightarrow_{hb}$ B;
- если A – последняя операция в конструкторе объекта, а B – первая в его финализаторе (finalizer), то A $\rightarrow_{hb}$ B;
- если A $\rightarrow_{hb}$ B и B $\rightarrow_{hb}$ C, то A $\rightarrow_{hb}$ C: отношение happens-before транзитивно замкнуто.

К важным понятиям модели памяти также относятся такие понятия как Atomicity, Visibility и Ordering (Атомарность, Видимость, Упорядоченность).

Часть проблем согласования потоков является следствием низкоуровневых операций. Это означает, что причины возникновения проблем лежат не на уровне семантики кода языка программирования, а на уровне компилятора, операционной системы или физического процессора. Классическим примером является *атомарность* операций, например, операции инкремента. Оператор инкремента существует во многих языках программирования, который воспринимается как атомарная операция (на уровне абстракции данного языка программирования). Но с точки зрения процессора это несколько операций. Прочитать данные из памяти, прибавить единицу к прочитанному значению, сохранить новое значение. Проблема заключается в том, что когда несколько потоков выполнения пытаются инкрементировать переменную, возможен такой момент, при котором несколько потоков одновременно прочитают одно и то же значение памяти, увеличат его на единицу и сохранят результат. Как следствие вычисления будут содержать ошибку. Действия, содержащие несколько подопераций, для которых нужно обеспечивать атомарность, будем называть составными (compound actions). Типичный пример составного действия прочитать-изменить-записать (read-modify-write) или проверить-затем-действовать (check-then-act). При выполнении этих действий может

возникнуть состояние гонки, когда корректность вычислений зависит от порядка выполнения потоков во времени. Например, как в случае с неатомарным инкрементированием итоговое значение зависит от того, как выполнялись потоки относительно друг друга. На загруженной системе можно получить корректный результат при небольшом количестве конкурирующих потоков выполняющих инкрементирование, но такой гарантии нет. Такая ситуация характерна для любых составных действий, которые не обеспечены должной атомарностью. Программы, иллюстрирующая эту ошибку, приведена ниже [7].

Пример1.

```
public class UnsafeCheckThenAct {
    private int number;
            public void changeNumber() {
        if (number == 0) {
            System.out.println(Thread.currentThread().getName() + " | Changed");
            number = -1;
        }
        else {
            System.out.println(Thread.currentThread().getName() + " | Not changed");
        }
    }
         public static void main(String[] args) {
        final UnsafeCheckThenAct checkAct = new UnsafeCheckThenAct();

        for (int i = 0; i < 50; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    checkAct.changeNumber();
                }
            }, "T" + i).start();              }      }      }
```

Пример2.

```
public class UnsafeReadModifyWrite {
      private int number;
      public void incrementNumber() {
            number++;          }
       public int getNumber() {
            return this.number;          }

       public static void main(String[] args) throws InterruptedException {
          final UnsafeReadModifyWrite rmw = new UnsafeReadModifyWrite();
            for (int i = 0; i < 1000; i++) {
             new Thread(new Runnable() {
               @Override
                public void run() {  rmw.incrementNumber();  } }, "T" + i).start();                }
                 Thread.sleep(6000);
          System.out.println("Final number (should be 1000): " + rmw.getNumber());     }
   }
```

Данный способ синхронизации по ресурсам используется при разработке класса, рассчитанного на взаимодействия в многопоточной среде. При этом методы, критичные к атомарности операций с данными (обычно - требующие согласованное изменение нескольких полей данных), объявляются как синхронизованные с помощью модификатора **synchronized.**

Проиллюстрируем на исправленных примерах:

Пример1.

```
public class SafeCheckThenAct {
      private int number;
public synchronized void changeNumber() {
          if (number == 0) {
             System.out.println(Thread.currentThread().getName() + " | Changed");
             number = -1;          }
```

```
            else {
                System.out.println(Thread.currentThread().getName() + " | Not changed");
            }
        }

        public static void main(String[] args) {
            final SafeCheckThenAct checkAct = new SafeCheckThenAct();

            for (int i = 0; i < 50; i++) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        checkAct.changeNumber();
                    }
                }, "T" + i).start();            }     }     }
```

Пример2.

```
public class SafeReadModifyWriteSynchronized {
    private int number;
            public synchronized void incrementNumber() {   number++;              }
            public synchronized int getNumber() {       return this.number;          }

            public static void main(String[] args) throws InterruptedException {
                    final SafeReadModifyWriteSynchronized rmw =
                    new SafeReadModifyWriteSynchronized();
                        for (int i = 0; i < 1000; i++) {
                new Thread(new Runnable() {
                  @Override
                   public void run() { rmw.incrementNumber(); } }, "T" + i).start();      }
                Thread.sleep(4000);
                System.out.println("Final number (should be 1000): " + rmw.getNumber());
            }     }
```

Если записать в переменную значение, то все последующие чтения из неё должны давать один и тот же результат, при условии, что между операциями чтения не будут производится дополнительные операции записи. Это утверждение верно только для однопоточного выполнения. В случае многопоточного выполнения, потоки в которых производятся операции чтения могут видеть разные значения, не смотря на то, что операция записи уже выполнена (в другом потоке). При этом не существует никаких гарантий, когда читающим потокам станет видно новое значение. Из этого вытекает *проблема видимости*. Наиболее распространённой причиной проявления проблемы видимости является наличия нескольких ядер/процессоров с собственными кэшами, на которых выполняется код. Допустим первый поток на первом ядре/процессоре произвёл вычисления и сохранил значение, которое не попадает сразу же в общую память, а задерживается в кэше до момента его сброса. При этом второй поток, выполняющийся на втором ядре/процессоре, читает значение переменной из общей памяти и видит старое значение. Большую часть времени ядра/процессоры выполняют работу с различными участками памяти и поэтому отсутствует взаимовлияние. Процессоры поддерживают механизмы управления способом взаимодействия с памятью при выполнении операции, но они реализованы в виде отдельных инструкций.

Причиной возникновения проблем с видимостью могут служить не только наличие кэшей, но и переупорядочивание операций. Большинство компиляторов языков и процессоры гарантируют только сохранение порядка выполнения операций, которые влияют на результат выполнения текущего потока. Для параллельных потоков такая гарантия отсутствует. То есть если в одном потоке выполняется код, который поочерёдно присваивает значения двум переменным $x1$ и $x2$, то второй поток может видеть старое значение $x1$ и новое $x2$. Это может произойти по причине изменения порядка операций присваивания с целью оптимизации (как на уровне компилятора, так и на уровне процессора).

```
public class NoVisibility {
   private static boolean ready;
      public static void main(String[] args) throws InterruptedException {
      new Thread(new Runnable() {
         @Override
         public void run() {
            while (true) {
```

```
                if (ready) {
                    System.out.println("Reader Thread - Flag change received.");
                    break;      }               }           }       }).start();

        Thread.sleep(3000);
        System.out.println("Writer thread - Changing flag...");
        ready = true;    }
}
```

Избежать эту ошибку можно, если объявить переменную как *volatile*. Это подходит только для контроля доступа к одиночному экземпляру или переменной примитивного типа: *int, boolean...* Когда переменная объявлена как *volatile*, любая запись её будет осуществляться прямо в память, минуя кеш. Также как и считываться будет прямо из памяти. Это значит, что все потоки будут "видеть" одно и то же значение переменной одновременно. Внесем исправления в предыдущий пример:

```
public class Visibility {
    private static volatile boolean ready;
    public static void main(String[] args) throws InterruptedException {
    new Thread(new Runnable() {
    public void run() {
    while (true) {
    if (ready) {
    System.out.println("Reader Thread - Flag change received. Finishing thread.");
    break; }).start();
    Thread.sleep(3000);
    System.out.println("Writer thread - Changing flag...");
    ready = true; } }
```

Когда к разделяемой переменной обращаются несколько потоков, все они должны использовать синхронизацию, в противном случае может возникнуть ситуация гонок данных. Главным средством синхронизации в языке программирования Java является ключевое слово *synchronized* (этот способ синхронизации известен также под названием

внутренняя блокировка (intrinsic locking)), которое обеспечивает взаимное исключение и гарантирует, что действия одного потока, выполняющего блок synchronized, видимы для остальных потоков, которые будут выполнять блок synchronized, защищенный этой же блокировкой. При правильном применении внутренняя блокировка позволяет сделатьпрограммы потокозащищенными, но она может быть тяжеловесной операцией при использовании для защиты маленьких участков кода, когда потоки часто соревнуются за блокировку. Модификатор volatile обеспечивает неблокирующую синхронизацию, не вызывает никаких блокировок и при ее использовании гарантированно нельзя попасть во взаимную блокировку.

При обеспечении безопасности потоков переменная volatile должна использоваться только для моделирования такой переменной, запись в которую не зависит от текущего (считываемого) состояния. В тех случаях, когда важно учитывать актуальное состояние, всегда нужно применять блокировку для обеспечения полной безопасности.

Существует целая группа классов пакета java.util.concurrent.atomic,обеспечивающая неблокирующую синхронизацию. Атомарные классы созданы для организации неблокирующих структур данных. Классы атомарных переменных AtomicInteger, AtomicLong, AtomicReference и др. расширяют нотацию volatile значений, полей и элементов массивов. Все атомарные классы являются изменяемыми в отличие от соответствующих им классов-оболочек. При реализации классов пакета использовались эффективные атомарные инструкции машинного уровня, которые доступны на современных процессорах. В некоторых ситуациях могут применяться варианты внутреннего блокирования. Экземпляры классов, например, AtomicInteger и AtomicReference,предоставляют доступ и разного рода обновления к одной-единственной переменной соответствующего типа. Каждый класс также обеспечивает набор методов для этого типа. В частности, класс AtomicInteger — атомарные методы инкремента и декремента. Инструкции при доступе и обновлении атомарных переменных, в общем, следуют правилам для volatile.

Пример приведенного ниже счетчика Counter является потокозащищенным, но необходимость использования блокировок снижают производительность программы. Однако блокировка необходима, поскольку, как было сказано выше, инкрементирование, хотя и выглядит одной операцией, является сокращенной записью трех отдельных операций: извлечение значения, добавление единицы и запись значения. Синхронизация необходима также в методе getValue для гарантии того, что вызывающие этот метод потоки видят самое новое значение.

При одновременном запросе несколькими потоками одной и той же блокировки один выигрывает и захватывает блокировку, другие ожидают своей очереди. JVM обычно реализуют блокировку путем остановки блокируемого потока и последующего его продолжения. Получаемое переключение контекста может вызвать существенные задержки в сравнении с небольшим количеством команд, защищенных блокировкой.

```
public final class Counter {
   private long value = 0;
   public synchronized long getValue() {
      return value;    }

   public synchronized long increment() {
      return ++value;    } }
```

Класс NonblockingCounter демонстрирует один из простейших неблокирующих алгоритмов: счетчик AtomicInteger, использующий метод compareAndSet(). Метод compareAndSet() означает "Обновить переменную этим новым значением, но отказать, если другой поток изменил значение после моего последнего просмотра".

```
public class NonblockingCounter {
   private AtomicInteger value;
   public int getValue() {
      return value.get();    }
   public int increment() {
      int v;
      do {
         v = value.get();
     while (!value.compareAndSet(v, v + 1));
      return v + 1;    }
}
```

Классы атомарных переменных называются *атомарными*, потому что они предоставляют мелкомодульные атомарные обновления чисел и объектных ссылок, но они также атомарны и в том смысле, что являются основными строительными блоками для неблокирующих алгоритмов. Неблокирующие алгоритмы являются предметом

изучения уже более 20 лет, но стали возможными в языке программирования Java только с появлением Java 5.0.

Неблокирующая версия имеет некоторые преимущества в производительности по отношению к версии, основанной на блокировках. Она выполняет синхронизацию на более низком уровне модульности (конкретное место в памяти), используя аппаратный примитив вместо фрагмента кода блокировки JVM. Проигравшие потоки могут выполнить повторную попытку немедленно, без остановки и продолжения. Более мелкая модульность уменьшает шанс возникновения состязания, а возможность повторения попытки без остановки и продолжения уменьшает стоимость состязания.

Неблокирующие алгоритмы часто называют *оптимистическими*, потому что они выполняются с предположением о том, что не будет конфликтных ситуаций. Если такая ситуация возникает, они возвращаются на шаг назад и повторяют действие. В случае со счетчиком гипотетическим шагом является операция инкрементирования – она извлекает и добавляет единицу к старому значению в надежде на то, что оно не изменится за время вычисления обновленного значения. Если это не так, она должна повторно извлечь значение и снова выполнить инкрементирование.

2.2. БЛОКИРУЮЩАЯ СИНХРОНИЗАЦИЯ

Синхронизация представляет собой координацию параллельно выполняемых действий с целью получения предсказуемых результатов. Синхронизация особенно важна, когда множество потоков получает доступ к одним и тем же данным.

Для борьбы с гонками за данными обычно рекомендуется использовать локальные по отношению к потоку, а не разделяемые переменные. Управление доступом к разделяемым переменным осуществляется с помощью различных средств синхронизации (они могут быть реализованы с помощью семафоров, критических секций, взаимных блокировок - мьютексов). Когда двум или более потокам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), необходимо чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из потоков. Таким образом, главное требование к механизмам разделения ресурсов является гарантированное обеспечение использования каждого разделяемого ресурса только одним потоком от момента выделения ресурса этому потоку до момента освобождения ресурса. Данное требование в литературе обычно именуется взаимоисключением потоков (*mutual exclusion*). Командные последовательности потоков,

в ходе которых поток использует ресурс на условиях взаимоисключения, называется *критической секцией потока*. С использованием последнего понятия условие взаимоисключения потоков может быть сформулировано как требование нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного потока. При этом важно, чтобы ожидающий поток или потоки ожидали, не используя время центрального процессора на опрос для постоянной проверки некоторых условий.

Как было сказано выше, критическая секция (*Critical Section*) это участок кода, в котором поток (thread) получает доступ к ресурсу (например, переменной), который доступен из других потоков. Критическая секция – участок кода в многопоточной программе, выполняемый всеми потоками последовательно. Для создания критической секции в Java используется подход, реализованный на основе synchronized методов и блоков. Поток, вошедший в synchronized метод, может в рамках этого метода работать с данными, разделяемыми с другими потоками, так как ни один другой поток не сможет в это же время войти в этот метод, а говоря точнее, другие потоки, попытавшиеся сделать это, будут заблокированы до тех пор, пока первый поток не покинет synchronized метод. Таким образом, объявление блока кода синхронным (synchronized) имеет два важных последствия, такие как atomicity (атомарность) и visibility (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 обычно связана с особенностями кэширования памяти и оптимизацией программы в процессе компилирования. Обычно потоки могут свободно кэшировать значения переменных (в регистрах, в процессорных кэшах, или через переупорядочение команд или другую оптимизацию программы в процессе компилирования). Новые значения переменных не обязательно сразу же будут видны другим потокам. Но если при разработке программы использовалась синхронизация, то во время выполнения программы будет проверяться, чтобы обновления переменных, выполненные одним потоком до выхода из блока synchronized, были бы видны другому потоку, когда он будет входить в блок synchronized, защищенный тем же монитором (блокировкой).

В Java все объекты имеют одну блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта. Если поток получил блокировку, ни один другой поток не может войти в синхронизированный

код, пока блокировка не будет снята. Когда поток, владеющий блокировкой, выходит из синхронизированного кода, блокировка снимается. Теперь другой поток может получить блокировку объекта и выполнить синхронизированный код. Если поток пытается получить блокировку объекта, когда другой поток владеет блокировкой, поток переходит в состояние блокировки до тех пор, пока блокировка не снимется. Когда поток заблокирован или находится в состоянии ожидания, он временно не активен. Он не выполняет никакого кода и потребляет минимум ресурсов. На планировщике потоков лежит обязанность повторно активизировать его.

Рассмотрим ситуацию, в которой несколько потоков пытаются получить доступ к общему ресурсу, например, доступ к переменной класса, метода или общему файлу. Допустим, есть класс, функциональность которого состоит в простой печати диапазона номеров, используя несколько потоков.

Пример1. Класс без синхронизации

```
class Range {
    void display(int low, int high) {
    System.out.println("Display between " + low + " and " + high);
    for(int i = low; i <= high; i++) {
      System.out.print(i + " ");
    }
    System.out.println();   }
}
```

```
/* Вспомогательный класс для печати диапазона */
class Helper extends Thread {
  Range r;
  int low, high;
  Helper(Range r_obj, int low_limit, int high_limit) {
    r = r_obj;
    low = low_limit; high = high_limit;
  }
  public void run() {
    r.display(low, high);   }
}

public class SyncProblemDemo {
  public static void main(String args[]) {
    /* создаем один объект для печати различных диапазонов */
    Range obj = new Range();
    /* создаем поток для печати диапазона [1, 10] */
    Helper th1 = new Helper(obj, 1, 10);
    /* создаем другой поток для печати диапазона [51, 60] */
    Helper th2 = new Helper(obj, 51, 60);
    th1.start();
    th2.start();  }     }
```

Работа программы может выглядеть следующим образом

```
Display between 1 and 10
Display between 51 and 60
1 51 2 52 3 53 4 54 5 55 6 56 7 57 8 58 9 59 60 10
```

Пример 2. Класс с синхронизацией

```
class Range {
  /* Метод display() – критическая секция
     и только один поток может находиться внутри сихронизированного метода.
     Другой поток может выполнять этот метод только тогда, когда первый поток
     закончит выполнение своей задачи с помощью этого метода   */
  synchronized void display(int low, int high) {
```

```
    System.out.println("Display between " + low + " and " + high);
    for(int i = low; i <= high; i++) {
      System.out.print(i + " ");
    }
    System.out.println();   }
}

/* Вспомогательный класс для печати диапазона */
class Helper extends Thread {
  Range r;
  int low, high;
  Helper(Range r_obj, int low_limit, int high_limit) {
    r = r_obj;
    low = low_limit; high = high_limit;
  }
  public void run() {
    r.display(low, high);   }
}
public class SynchronizationDemo {
  public static void main(String args[]) {
    Range obj = new Range();
    /* создаем поток для печати диапазона [1, 10] */
    Helper th1 = new Helper(obj, 1, 10);
    /* создаем другой поток для печати диапазона [51, 60] */
    Helper th2 = new Helper(obj, 51, 60);
    th1.start();
    th2.start();   }
}
```

Вывод:

```
Display between 1 and 10
1 2 3 4 5 6 7 8 9 10
Display between 51 and 60
51 52 53 54 55 56 57 58 59 60
```

В большинстве случаев блоки синхронизации предпочтительней синхронных методов по скорости, так как ими можно ограничить только критичный участок кода (где происходит чтение/запись объекта совместного пользования).

```
public class SynhroDemo1 {
   // объект мониторинга,  должен быть final
   public final Object obj = new Object();
   int i=0;

   // пример синхронного метода
   public synchronized void method1(){
      // ...
      ++i;
      String s =obj.toString()+i;
     // ...
   }

   public void method2(){
      // ...
      // пример блока синхронизации  с монитором obj
      synchronized (obj) {
         ++i;
         String s =obj.toString()+i;
      }
     // ...
   }          }
```

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции **synchronized**, и он становится недоступным для других синхронизированных методов и блоков.

Следующий пример показывает, как два потока входят в объект, когда методы этого объекта синхронизированы различными блокировками:

```
import static net.mindview.util.Print.*;
 class DualSynch {
  private Object syncObject = new Object();
  public synchronized void f() {
    for(int i = 0; i < 5; i++) {
      print("f()");
      Thread.yield();    }  }
  public void g() {
    synchronized(syncObject) {
      for(int i = 0; i < 5; i++) {
        print("g()");        Thread.yield();      }    }
  } }
public class SyncObject {
  public static void main(String[] args) {
    final DualSynch ds = new DualSynch();
    new Thread() {
      public void run() {        ds.f();      }    }.start();
    ds.g();  }}
```

Метод f() класса DualSync синхронизируется по объекту this (синхронизируя метод целиком), а метод g() использует синхронизацию посредством объекта syncObject. Таким образом, два варианта синхронизации независимы. Демонстрируется этот факт методом main(), в котором создается поток Threadc вызовом метода f(). Поток main() после этого вызывает метод g(). Из результата работы программы видно, что оба метода работают одновременно и ни один из них не блокируется соседом.

2.3. МОНИТОР, СЕМАФОР

Многопоточное приложение создает некоторое количество потоков (пул потоков), передавая каждому из них задачу и данные для обработки. Задачи выполняются параллельно. Если потоки не имеют общих данных, то программа не имеет накладных расходов на синхронизацию, что делает работу программы достаточно быстрой. При организации взаимодействия между потоками необходимо исключить конфликты между ними. Например, должен быть способ предотвратить запись данных в одном потоке, когда другой занимается их чтением. Для этой цели в Java реализован *монитор*. Монитор — это

управляющий механизм, впервые реализованный Ч.Хоаром [8]. Монитор можно интерпретировать как ящик, который принимает только один поток в единицу времени. Как только поток вошел в монитор, все другие потоки должны ждать, пока тот не покинет его. Таким образом, монитор может быть использован для защиты разделяемых ресурсов от одновременного использования более чем одним потоком.

Поддержка синхронизации встроена в язык Java. Каждый экземпляр любого класса имеет монитор. Под монитором понимается некая управляющая конструкция, обеспечивающая монопольный доступ к объекту. Методы wait(), notify(), notifyAll() корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Для того чтобы войти в монитор, необходимо вызвать метод этого объекта, отмеченный ключевым словом synchronized. Для того чтобы выйти из монитора и, тем самым, передать управление объектом другому потоку, владелец монитора должен всего лишь вернуться из синхронизованного метода. Если во время выполнения синхронизованного метода объекта другой поток попробует обратиться к методам или данным этого объекта, он будет блокирован до тех пор, пока не закончится выполнение синхронизованного метода. При запуске синхронизованного метода говорят, что объект входит в монитор, при завершении – что объект выходит из монитора. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора. Пока один поток меняет данные, второй не должен иметь права их читать или менять. Такой тип синхронизации называется синхронизацией по ресурсам и обеспечивает блокировку данных на то время, которое необходимо потоку для выполнения тех или иных действий. Данный способ синхронизации по ресурсам используется при разработке класса, рассчитанного на взаимодействия в многопоточной среде.

Монитор объекта захватывается с помощью вызова синхронного метода либо в синхронном блоке:

```
class MyClass {
  public synchronized void doIt() {
    …
  }
}
…
MyClass obj = new MyClass();
obj.doIt(); //Захват монитора obj
```

```
class AnotherClass {
  public void doIt() {…}
}
…
AnotherClass obj = new AnotherClass();
//Захват монитора obj
synchronized (obj) {
  obj.doIt(); }
```

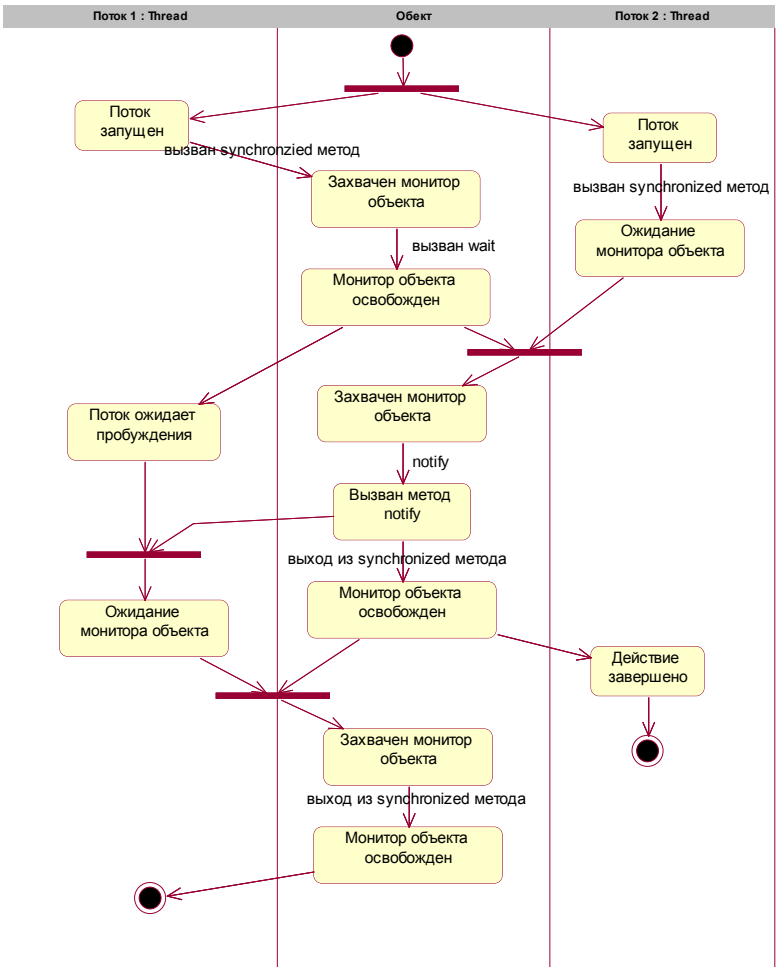
Монитор объекта освобождается автоматически по завершению выполнения синхронизированного метода или блока

Синхронизация имеет несколько аспектов. Одним из них является взаимное исключение (***mutual exclusion***) — только один поток может владеть монитором, таким образом синхронизации на мониторе означает, что как только один поток входит в synchronized-блок, защищенный монитором, никакой другой поток не может войти в блок, защищенный этом монитором пока первый поток не выйдет из synchronized-блока.

Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции:

- monitorenter: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.
- monitorexit: освобождение монитора
- wait: перемещение текущего потока в так называемый wait set монитора и ожидание того, как произойдёт notify. Выход из метода wait может оказаться и ложным. После того, как поток, владеющий монитором, сделал wait, монитором может завладеть любой другой поток.
- notify(all): пробуждается один (или все) потоки, которые сейчас находятся в wait set монитора. Чтобы получить управление, пробуждённый поток должен успешно захватить монитор (monitorenter)
- Монитор объекта освобождается автоматически по завершению выполнения синхронизированного метода или блока
- Монитор может быть освобожден добровольно с помощью вызова на объекте методов wait(), wait(…). При этом вызвавший метод поток блокируется.
- Для того чтобы воспользоваться семейством методов wait нужно захватить монитор объекта
- Методы notify() и notifyAll() позволяют возобновить исполнение потока(ов) заблокированного(ых) на мониторе объекта с помощью вызова метода из семейства wait()
- Для того чтобы вызвать notify или notifyAll нужно захватить монитор объекта

- Вызов метода notify() приведет к пробуждению первого попавшегося блокированного на объекте потока по завершению синхронизированного метода или блока в котором вызван notify()



Если поток захватил монитор и вызвал wait(), чтобы дождаться некоторого состояния, проверка наступления этого события должна быть завернута в цикл вида (guarded block):

```
public synchronized guardedJoy() {
while(!joy) {
    try {
        wait();
    } catch (InterruptedException e) {}
}
System.out.println("Joy and efficiency have been achieved!");}
```

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Такой прием, как было сказано выше, называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то только взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

Чтобы создать надёжную и стабильную систему, должны быть ограничения на ресурсы (базы данных, файловую систему, сокеты и т.д.). Для создания ограничений на использование дорогих ресурсов, таких как подключения к базе данных (в этом случае можно использовать пул подключений), могут использоваться *семафоры* (java.util.concurrent.Semaphore). Способ синхронизации процессов, ставший классическим и получившим название семафор, был предложен Э. Дейкстрой [9].

Семафор — это механизм синхронизации, который можно использовать для управления отношениями между параллельно выполняющимися программными компонентами и реализации стратегий доступа к данным. Семафор — это переменная специального вида, которая может быть доступна только для выполнения узкого диапазона операций. Семафор используется для синхронизации доступа процессов и потоков к разделяемой модифицируемой памяти или для управления доступом к устройствам или другим ресурсам. Семафор можно рассматривать как ключ к ресурсам. Этим ключом может владеть в любой момент времени только один процесс или поток. Какая бы задача ни владела этим ключом, он надежно запирает (блокирует) нужные ей ресурсы для ее монопольного использования. Блокирование ресурсов заставляет другие задачи, которые желают воспользоваться этими ресурсами, ожидать до тех пор, пока они не будут разблокированы и снова станут доступными. После разблокирования ресурсов следующая задача, ожидающая семафор, получает его и доступ к ресурсам. Какая задача будет следующей, определяется стратегией планирования, действующей для данного потока или процесса.

Семафоры помогут создать ограничения и заблокируют потоки в случае недоступности ресурса.

Предположим, что есть база данных служащих. Она содержит информацию об адресах и номерах телефонов служащих. Эти значения иногда изменяются, но гораздо чаще они считываются. Нужно защитить данные от искажений, при этом программа должна быть настолько эффективна, насколько это возможно. База данных может иметь столько читателей, сколько требуется, но когда кто-то захочет записать новую

информацию в файл, он должен получить исключительную блокировку. Это выполняется с помощью семафора (*semaphore*). Данный объект синхронизации позволяет ограничить доступ потоков к объекту синхронизации на основании их количества. Например, в программе к какому-нибудь объекту должны иметь возможность обратиться максимум три потока. Тогда сначала семафор инициализируется и ему передается количество потоков, которые к нему могут обратиться. Дальше при каждом обращении к ресурсу его счетчик уменьшается. Когда счетчик уменьшиться до 0, к ресурсу обратиться больше нельзя. При отсоединении потока от семафора его счетчик увеличивается, что позволяет другим потокам обратиться к нему. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen — проверять) и V (от verhogen — увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S = = 0 процесс блокируется;

S = S - 1;

V (S) : S = S + 1 ;

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0 , то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Виды семафоров:

Двоичный семафор

- S может принимать значения 0 и 1, инициализируется значением 1
- обеспечивает эксклюзивный доступ к ресурсу (например, при работе в критической секции)
- одновременно может выполняться только один поток

Счетный семафор

- S инициализируется значением N (число доступных единиц ресурса)
- представляет ресурсы, состоящие из нескольких однородных элементов

- позволяет потокам исполняться, пока есть неиспользуемые элементы

Семафор всегда устанавливается на предельное положительное число потоков, одновременное функционирование которых может быть разрешено. При превышении предельного числа все желающие работать потоки будут приостановлены до освобождения семафора одним из работающих по его разрешению потоков. Уменьшение счетчика доступа производится методами void acquire() и его оболочки boolean tryAcquire(). Оба метода занимают семафор, если он свободен. Если же семафор занят, то метод tryAcquire() возвращает ложь и пропускает поток дальше, что позволяет при необходимости отказаться от дальнейшей работы потоку, который не смог получить семафор. Метод acquire() при невозможности захвата семафора остановит поток до тех пор, пока хотя бы другой поток не освободит семафор. Метод boolean tryAcquire(long timeout,TimeUnit unit) возвращает ложь, если время ожидания превышено, т. е. за указанное время поток не получил от семафора разрешение работать и пропускает поток дальше. Метод release() освобождает семафор и увеличивает счетчик на единицу. Стандартное взаимодействие методов acquire() и release() демонстрирует следующий фрагмент:

```
public void run() {
try {
semaphore.acquire();
// код использования защищаемого ресурса
} catch (InterruptedException e) {
} finally {
semaphore.release(); // освобождение семафора
}  }
```

Проиллюстрируем работу семафору на классической задаче «производитель-потребитель» («producer-consumer»). Producer (изготовитель) - это некоторый поток, который генерирует “задания” и складывает их в очередь Queue. Consumer (исполнитель) - это поток, который берет задания из очереди, выполняет и отправляет результаты туда, куда нужно. Очередь Queue - это ограниченный буфер заданий с заранее заданной вместимостью [10].

```
import java.util.concurrent.Semaphore;
class Queue {
  int value;
```

```
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);
  void get() {
    try {
      semCon.acquire();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }
    System.out.println("Got: " + value);
    semProd.release();
  }
  void put(int n) {
    try {
      semProd.acquire();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }
    this.value = n;
    System.out.println("Put: " + n);
    semCon.release();
  }
}
class Producer implements Runnable {
  Queue q;
  Producer(Queue q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }
  public void run() {
    for (int i = 0; i < 20; i++)
      q.put(i);
  }
}
class Consumer implements Runnable {
```

```
  Queue q;
  Consumer(Queue q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run() {
    for (int i = 0; i < 20; i++)
      q.get();
  }
}
public class ProdCon {
  public static void main(String args[]) {
    Queue q = new Queue();
    new Consumer(q);
    new Producer(q);
  }  }
```

Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят).

Операции над мьютексами

- acquire(mutex) – уменьшить (занять) мьютекс
- release(mutex) – увеличить (освободить) мьютекс
- tryacquire(mutex) – часто реализуемая неблокирующая операция, выполняющая попытку уменьшить (занять) мьютекс

Приведенный ниже пример реализует простой мьютекс, а метод acquire() также является атомарной операцией читать-модифицировать-записать. Для получения этого мьютекса надо убедиться, никто в данное время не блокирует его. (curOwner == null), а потом сделать запись о том, что мьютекс забирается (curOwner = Thread.currentThread()), все это исключит возможность появления в середине работы другого потока и модификации поля curOwner field. Таким образом, мьютекс всегда приналежит одному единственному потоку. Этот поток его получает и он же обязан его вернуть. Семафор же, напротив, никому не принадлежит и может получить сигнал из любого потока (процесса).

```
public class SynchronizedMutex {
    private Thread curOwner = null;

    public synchronized void acquire() throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        while (curOwner != null)
            wait();
        curOwner = Thread.currentThread();
    }
    public synchronized void release() {
        if (curOwner == Thread.currentThread()) {
            curOwner = null;
            notify();
        } else
            throw new IllegalStateException("not owner of mutex");
    }
}
```

У традиционного подхода, использующего критические секции и блокирование потоков, есть несколько недостатков, и, возможно, главным из них является производительность. Если блокировка не была захвачена другим потоком, то операция захвата этой блокировки в современных виртуальных машинах выполняется очень быстро. Однако если блокировка захвачена другим потоком, то в этом случае должен быть задействован сравнительно дорогой механизм блокирования и последующего разблокирования потока. Также очень часто критические секции выполняют очень небольшой объём действий (например, просто увеличивают значение счётчика), при этом время, затрачиваемое на блокирование/разблокирование потока, оказывается гораздо больше по сравнению со временем работы критической секции, и в таких случаях было бы хорошо иметь более легковесный способ синхронизации. Помимо этого, при использовании синхронизации, основанной на блокировании, часто возникают такие проблемы, как взаимоблокировки (deadlocks) и инверсия приоритетов (priority inversion). Инверсия приоритетов возникает в том случае, если высокоприоритетный поток приостанавливается из-за того, что он не может получить блокировку, захваченную низкоприоритетным потоком, в этом случае высокоприоритетный поток не может

продолжить выполнение до тех пор, пока блокировка не освобождена, то есть на время ожидания его приоритет как бы понижается до приоритета потока, удерживающего блокировку. Низкоприоритетному потоку, наоборот, искусственно и временно дается больший приоритет, просто потому, что он удерживает блокировку. Эта ситуация со временем разрешится сама собой, когда поток низкоприоритетный освободит блокировку. Ситуация может осложниться, если появится поток со средним приоритетом. Хотя этот поток может и не нуждаться в блокировке, его простое присутствие может полностью исключить выполнение низкоприоритетного потока, что непрямым образом скажется на выполнении высокоприоритетного потока.

Инверсия приоритетов особенно опасна в real time системах. Способами предотвращения подобной ситуации могут быть: отказ от прерываний на время выполнения критичного высокоприоритетного кода; временный подьем приоритета до максимального у каждой задачи которая захватила ресурс, чтобы предотвратить задвигание высокоприоритетных заблокированных на ресурсе задач в очереди ожидания низкоприоритетными и незаблокированными.

2.4. КЛАСС ReentrantLock

Пакет java.util.concurrent содержит классы, которые эмулируют и расширяют встроенные средства синхронизации, обеспечивают их более гибкое использование за счет некоторой встроенной безопасности синхронизации, добавляют блокировки для чтения/записи. Классы, входящие в java.util.concurrent, обеспечивают большую гибкость при разработке программ в отличие от старых методов синхронизации. Классы можно рассматривать как обособленный механизм, предлагающий альтернативный способ решения тех же проблем синхронизации. При этом они обеспечивают такую же семантику операций с памятью, что и мониторы. Базовые операции с блокировками описывает интерфейс java.util.concurrent.locks.Lock. В отличие от мониторов функциональность блокировок из стандартной библиотеки помимо методов безусловного захвата блокировки, содержит также методы, обеспечивающие условные попытки захвата, в то числе и с указанием таймаута.

Основу пакета locks составляют интерфейсы Lock и Condition. Lock представляет ту же идею, что и блокировка Java (блок контроля), которая связана с каждым объектом и классом для использования с синхронизированными методами и блокировками. Класс Lock предоставляет эксклюзивный доступ владельца блокировки, позволяя только одной

стороне иметь блокировку в определенный момент времени посредством методов lock () и unlock (). В синхронизации языка Java это выполняется косвенно при помощи ключевого слова synchronized:

```
// синхронизированный метод synchronized void writeDataf) { ... }
// синхронизированный блок synchronized ( someObject ) { … }
```

При входе в синхронизированный метод или блок Java приобретает блокировку и автоматически отпускает ее при выходе. Даже если выбрасывается исключение или поток погибает неожиданно, происходит автоматическое освобождение от всех блокировок. Использование класса Lock позволяет явно использовать блокировку, когда нужен ресурс и его разблокирование. Типичной последовательностью использования является: получить блокировку, обратиться к защищенному ресурсу, снять блокировку:

```
lock.lock();
try {
   //Синхронизируемые действия
}
finally {
   lock.unlock();
},
```

где lock это объект класса реализующего интерфейс Lock. Первый оператор вызова lock() получает блокировку. Последующие вызовы другими потоками блокируются, пока блокировка не будет отпущена.

Реализация блокировки в приведенном ниже примере осуществляется с помощью класса ReentrantLock, который представляет реентрантную блокировку. Reentrancy - свойство функции или программы, предполагающее её корректный повторный вызов во время исполнения (например, рекурсивный). Реентрантный (reentrant) означает, что текущий владелец (поток) блокировки не будет блокироваться при многократной попытки захвата, наоборот, такая попытка всегда будет удачной. При этом необходимо на каждую операцию захвата выполнять операции освобождения. Монитор также является примером реентрантной блокировки.

```
java.util.concurrent.locks.ReentrantLock;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ReentrantLockHowto {
```

```
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

     public int getCount() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + " gets Count: " + count);
            return count++;
        } finally {
            lock.unlock();        }
     }
     public synchronized int getCountTwo() {
            return count++;     }
    public static void main(String args[]) {
        final ThreadTest counter = new ThreadTest();
        Thread t1 = new Thread() {
            public void run() {
                while (counter.getCount() &lt; 6) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ex) {
                        ex.printStackTrace();         }
}           }        };

        Thread t2 = new Thread() {
            public void run() {
                while (counter.getCount() &lt; 6) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ex) {
                        ex.printStackTrace();      }
      }           }        };

        t1.start();
```

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

Output:

```
Thread-0 gets Count: 0
Thread-1 gets Count: 1
Thread-1 gets Count: 2
Thread-0 gets Count: 3
Thread-1 gets Count: 4
Thread-0 gets Count: 5
Thread-0 gets Count: 6
Thread-1 gets Count: 7
```

Монитор и ReentrantLock обеспечивают одноранговый доступ к разделяемым данным. Все потоки, пытающиеся войти в защищаемую область, равны. Внутри защищаемой области может находиться только один поток. Такое поведение не всегда эффективно. Допустим, у нас есть несколько потоков считывания/изменения данных и несколько чтения, при этом код содержит много считывающих и сравнительно мало записывающих операций. В случае использования общей блокировки все конкурирующие потоки будут выполняться последовательно на участке доступа к разделяемой памяти. Но потоки чтения вполне могут иметь одновременный доступ к данным, что позволит им выполняться параллельно. При этом поток, изменяющий данные, должен быть единственным владельцем защищаемого участка памяти. Такое поведение поддерживается классом ReentrantReadWriteLock реализующим интерфейс ReadWriteLock. Для пользователя объект данного класс выглядит как контейнер с двумя методами, возвращающими ссылки на две блокировки чтения и записи. Использование этих блокировок осуществляется в соответствие с контрактом Lock. Блокировка записи ведёт себя аналогично ReentrantLock, а блокировка чтения позволяет нескольким потокам становиться её владельцем.

Интерфейс Condition предоставляет функционал, аналогичный wait(), notify(), notifyAll(), с помощью методов await(), signal(), signalAll().Condition ассоциируется с Lock при помощи блокировки newCondition(). В отличие от обычной блокировки Java, Lock может иметь множество объектов интерфейса Condition.

Метод await() класса Condition используется как метод wait() объекта Java в блоке synchronized:

```
Lock lock = . . .

Condition condition=lock.newCondition();

Lock.lock();

condition.await();// блок, ожидающий signal();

lock.unlock();

// в другом потоке

lock.lock();

condition.signal();

lock.unlock();
```

Как и метод wait(), метод await() класса Condition может вызываться только когда поток является владельцем блокировки, связанной условием, и метод signal() может быть вызван только другим потоком, который приобрел блокировку.

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

final Condition hasMessage = l.newCondition();

long msgCount = 0;

...

public void doSome() {

l.lock();

try{

 while(msgCount == 0){
```

```
 hasMessage.await();  }
 //do something with msgs
} finally {
 l.unlock();
}   }
public void addMessage(Msg msg){
l.lock();
try{
 //add msg
hasMessage.signal();
} finally {
 l.unlock();  }    }
```

Объект Lock действует как фабричный объект (factory object) для переменных условия, связанных с этой блокировкой, и в отличие от стандартных методов wait и notify, здесь может быть более одной переменной условия, связанной с данным Lock. Это облегчает разработку многих параллельных алгоритмов.

```
final Condition notFull = l.newCondition();
final Condition notEmpty = l.newCondition();
```

Соответственно методы await()/signal() будут работать относительно конкретного Condition и потоки будут пробуждаться не все, а только необходимые, которые ждут на конкретном условии.

```
import java.util.concurrent.locks.*;

 public class BlockingQueue {

      private final static int SIZE = 10;

      private Object[] buf = new Object[SIZE];

      private int first = 0, last = 0;

      private Lock lock = new ReentrantLock();

     private Condition notFull = lock.newCondition();

     private Condition notEmpty = lock.newCondition();

       public Object dequeue() throws InterruptedException {

         lock.lock();

             try {

             while (first == last)

                notEmpty.await();

             Object e = buf[last];

             last = (last + 1) % SIZE;

             System.out.println("dequeue() returning, value = " + e);

             notFull.signal();

             return e;

          } finally {

             lock.unlock();          }      }
```

```
public void enqueue(Object c) throws InterruptedException {
 lock.lock();
 try {
    while ((first + 1) % SIZE == last)
       notFull.await();
    buf[first] = c;
    first = (first + 1) % SIZE;
    System.out.println("enqueue(" + c + ") returning");
    notEmpty.signal();
 } finally {
    lock.unlock();         }      }  }
```

2.5. ПОТОКОБЕЗОПАСНЫЙ КЛАСС И ШАБЛОН (ПАТТЕРН) ПРОЕКТИРОВАНИЯ Singeleton

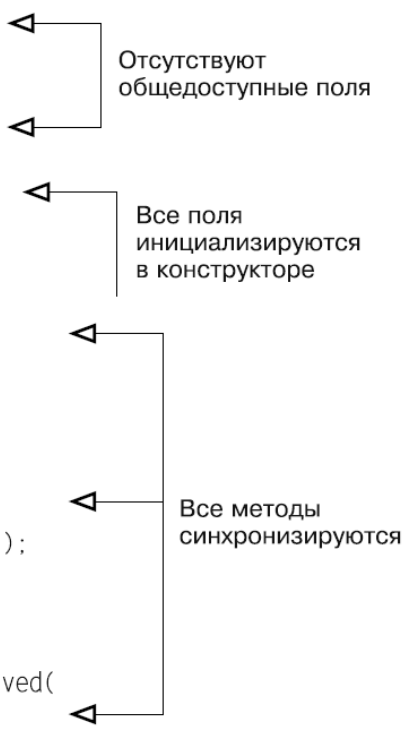
Потоковая безопасность (англ. thread-safety) — это концепция программирования, применимая к многопоточным программам. Ненамеренные ошибки конкурентного доступа, такие как повреждение данных или нарушение логики одного или нескольких потоков, называются нарушениями безопасности потоков (thread safety violations). Их сложно воспроизводить, сложно находить и сложно тестировать.

Полностью потокобезопасный синхронизированный класс — это класс, удовлетворяющий следующим условиям [10]:

- все поля всегда инициализируются в согласованное состояние в каждом конструкторе;

- отсутствуют общедоступные поля;
- экземпляры объектов гарантированно являются согласованными после возврата из любого незакрытого (non-private) метода (при условии, что на момент вызова метода состояние было согласованным);
- все методы доказуемо завершаются в ограниченное время;
- все методы синхронизированы;
- при нахождении в несогласованном состоянии не выполняются вызовы к методам другого экземпляра;
- при нахождении в несогласованном состоянии не выполняются вызовы к какому либо незакрытому методу.

```
public class ExampleTimingNode implements SimpleMicroBlogNode {

  private final String identifier;

  private final Map<Update, Long> arrivalTime
  ➥ = new HashMap<>();

  public ExampleTimingNode(String identifier_) {
    identifier = identifier_;
  }

  public synchronized String getIdentifier() {
    return identifier;
  }

  public synchronized void propagateUpdate(
  ➥ Update update_) {
    long currentTime = System.currentTimeMillis();
    arrivalTime.put(update_, currentTime);
  }

  public synchronized boolean confirmUpdateReceived(
  ➥ Update update_) {
    Long timeRecvd = arrivalTime.get(update_);
    return timeRecvd != null;
  }
}
```



Предположим, нужно написать класс, который будет управлять пулами потоков и соединений к базе данных, реестром, конфигурациями и т.д. В приложении может быть только один менеджер работы с базой данных, а создание еще одного не только нецелесообразно, но и вредно. Поэтому нескольким различным объектам-клиентам требуется обращаться к одному и тому же объекту и иметь гарантии, что объектов этого типа в системе будет не более одного. Синхронизированные методы снижают производительность, обращение к ним будет выполняться в несколько раз медленней, нежели к обычному методу. Если приложение рассчитано на тысячи пользователей одновременно, многие потоки будут просто ждать пока один единственный поток не выйдет из метода. В этом случае необходимо использовать шаблон (паттерн) проектирования Singleton, который обеспечивает существование одного (реже более одного) экземпляра класса, без возможности прямого создания этого класса. Самая простая реализация этого паттерна может иметь вид:

```
public class Singleton {
   private static final Singleton INSTANCE = new Singleton();
   // Private constructor prevents instantiation from other classes
   private Singleton() {
   }
   public static Singleton getInstance() {
      return INSTANCE;
   }
}
```

Эта реализация *потокобезопасна*. Объект INSTANCE создается тогда, когда класс инициализируется. Например, это может произойти, когда будет вызван какой-нибудь статический метод класса. Конструктор объявляется private, доступ к экземпляру класса возможен только через getInstance(). Такая реализация, однако, может не подойти, если создание экземпляра класса требует много ресурсов. В этом случае нужно подумать об *отложенной инициализации* синглетона. Отложенной (ленивой) инициализацией (lazy initialization) называется приём в программировании, когда некоторая ресурсоёмкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован её результат. Таким образом, инициализация выполняется «по требованию», а не заблаговременно.

```
public class Singleton {
   private static Singleton instance;
```

```
public static synchronized Singleton getInstance() {
   if (instance == null) {
      instance = new Singleton();        }
   return instance;
}    }
```

В класс добавляется закрытая статическая переменная-член класса, содержащая ссылку на соответствующий объект (ее начальное значение NULL). В класс добавляется открытый статический метод, который создает экземпляр объекта данного класса, если значение упомянутой выше переменной-члена класса равно NULL (и помещает в нее указатель на созданный объект). Метод возвращает значение этого экземпляра класса. Статус конструктора класса устанавливается защищенным или закрытым, так что никто не сможет самостоятельно создать экземпляр объекта данного класса в обход его механизма статического конструктора.

Как было сказано выше, синхронизированные методы снижают производительность. Решение данной проблемы состоит в том, чтобы выполнить синхронизацию после проверки статической переменной на NULL, а затем осуществить повторную проверку, чтобы удостовериться, что экземпляр объекта данного класса все еще не создан. Данный метод получил название double_checked locking (блокировка с двойной проверкой).

```
public class Singleton {
      private static volatile Singleton instance;

      public static Singleton getInstance() {
      Singleton localInstance = instance;
      if (localInstance == null) {
         synchronized (Singleton.class) {
            localInstance = instance;
            if (localInstance == null) {
               instance = localInstance = new Singleton();
            }
         }
      }
      return localInstance;    }    }
```

2.6. БЕСКОНЕЧНАЯ ОТСРОЧКА (INDEFINITE POSTPONEMENT)

При организации доступа параллельных задач к некоторому общему ресурсу данных возможно создание бесконечной отсрочки. Если одна или несколько задач должны ожидать до тех пор, пока не произойдет некоторое событие или не создадутся определенные условия, а ожидаемое событие или условие, которое должно произойти (или наступить), в действительности не происходит (или не наступает), то приостановленные задачи будут вечно находиться в состоянии ожидания. Один поток может заставить «голодать» другие потоки, в то время как он потребляет все доступное процессорное время. Если приложение содержит приоритетные и низко приоритетные потоки, которые должны взаимодействовать друг с другом, голодание более низко приоритетных потоков может блокировать другие потоки и создать узкие места в производительности. Такое состояние получило название бесконечной отстрочки (indefinite postponement), lockout или starvation (голодание).

В качестве примера может быть рассмотренной выше ситуации может быть следующий код апплета, который должен обеспечить постоянно обновляемое отображение времени в окне апплета:

```
import java.applet.*;

import java.awt.*;

import java.util.*;

public class TimeDisplay extends Applet {

int hours, mins, secs;

public void start() {

while(true) {

Calendar tm = Calendar.getInstance();

//Calendar object has date and time from machine clock
```

```
hours = tm.get(Calendar.HOUR);

mins = tm.get(Calendar.MINUTE);

secs = tm.get(Calendar.SECOND);

repaint(); }}

public void paint(Graphics g) {

g.drawString("Time: "+hours+":"+mins+":"+secs, 50, 50);}}
```

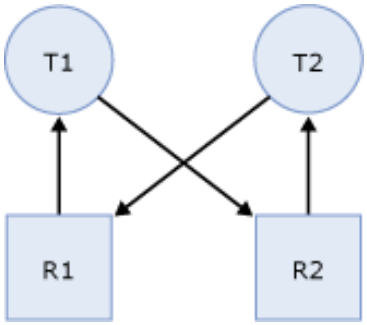
Этот апплет не будет работать, так как графический вывод в методе paint(), реализован в отличном и имеющем более низкий приоритет потоке, чем поток самого апплета, и, поэтому, его выполнение будет отложено на неопределенный срок. Работающий пример апплета приведен ниже.

```
import java.applet.*;

import java.awt.*;

import java.util.*;

public class TimeDisplay_Correct extends Applet implements Runnable {

int hours, mins, secs;

Thread apltThread;

public void init() {

if (apltThread == null) {

apltThread = new Thread(this); //make thread from runnable object

apltThread.start();}}

public void run() {

while(true) {

Calendar tm = Calendar.getInstance();
```

```
//Calendar object has date and time from machine clock
hours = tm.get(Calendar.HOUR);
mins = tm.get(Calendar.MINUTE);
secs = tm.get(Calendar.SECOND);
repaint();
//pause between iterations: give lower-priority threads a chance:
try {
Thread.sleep(500);
} catch(InterruptedException e) {System.exit(1);}
}}
public void paint(Graphics g) {g.drawString("Time: "+hours+":"+mins+":"+secs, 50, 50);}}
```

2.7. ВЗАИМНАЯ БЛОКИРОВКА

Взаимная блокировка *(deadlock*) — ситуация в многозадачной среде, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами. Один поток ожидает другой поток, тот, в свою очередь, ждет освобождения еще одного потока и т. д., пока эта цепочка не замыкается на поток, который ожидает освобождения первого потока. Получается замкнутый круг потоков, которые дожидаются освобождения друг друга. Например, задача T1 блокирует ресурс R1 (изображается в виде стрелки от R1 к T1) и запросила блокировку ресурса R2 (изображается в виде стрелки от T1 к R2). Задача T2 блокирует ресурс R2 (изображается в виде стрелки от R2 к T2) и запросила блокировку ресурса R1 (изображается в виде стрелки от T2 к R1). Так как ни одна из задач не может продолжиться до тех пор, пока не освободится ресурс, а ни один из ресурсов не может быть освобожден до тех пор, пока не продолжится задание, существует состояние взаимоблокировки.



Коффман [12] и другие исследователи доказали, что для возникновения тупиковой ситуации должны выполняться четыре условия одновременно.

1. Условие взаимного исключения. Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. Условие удерживания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурсов. У процесса нельзя забрать принудительно ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Пример:

```
public class TestThread {
  public static Object Lock1 = new Object();
  public static Object Lock2 = new Object();
    public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
    ThreadDemo2 T2 = new ThreadDemo2();
    T1.start();
    T2.start();   }
    private static class ThreadDemo1 extends Thread {
    public void run() {
      synchronized (Lock1) {
```

```
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2) {
              System.out.println("Thread 1: Holding lock 1 & 2...");
            }        }      }   }
      private static class ThreadDemo2 extends Thread {
        public void run() {
          synchronized (Lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (Lock1) {
              System.out.println("Thread 2: Holding lock 1 & 2...");
            }        }      }   } }
```

Для того чтобы разрешить возникший в предыдущей программе deadlock, достаточно поменять порядок выполняемы блокировок synchronized (Lock1), synchronized (Lock2).

```
    public class TestThread {
      public static Object Lock1 = new Object();
      public static Object Lock2 = new Object();
        public static void main(String args[]) {
          ThreadDemo1 T1 = new ThreadDemo1();
        ThreadDemo2 T2 = new ThreadDemo2();
        T1.start();
        T2.start();   }
        private static class ThreadDemo1 extends Thread {
        public void run() {
          synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
```

```
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2) {
              System.out.println("Thread 1: Holding lock 1 & 2...");
            }        }      }   }
      private static class ThreadDemo2 extends Thread {
        public void run() {
          synchronized (Lock1) {
            System.out.println("Thread 2: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 2...");
            synchronized (Lock2) {
              System.out.println("Thread 2: Holding lock 1 & 2...");
            }        }      }     }     }
```

Для избежания взаимных блокировок можно предложить следующие рекомендации: захватывать везде ресурсы в одинаковом порядке или знать заранее какие ресурсы в каком порядке будут захвачены, для чего преварительно построить граф переходов между состояниями.

2.8. АКТИВНАЯ БЛОКИРОВКА

Активная блокировка (livelock) возникает, когда два потока одновременно пытаются изменить структуру данных, но каждый из них должен начинать свою операцию сначала из-за изменений, произведенных другим потоком. Таким образом, каждый поток беспрестанно повторяет попытки в цикле. Если состояние deadlock можно характеризовать как: "Me first, Me first", то состояние livelock: " You first, You first". Ниже приведен простой пример активной блокировки, иллюстрирующий ситуацию, когда когда муж и жена пытается съесть суп, но имеют только одну ложку. Каждый из супругов слишком вежлив, и предлагает ложку другому [13].

```
public class Livelock {
   static class Spoon {
      private Diner owner;
```

```
    public Spoon(Diner d) { owner = d; }
    public Diner getOwner() { return owner; }
    public synchronized void setOwner(Diner d) { owner = d; }
    public synchronized void use() {
        System.out.printf("%s has eaten!", owner.name);        }    }

static class Diner {
    private String name;
    private boolean isHungry;
    public Diner(String n) { name = n; isHungry = true; }
    public String getName() { return name; }
    public boolean isHungry() { return isHungry; }
    public void eatWith(Spoon spoon, Diner spouse) {
        while (isHungry) {
            // Don't have the spoon, so wait patiently for spouse.
            if (spoon.owner != this) {
                try { Thread.sleep(1); }
                catch(InterruptedException e) { continue; }
                continue;               }
            // If spouse is hungry, insist upon passing the spoon.
            if (spouse.isHungry()) {
                System.out.printf(
                    "%s: You eat first my darling %s!%n", name, spouse.getName());
                spoon.setOwner(spouse);
                continue;               }
            // Spouse wasn't hungry, so finally eat
            spoon.use();
            isHungry = false;
            System.out.printf(
                "%s: I am stuffed, my darling %s!%n",  name, spouse.getName());
            spoon.setOwner(spouse);
        }       }    }
public static void main(String[] args) {
    final Diner husband = new Diner("Bob");
```

```
        final Diner wife = new Diner("Alice");
        final Spoon s = new Spoon(husband);
        new Thread(new Runnable() {
            public void run() { husband.eatWith(s, wife); }
        }).start();
        new Thread(new Runnable() {
            public void run() { wife.eatWith(s, husband); }
        }).start();
    }       }
```

ГЛАВА 3.
ТИПОВЫЕ МОДЕЛИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Итеративный параллелизм используется в том случае, когда в программе есть несколько процессов (часто идентичных), каждый из которых содержит один или несколько циклов. Таким образом, каждый процесс является итеративной программой. Процессы программы работают совместно над решением одной задачи; они взаимодействуют и синхронизируются с помощью разделяемых переменных или передачи сообщений. Итеративный параллелизм чаще всего встречается в научных вычислениях.

Рекурсивный параллелизм может использоваться, когда в программе есть одна или несколько рекурсивных процедур, и их вызовы независимы, т.е. каждая из них работает над своей частью общих данных. Рекурсия часто применяется в императивных языках программирования, особенно при реализации алгоритмов типа “разделяй и властвуй” или “перебор с возвращением” (backtracking). Рекурсия является одной из фундаментальных парадигм и в символических, логических, функциональных языках программирования. Рекурсивный параллелизм используется для решения таких комбинаторных проблем, как сортировка, планирование (задача коммивояжера) и игры (шахматы и другие).

3.1. ИТЕРАТИВНЫЙ ПАРАЛЛЕЛИЗМ: УМНОЖЕНИЕ МАТРИЦ

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции дают прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

Пусть имеются массивы MatrixA[n][n], MatrixB[n][n]. Необходимо вычислить их произведение MatrixC[n][n]. Типичная программа умножения матриц выглядит следующим образом:

```
for (int i=0;i<n;i++)
  for (int j=0;j<n;j++) {
      MatrixC [i][j]=0.0;
    for (int k=0;k<n;k++)
      MatrixC [i][j]+= MatrixA [i,k]* MatrixB[k,j];   }
```

Части программы могут выполняться параллельно, если они независимы по данным, по управлению и по операциям вывода. При умножении матриц вычисления промежуточных произведений являются независимыми операциями, так как внутренний цикл только читает переменные из массивов MatrixA и MatrixB (строку из MatrixA и столбец из MatrixB). Запись же происходит в единственный элемент массива MatrixC. Поскольку множества переменных, в которых производится запись, не пересекаются, вычисление элементов с могут быть выполнены параллельно.

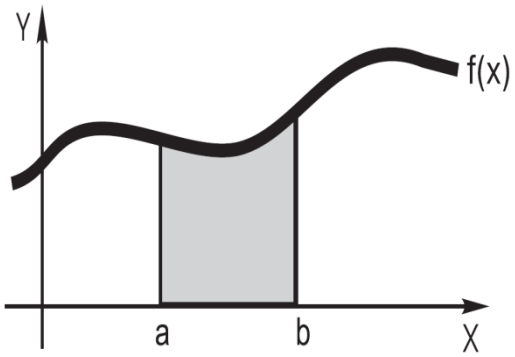
```
public class MatrixMultiThread {
  public static final int NUM_OF_THREADS = 9;
    public static void main(String args[])
   {
      int row;
      int col;
      int MatrixA[][] = {{1,4},{2,5},{3,6}};
      int MatrixB[][] = {{8,7,6},{5,4,3}};
```

```
int MatrixC[][] = new int[3][3];
int threadcount = 0;
    Thread[] thrd = new Thread[NUM_OF_THREADS];
     try{
        for(row = 0 ; row < 3; row++)  {
           for (col = 0 ; col < 3; col++ ){
                 // creating thread for multiplications
        thrd[threadcount] =
        new Thread(new MultiplicationThreading(row, col,MatrixA, MatrixB, MatrixC));
        thrd[threadcount].start(); //thread start
        thrd[threadcount].join(); // joining threads wait until all thread complete their work
        threadcount++; }   }
     }
     catch (InterruptedException ie){}

     // printing matrix A
     System.out.println(" Matrix A: ");
     for(row = 0 ; row < 3; row++){
           for (col = 0 ; col < 2; col++ )
              System.out.print("  "+MatrixA[row][col]);
              System.out.println();                }
     // printing matrix B
     System.out.println(" Matrix B: ");
     for(row = 0 ; row < 2; row++) {
           for (col = 0 ; col < 3; col++ )
           System.out.print("  "+MatrixB[row][col]);
           System.out.println();     }
     // printing resulting matrix C after multiplication
     System.out.println(" Resulting Matrix C: ");
     for(row = 0 ; row < 3; row++) {
     for (col = 0 ; col < 3; col++ )
           System.out.print("  "+MatrixC[row][col]);
           System.out.println();     }       }      }
```

```
public class MultiplicationThreading implements Runnable
{
    private int row;
    private int col;
    private int MatrixA[][];
    private int MatrixB[][];
    private int MatrixC[][];
    public MultiplicationThreading(int row, int col,int MatrixA[][],
                                   int MatrixB[][], int MatrixC[][] )
    {
       this.row = row;
       this.col = col;
       this.MatrixA = MatrixA;
       this.MatrixB = MatrixB;
       this.MatrixC = MatrixC;
    }
     @Override
    public void run()
    {
      // multiplication perform here
          for(int k = 0; k < MatrixB.length; k++)
          MatrixC[row][col] += MatrixA[row][k] * MatrixB[k][col];
            }       }
```

3.2. РЕКУРСИВНЫЙ ПАРАЛЛЕЛИЗМ: АДАПТИВНАЯ КВАДРАТУРА

Рассмотрим применение рекурсивного параллелизма на примере вычисления определенного интеграла функции по методу адаптивной квадратуры. Пусть необходимо вычислить значение определенного интеграла функции f(x) на отрезке [a,b] с точность epsilon. Известно, что определенный интеграл равен площади под кривой, определенной функцией f(x).



Поступим следующим образом: найдем m – середину отрезка между a и b. По правилу трапеций найдем значение площадей на отрезках [a,m], [m,b], [a,b]. Если сумма меньших площадей равна большей площади с заданной точностью, то решение найдено. Если точность не достигнута, каждая задача делится на две подзадачи, и процесс решения повторяется. Этот процесс можно запрограммировать следующим образом:

```
double quad(double left, right, fleft, fright, lrarea) {
  double mid=(left+right)/2;
  double fmid=f(mid);
  double larea=(fleft+fmid)*(mid-left)/2;
  double rarea=(fmid+fright)(right-mid)/2;
  if (abs(larea+rarea-lrarea)>epsilon) {
        larea=quad(left,mid,fleft,fmid,larea);
        rarea=quad(mid,right,fmid,fright,rarea);
 }
 return (larea+rarea);
}
area=quad(a,b,f(a),f(b),(f(a)+f(b))*(b-a)/2);
```

В этой схеме вызовы рекурсивной функции quad независимы, и следовательно, могут быть произведены параллельно.

```
interface TheFunction {
```

```
   public double evaluate(double x);
   public String toString();}
class MyFunction implements TheFunction {
   public double evaluate(double x) { return x*x; }
   public String toString() { return " x**2"; }
   public double definiteIntegral(double a, double b) {
      return (b*b*b - a*a*a)/3.0;   }}
class Area extends Thread {
   private double p;
   private double q;
   private double epsilon;
   private double result;
   private TheFunction f;
   public Area(double a, double b, double eps, TheFunction fn) {
      p = a;  q = b;  epsilon = eps;  f = fn;   }
   public double getResult() { return result; }
   private static double trapezoidArea(double p, double q, TheFunction f) {
      double area = (Math.abs(q-p))/2 * (f.evaluate(p) + f.evaluate(q));
      return area;   }
    public void run() {
```

```
        double bigArea = trapezoidArea(p, q, f);
        double leftSmallArea = trapezoidArea(p, ((p+q)/2), f);
        double rightSmallArea = trapezoidArea(((p+q)/2), q, f);
        double sumOfAreas = leftSmallArea + rightSmallArea;
        double relError = Math.abs(bigArea - sumOfAreas);
        if (relError <= (epsilon * sumOfAreas)) result = bigArea;
        else {
          Area leftArea =  new Area(p, (p+q)/2, epsilon, f);
          leftArea.start();
          Area rightArea = new Area((p+q)/2, q, epsilon, f);
          rightArea.start();
          try { leftArea.join(); }
            catch (InterruptedException e) { /* ignored */ }
          try { rightArea.join(); }
            catch (InterruptedException e) { /* ignored */ }
          result = leftArea.getResult() + rightArea.getResult();     }   } }
class AdaptiveQuadrature {
  public static void main(String[] args) {
    System.out.println("Java version=" + System.getProperty("java.version")
      + ", Java vendor=" + System.getProperty("java.vendor")
```

```
    + "\nOS name=" + System.getProperty("os.name")
    + ", OS arch=" + System.getProperty("os.arch")
    + ", OS version=" + System.getProperty("os.version")
    + ", CPUs=" + Runtime.getRuntime().availableProcessors()
    + "\n" + new java.util.Date());
  double a = 0, b = 1, epsilon = 0.01;
  TheFunction fn = new MyFunction();
  System.out.println("Adaptive Quadrature of" + fn + " from "
    + a + " to " + b + " with relative error " + epsilon);
  Area area = new Area(a, b, epsilon, fn);
  area.start();
  try { area.join(); }
    catch (InterruptedException e) { /* ignored */ }
  double result = area.getResult();
  System.out.println("Result for" + fn + " = " + result);
  System.out.println("Correct result = "    + ((MyFunction) fn).definiteIntegral(a, b));
  System.exit(0);    } }
```

3.3. ПАКЕТ JAVA.UTIL.CONCURRENT

Параллелизм Java, основанный на потоках и блокировках, — это очень низкоуровневый механизм, работать с которым зачастую довольно сложно. Чтобы упростить

этот процесс, в Java 5 появился набор библиотек для обеспечения параллелизма, называемый java.util.concurrent. В нем предоставляется несколько инструментов для написания параллельного кода. К таким механизмам относятся CountDownLatch, CyclicBarrier, Semaphore, BlockingQueue и другие.

Класс CountDownLatch реализует механизм синхронизации с помощью которого несколько потоков могут блокироваться в ожидании совершения некоторого множества событий.

CyclicBarrier позволяет нескольким потоком ожидать друг друга в определенной точке выполнения. Этот класс часто используется в реализациях алгоритмов, в которых присутствуют некие фиксированные повторяющиеся действия с необходимостью синхронизации в какой-либо момент.

Semaphore – механизм синхронизации управляющий определенным количеством разрешений. При создании семафора в конструктор передаётся целое число, характеризующее максимально количество доступных захватов семафора (методы acquire). По завершению действий ограниченных семафором его нужно освободить (методы release) подобно блокировкам. Семафоры часто используются для ограничения доступа к фиксированным ресурсам.

Exchanger – основное предназначение данного класса организация обмена объектами между двумя потоками. Класс может быть испозвользован для передачи только одного объекта или же просто как синхронизатор двух потоков.

Интерфейс BlockingQueue расширяет интерфейс коллекции Queue блокирующимися методами, которые возможно использовать в качестве механизмов синхронизации. BlockingQueue реализуют несколько классов, среди которых ArrayBlockingQueue, ConcurrentLinkedQueue, LinkedBlockingQueue, PriorityBlockingQueue, различные классы деков (двусторонняя очередь). Каждая из реализаций имеет свои слабые и сильные стороны: вместимость очереди, расположение элементов при добавлении (приоритет), накладные расходы при параллельном доступе и прочее. Блокирующиеся очереди могут быть полезны для реализации паттерна производитель-потребитель (producer-consumer).

В Java 7 появился набор библиотек для обеспечения параллелизма технологии Fork/Join. Модель *fork-join framework*, реализованная в языке Java, основывается на модели программирования "разветвление-объединение" (fork-join). Если исходную задачу можно рекурсивно разбить на подзадачи, тогда каждую подзадачу можно поставить в очередь на решение (шаг fork), затем получить результаты от подзадач (шаг join) и их слить. Таким образом, выполняется разветвление на несколько потоков, чтобы создать группу потоков. Эти потоки

выполняются параллельно. В конце параллельного участка все потоки заканчивают свою работу и снова объединяются вместе. После этого исходный (или "главный") поток продолжает выполняться до тех пор, пока не начнется следующий параллельный участок (или не наступит конец программы). Пример перемножения матриц, с использованием fork-join framework, приведен на http://www.java2s.com/Book/Java/Thread-Conncurrent/ForkJoin_Framework.htm.

3.4. ТЕСТИРОВАНИЕ МНОГОПОТОЧНЫХ ПРОГРАММ

При многопоточном программировании имеется несколько потоков, которые выполняют разные "программы", взаимодействующие друг с другом. Например, поток пользовательского интерфейса, поток вычислений, поток обработки ввода/вывода, клиентских запросов на сервер. Многопоточное программирование позволяет упростить (при адекватном подходе) архитектуру программы. Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря возможности выгодно использовать время простоя, неизбежно возникающее в ходе выполнения большинства программ. Как известно, большинство устройств ввода-вывода, будь то устройства, подключенные к сетевым портам, накопители на дисках или клавиатура, работают намного медленнее, чем центральный процессор. Поэтому большую часть своего времени программе приходится ожидать отправки данных на устройство ввода-вывода или приема информации из него. А благодаря многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя.

Параллельное программирование применяется для численных расчетов, или, например, в компьютерной графике. Использование конвейеризации и большого количества вычислительных ядер позволяет получить значительный прирост в скорости вычислений.

Тестирование и отладка многопоточных программ чрезвычайно сложны. Большинство проблем, связанных с поточностью, непредсказуемы по своей природе, и могут вообще не проявляться на определенных платформах (например, таких как однопроцессорные системы) или при нагрузке ниже определенного уровня. Поэтому важно разрабатывать приложения, с самого начала помня о безопасности потоков.

При тестировании последовательной программы разработчик может отследить ее логику в пошаговом режиме. Программист может отыскать ошибки в программе, используя соответствующие входные данные и исходное состояние программы в пошаговом режиме. В многопоточных программах трудно воспроизвести точный контекст

задач из-за разных стратегий планирования, применяемых в операционной системе, динамически меняющейся рабочей нагрузки, квантов процессорного времени, приоритетов процессов и потоков. Даже если ошибка обнаружена, ее часто сложно воспроизвести повторно.

Отладка последовательной программы основана на том, что степень предсказуемости начального и текущего состояний программы определяется входными данными. Когда программист переходит к отладке многопоточного кода, он обычно сталкивается с совершенно уникальными проблемами: в различных операционных системах применяются разные стратегии планирования, нагрузка на вычислительную систему динамически изменяется, приоритеты процессов могут различаться и т. д. Так, точное воссоздание программы в некоторый момент ее выполнения (тривиальная задача для последовательной отладки) значительно усложняется при переходе к параллельной программе, что связано с недетерминированным поведением последней. Иными словами, поведение запущенных в системе процессов, а именно их выполнение и ожидание выполнения, взаимные блокировки и проч., зависит от случайных событий, происходящих в системе.

При тестировании многопоточных программ можно рекомендовать разбить программу на несколько блоков вида читать *разделяемые данные /преобразовать данные /обновить разделяемые данные*. Тогда часть программы *преобразовать данные* можно будет протестировать с помощью стандартных методов, поскольку это будет обычный однопоточный код. И трудная задача тестирования многопоточных преобразований будет сведена к тестированию чтения и обновления разделяемых данных. Программу необходимо подвергнуть большей нагрузке, т.е. выполнить ее многократно и, возможно, с большим количеством потоков. Если некоторая ошибка возникает только при определенном порядке планирования потоков, то в этом случае будет больше вероятность ее проявления.

Для выявления возможных ошибок работы многопоточной программы необходимо ответить на вопросы:

• Какие данные программы нужно защищать от одновременного доступа?

• Как обеспечивается защита этих данных?

• В каком участке программы могут в этот момент находиться другие потоки?

• Существуют ли ограничения на порядок выполнения операций в этом и каком-либо другом потоке? Как гарантируется соблюдение этих ограничений?

• Верно ли, что данные, загруженные этим потоком, все еще действительны? Не могло ли случиться, что их изменили другие потоки?

• Если предположить, что другой поток может изменить данные, то к чему это приведет и как гарантировать, что этого никогда не случится?

Методики поиски ошибок в параллельных приложениях, как и в последовательных можно разделить на динамический анализ, статический анализ, анализ ошибок на уровне моделей вычислений и доказательство корректности программы [32].

Динамический анализ подразумевает под собой необходимость запуска приложения и выполнения различный последовательностей действий, целью которых ставится выявление некорректного поведения программы. Последовательность действий может задаваться как человеком при ручном тестировании, так и с использованием различных инструментов реализующих нагрузочное тестирование или, например, проверку целостности данных. Достаточно трудно осуществить покрытие тестами всего параллельного кода. Часто зафиксировать состояние гонки (race conditions) удается, только если оно было в данном сеансе работы программы.

Статический анализ работает только с программным кодом приложения, не требуя его запуска. Статический анализ кода это процесс выявления ошибок и недочетов в исходном коде программ. Данный подход имеет ряд преимуществ и недостатков, которые, впрочем, можно компенсировать параллельным применением динамического анализа. В случае параллельных приложений статический анализ крайне сложен, так как часто неизвестен допустимый набор входных значений для различных функций и способ их вызова.

Анализ ошибок на уровне моделей вычислений представляет собой автоматическую генерацию тестов по заданным правилам. На практике проверка на основе моделей действенна лишь для небольших базовых блоков приложения.

Все перечисленные методики имеют свои недостатки, что не позволяет положиться при разработке параллельных программ только на одну из них.

Контрольные вопросы

1. Преимущества многопоточной архитектуры Java.
2. Парадигмы параллельного программирования.
3. Процессы и потоки.
4. Порядок передачи управления потокам. Приоритет потока.
5. Что такое ordering, visibility, atomicity, happend-before, mutual exclusion?
6. Способы реализации многопоточности в Java.
7. Методы класса Thread, запускающие и останавливающие поток на выполнение.
8. Реализации классом интерфейса Runnable.
9. Отличие Thread.start() и Thread.run().
10. В каких состояниях может находиться поток?
11. В каких ситуациях поток является невыполняемым?
12. Что такое прерывание потока?
13. Когда могут возникать исключительные ситуации при работе с потоками?
14. Что такое группы потоков?
15. Чем интерфейс Callable отличается от интерфейса Runnable?
16. Базовый набор средств синхронизации потоков.
17. Управлению потоками и задачами с помощью интерфейса ExecutorService.
18. Понятие пула потоков, его создание и настройка.
19. Преимущества использования пулов потоков.
20. Блокирующая и неблокирующая синхронизации.
21. Volatile переменные и atomic переменные.
22. Что такое монитор? Как и для чего используются монитор?
23. Что такое семафор? Как и для чего используются семафоры?
24. Что такое мьютекс? Привести пример использования.
25. Назовите отличия synchronize{} и ReentrantLock.
26. Перечислите известные Вам способы борьбы с priority inversion.
27. Потокобезопасные классы. Что такое "safe publication"?
28. Потенциальные ошибки, связанные с параллелизмом данных и задач. Гонка данных.
29. Бесконечное ожидание (starvation). Привести пример.
30. Взаимная (deadlock) и активная (livelock) блокировки. Привести пример.
31. Может ли единственный процесс находиться в состоянии deadlock?

32. Что такое guarded lock?
33. Что такое "recursive parallelism"?
34. Что такое "iterative parallelism"?
35. Отличие параллельности, асинхронности и многопоточности.
36. Какие средства синхронизации взаимодействующих потоков содержит пакет java.util.concurrent?

Литература

1. Гергель В.П. Теория и практика параллельных вычислений. М.:Интернет-Университет, БИНОМ. Лаборатория знаний, 2007.
2. Богачев К. Основы параллельного программирования. Учебное пособие. Изд. Бином, 2013
3. Б. Эккель. Философия Java (Thinking in Java). Изд.: Питер, 2014 .
4. Герберт Шилдт. Java 8. Полное руководство. Изд.: Вильямс, 2014
5. Magee J., Kramer J. Concurrency: State Models & Java Programs, Wiley-IEEE Press, 2006
6. Blackout Final Report, August 14, 2003, http://www.ferc.gov/industries/electric/indus-act/reliability/blackout/ch5.pdf
7. http://www.javacodegeeks.com/
8. C. A. R. Hoare. Monitors: an operating system structuring concept // Communications of the ACM, v.17 n.10, p.549-557, 1974
9. E. Dijkstra. Go to statement considered harmful// Communications of the ACM, 11(3):147–148, 1968
10. https://docs.oracle.com/javase/tutorial/
11. Б. Эванс, М. Вербург, Java. Новое поколение разработки. Изд.: Питер, 2014
12. Coffman E.G.Jr., Elphick, M.J., Shoshani A. System deadlock ACM Computing, Surveys 3, 2 (June), 67-78., 1971
13. http://stackoverflow.com/
14. П. Нимейр, Д. Леук. Программирование на Java Исчерпывающее руководство для профессионалов. Изд.: ЭКСМО, 2014

15. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Изд.: Питер, 2014 .
16. D. Lea Concurrent Programming in Java: Design Principles and Pattern. Addison-Wesley,
17. B. Goetiz, T. Peierls. Java concurrency in practice. Addison-Wesley Professional, 2006.
18. V. K. Garg. Concurrent and Distributed Computing in Java. Wiley-IEEE Press, 2004.
19. Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-805
20. K.Sierra, B. Bates. SCJP Sun Certified Programmer for Java 6 Exam 310-065
21. S.Oaks, H.Wong. Java Threads, 3rd Edition - O'Reilly Media
22. B. Lewis, D. J. Berg Multithreaded Programming with Java Technology
23. З. Медникс, Л. Дорнин, Б. Мик, Н. Масуми. Программирование под Android (Java Programming for the New Generation of Mobile Devices). Изд.: Питер, 2014
24. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull The Real-Time Specification for Java, Addison-Wesley Professional, 2013.
25. B. Whitman, A. Mathur, T. Korb, Start Concurrent: an Introduction to Problem Solving in Java with a Focus on Concurrency. Kindle Edition, 2014
26. M. Herlihy, N.Shavit. The Art of Multiprocessor Programming Morgan Kaufmann Publishers Inc., 2012
27. D. C. Schmidt, M.Stal, H.Rohnert, F. Buschmann. Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects. Publisher: Wiley, 2013
28. Netzer R., Miller B. What Are Race Conditions? Some Issues and Formalizations. In ACM Letters On Programming Languages and Systems, 1(1), 1992. P. 74–88
29. Allen B. Downey, The Little Book of Semaphores. Green Tea Press, 2009
30. Грегори Р. Эндрюс. Основы многопоточного, параллельного и распределённого программирования. Изд. Вильямс, 2003.
31. Garg V. Concurrent and Distributed Computing in Java, Wiley-IEEE Press, 2004
32. Z. D. Luo, Y. Nir-Buchbinder, R. Das. Java concurrency bug patterns for multicore systems. http://www.ibm.com/developerworks/library/j-concurrencybugpatterns/
33. Таненбаум Э., М. ван Стеен Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003

Задания

1. Написать программу, которая печатает "Hello World" пять раз из пяти разных потоков.
2. При заданном значении x вычислить значение функции f(x)=sin(x)+cos(x)+tg(x), вычисление значений sin(x), cos(x), tg(x) реализовать в отдельных потоках.
3. Написать программу, в которой два потока прибавляли единицу к общей целой переменной 1000000 раз.
4. Написать программу, в которой один поток увеличивает целую переменную на единицу 1000000 раз, и второй поток печатает ее значение.
5. Распечатать четные и нечетные числа в диапазоне от 1 до 10000, используя потоки.
6. Изобразить в приложении различные фигуры, вращающиеся в плоскости экрана вокруг своего центра с разной скоростью. Каждому объекту соответствует поток с заданным приоритетом.
7. Организовать сортировку массива с помощью разных потоков.
8. Написать программу, в которой одному потоку разрешается присваивать общей переменной только нечетные значения, второму – только четные. Модифицируя общую переменную только этими двумя потоками, присвоить ей значения от 1 до 120. Изначально общая переменная имеет значение 0.
9. Написать программу для нахождения количества простых чисел в диапазоне от 1 до 10000, разделив этот диапазон равномерно между имеющимися потоками.
10. Написать программу для нахождения суммы всех целых чисел от 1 до 10000, разделив диапазон слагаемых равномерно между имеющимися потоками.
11. Написать программу для поиска минимального элемента в массиве. Массив разделить равномерно на части между потоками.

12. Напишите программу, которая вычисляет число π с заданной точностью при помощи ряда Лейбница

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\ldots=\frac{\pi}{4}$$

Количество потоков программы должно определяться параметром командной строки.

13. Написать программу для вычисления интеграла от заданной функции на заданном отрезке методом центральных прямоугольников (трапеции, Симпсона). Количество потоков программы должно определяться при запуске программы.

14. Реализовать процедуру поиска выхода из лабиринта размерности N на N. Лабиринт хранится в текстовом файле. В первой строке файла хранятся числа N и K (размерность лабиринта и количество расчетных блоков по строкам и по столбцам соответственно). В лабиринте могут находиться следующие элементы: '.' - пустая клетка, '#' - стена, '*' - вход в лабиринт, 'E' - выход из лабиринта.

15. Написать программу для нахождения минимального значения функции от одной переменной на заданном отрезке, используя потоки.

16. Написать многопоточную программу поиска максимального элемента квадратной матрицы.

17. Написать многопоточную программу перемножения двух векторов.

18. Написать многопоточную программу умножения матрицы на вектор.

19. Написать многопоточную программу, которая создает четыре потока, исполняющих одну и ту же функцию. Эта функция должна распечатать последовательность текстовых строк, переданных как параметр.

20. Программа должна содержать два потока. Один из них должен считывать вводимые пользователем строки и помещать их в начало списка. При вводе пустой строки программа должна выдавать текущее состояние списка. Второй поток должен каждые пять секунд сортировать список в алфавитном порядке.

21. Напишите многопоточную программу для подсчёта кол-ва слов в тексте.

22. Напишите многопоточную программу чтения-записи данных файл.

23. Напишите оконное приложение или апплет с двумя кнопками. Нажатие каждой кнопки инициирует появление строки с соответствующим текстом. Строки движутся навстречу друг другу.

24. Разработать модель системы управления лифта. Лифт имеет максимальную вместимость N человек и может перемещаться вверх и вниз.

25. Напишите программу моделирования железнодорожной станции. На станции может находиться только один поезд.

26. Напишите программу моделирования работы музея. Музей позволяет посетителям войти через восточный вход и выйти через западный выход. Вход и выход каждого посетителя контролируется турникетом. Во время работы музея турникеты открыты и на вход и на выход. При закрытии музея только на выход.

27. Напишите программу моделирования магазина фруктов. Фермер может производить различные типы фруктов (яблоко, апельсин, виноград, арбуз и др.), и привозить их в магазин. Магазин имеет ограниченную емкость склада, и фермеры должны стоять в очереди, если склад уже заполнен. Потребители могут придти в магазин в любое время и приобрести желаемые фрукты.

28. Напишите программу, реализующую функции простейшего онлайн банкомата. Доступ к одному и тому же сберегательному счету имеет несколько человек, каждый из них может внести или снять деньги.

29. Разработайте имитатор производственной линии, изготавливающей детали. Деталь собирается из винтика С и модуля, который, в свою очередь, состоит из болтов А и В. Для изготовления болта А требуется одна секунда, В – две секунды, С – три секунды. Задержку изготовления деталей имитируйте при помощи sleep(). Используйте семафоры-счетчики.

30. Написать программу синхронизированный доступ к ресурсам (файлам). Для каждого процесса необходимо создать отдельный поток выполнения.

31. Резервировать места в концертном зале можно удаленно. Необходимо разработать систему бронирования.

32. Бензоколонка самообслуживания имеет ряд терминалов управления отпуска топлива. Разработать систему работы бензоколонки.

33. Написать программу управления парихмахерской, в которой три кресла, три парикмахера, зал ожидания, в котором четыре клиента могут разместиться на диване. Клиент не может войти в парикмахерскую, если места ожидания заняты. Плата принимается в один момент времени только от одного клиента.

34. Написать систему, моделирующую работу call-центра организации, в котором работает несколько операторов. Оператор может обслуживать только одного клиента, остальные должны ждать своей очереди. Клиент может положить трубку и перезвонить еще раз через некоторое время.

35. На маршруте несколько остановок. На одной остановке может останавливаться несколько автобусов одновременно, но не более заданного числа. Написать программу, моделирующую автобусное движение.

36. В ресторане быстрого обслуживания есть несколько касс. Посетители стоят в очереди в конкретную кассу, но могут перейти в другую очередь при уменьшении или исчезновении там очереди. Написать программу, моделирующую работу ресторана.

37. В горах существует два железнодорожных тоннеля, по которым поезда могут двигаться в обоих направлениях. Необходимо обеспечить безопасное прохождение тоннелей в обоих направлениях. Поезд можно перенаправить из одного тоннеля в другой при превышении заданного времени ожидания на проезд. Написать программу, моделирующую движение поездов.

38. Написать программу сетевого чата, предназначенного для обмена текстовыми сообщениями в сети и работающего по принципу клиент-сервер.

Никитенкова Светлана Павловна

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA

Учебно-методическое пособие

Компьютерная верстка – Никитенкова С.П.

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
603950, Нижний Новгород, пр. Гагарина, 23