Нижегородский государственный университет им. Н.И. Лобачевского

Национальный исследовательский университет

Программа повышение конкурентоспособности ННГУ им. Н.И. Лобачевского

Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

Основная образовательная программа

010300 «Фундаментальная информатика и информационные технологии»

Квалификация (степень) выпускника – бакалавр

Форма обучения:

Очная

Учебно-методическая разработка по дисциплине

"Языки программирования"

С.П.Никитенкова

ДОПОЛНИТЕЛЬНЫЕ ГЛАВЫ КУРСА «ЯЗЫКИ ПРОГРАММИРОВАНИЯ»: ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ НА ЯЗЫКЕ JAVA

Мероприятие 7.1.1. Разработка образовательных программ подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий и высокопроизводительных вычислений

Нижний Новгород
2014 год

Аннотация

к методической работе: «**ДОПОЛНИТЕЛЬНЫЕ ГЛАВЫ КУРСА «ЯЗЫКИ ПРОГРАММИРОВАНИЯ»: ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ НА ЯЗЫКЕ JAVA »;**

автор: Никитенкова С.П.

Пособие предназначено для начального освоения практического курса параллельных вычислений на языке Java. Рассмотрен процесс проектирования приложения и отладки параллельных программ. Особое внимание уделено основным проблемам, возникающим при разработке параллельных программ: “гонка” данных, обнаружение взаимоблокировки, бесконечное ожидание и т.д. и их решение средствами языка Java. Рассмотрены примеры параллельного программирования алгоритмов решения различных типовых задач, возникающих при исследованиях математических моделей физических явлений, такие как параллельное умножение матриц, адаптивная квадратура и т.д.

.

Данное пособие предназначено для студентов младших курсов радиофизического факультета ННГУ, обучающихся по направлениям подготовки 010300 «Фундаментальная информатика и информационные технологии».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время в промышленно развитых странах 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. Для того, чтобы разрабатывать программное обеспечение необходимы практические знания параллельного и распределенного программирования.

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. Java - это наиболее распространённый открытый язык программирования, при помощи которого работают миллиарды устройств.

Параллельные вычисления — способ организации компьютерных вычислений , при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Параллельность в программе достигается путем разбиения программы на несколько процессов или потоков.

Понятие процесса является одним из основополагающих в теории и практике параллельного программирования. Процесс, это выполняющийся экземпляр программы. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (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). Для каждого процесса операционная система создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

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», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного

использования ресурсов вычислительной машины. Потоки представляют собой ключевую модель параллелизма, поддерживаемую современными компьютерами, языками программирования и операционными системами. Java является первым массовым языком программирования, который явно включает в свой состав потоки, а не рассматривает их как функцию нижележащей операционной системы.

ПОТОКИ И ПРОЦЕССЫ

Под потоком подразумевается часть выполняемого кода в процессе. Потоки позволяют одной программе состоять из параллельно выполняемых частей, причем все части имеют доступ к одним и тем же переменным. Потоки можно рассматривать как облегченные процессы, т.е. они позволяют воспользоваться многими преимуществами процессов без больших затрат на организацию взаимодействия между ними. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. 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. Поток (точно так же, как и процесс) можно представлять как последовательность команд программы, однако – в отличие от процесса – потоки одной и той же программы 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.

СОЗДАНИЕ ПОТОКА

Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существует два способа создания и запуска потока: на основе расширения класса Thread или реализации интерфейса Runnable.

```
public class My_Thread extends Thread {
@Override
public void run() {
for (int i = 0; i < 10; i++) {
System.out.println(Thread.currentThread().getName() + " - " + i);
try {
Thread.sleep(7); // остановка на 7 миллисекунд
} catch (InterruptedException e) {
System.err.print(e); } } } }
```

При реализации интерфейса Runnable необходимо определить его единственный абстрактный метод run(). Например:

```
public class My_Runnable implements Runnable {
@Override
public void run() {
for (int i = 0; i < 10; i++) {
System.out.println(Thread.currentThread().getName() + " - " + i);
try {
Thread.sleep(7);
```

```
} catch (InterruptedException e) {
System.err.println(e); } } } }
public class Main{
public static void main(String[ ] args) {
// новые объекты потоков
My_Thread t1= new My_Thread();
Thread t2= new Thread(new My_Runnable());
// запуск потоков
t1.start();
t2.start();
} }
```

Каждый поток выполняет вывод на экран чисел от 0 до 9 с указанием имени потока. При этом порядок вывода цифр не детерминирован. Это связанно с тем, что не представляется возможным определить точные периоды, когда код потоков выполняется на процессоре. На этот порядок влияют как алгоритм и реализация планировщика задач операционной системы, так и текущая загруженность системы в купе с множеством других факторов.

Поток Java может находиться в одном из следующих состояний в течение периода существования: *новый* - поток создан, но еще не запущен. *Исполняемый* - поток запущен и готов продолжить выполнение, т.е. ему может быть выделено операционной системой процессорное время (когда процессор окажется свободным). Поток, которому выделено процессорное время, является *выполняющимся*(running). *Неисполняемый* - в данное состояние поток переходит после наступления определенного события (ожидание завершения операции ввода/вывода, перевод в неактивный режим на определенное время методом sleep(), вызов методов wait()). Неисполняемый поток становится опять исполняемым при изменении его состояния (завершен ввод/вывод, завершен период пребывания в неактивном режиме и т.д.). В течение периода своего существования поток может неоднократно переходить из состояния исполняемый в состояние неисполняемый. Потоку можно назначить приоритет от 1 (константа MIN_PRIORITY) до 10 (MAX_PRIORITY) с помощью

метода setPriority(int prior). Получить значение приоритета потока можно с помощью метода getPriority().

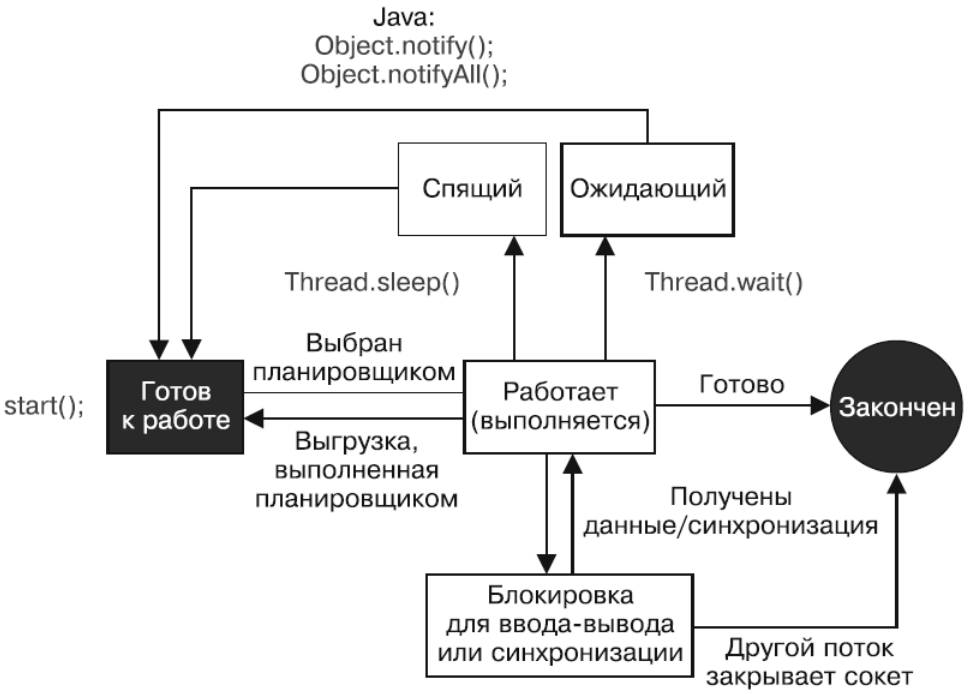
```
PriorThread norm = new PriorThread("Norm");

min.setPriority(Thread.MIN_PRIORITY);
```

Потоки могут быть объединены в группы потоков.

```
ThreadGroup tg = new ThreadGroup("Группа потоков 1");
Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные в группу, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод **getThreadGroup()**.

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



Потоки-демоны

Демоном называется поток, предоставляющий некоторый сервис, работая в фоновом режиме во время выполнения программы, но при этом не является 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);
 }
}
```

СОГЛАСОВАНИЕ ПОТОКОВ

Многопоточный режим работы открывает новые возможности для программистов, однако за 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.

АТОМАРНОСТЬ ОПЕРАЦИЙ

Часть проблем 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 (если судить на уровне абстракции данного языка программирования). Действительно, на уровне языка, реализующего поддержку оператора инкремента, мы не имеем возможность разбить его на более мелкие. Но с точки зрения процессора это несколько операций. Прочитать данные из памяти, прибавить единицу к прочитанному значению, сохранить новое значение обратно. Проблема заключается в том, что когда несколько потоков выполнения пытаются инкрементировать переменную, возможен такой момент, при котором несколько потоков одновременно прочитают одно и тоже 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-rite) 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.

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());
                }     }
```

ПРОБЛЕМА ВИДИМОСТИ

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, а не из всевозможного кеша. 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; } }
```

СИХРОНИЗАЦИЯ

Если несколько задач одновременно попытаются изменить некоторую общую область данных, конечное значение данных при этом будет зависеть от того, какая задача обратится к этой области первой, возникнет ситуация, которую называют состоянием "гонок" (race condition). В случае, когда несколько задач попытаются обновить один и тот же ресурс данных, такое состояние "гонок" называют "гонкой" данных (data race). Гонки за данными могут приводить к конфликтам двух типов: конфликт "чтение-запись", конфликт "запись-запись". Для борьбы с гонками за данными обычно рекомендуется использовать локальные по отношению к потоку, а не разделяемые переменных; управление доступом к разделяемым переменным с помощью различных средств синхронизации (они могут быть реализованы с помощью семафоров, критических секций, взаимных блокировок - мьютексов). 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. Когда двум или более потокам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из потоков. Java для такой синхронизации предоставляет встроенную в язык программирования поддержку.

Системная синхронизация может быть выполнена с использованием:

- механизма взаимодействия между потоками через методы wait(), notify(), notifyall(). Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait, предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток "просыпается" и продолжает свое выполнение;

- с использованием метода join(), который блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания;

- с использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия (классы Lock, семафор Semaphore и т.д). Концепция данного подхода заключается в использовании атомарных операций и переменных.

У каждого Java-объекта есть связанный с ним неявный монитор, а для того чтобы войти в него, надо 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. Под монитором понимается некая управляющая конструкция, обеспечивающая монопольный доступ к объекту. 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. При этом методы, критичные к атомарности операций с данными (обычно - требующие согласованное изменение нескольких полей данных), объявляются как синхронизованные с помощью модификатора **synchronized**.

Блокировка на уровне объекта

Это механизм синхронизации не статического метода или не статического блока кода, такой, что только один поток сможет выполнить данный блок или метод на данном экземпляре класса.

Пример:

```
public class DemoClass{
 public synchronized void demoMethod(){}
}
```

или

```
public class DemoClass{
public void demoMethod(){
 synchronized (this) {
//other thread safe code
} } }
```

или

```
public class DemoClass{
 private final Object lock = new Object();
 public void demoMethod(){
 synchronized (lock)
{
```

```
//other thread safe code
} } }
```

Можно синхронизировать вызов метода целиком или только потокобезопасное подмножество этого метода.

```
public class SynchronizationExample {
   private int i;

   public synchronized int synchronizedMethodGet() {
      return i;
   }

   public int synchronizedBlockGet() {
      synchronized( this ) {
         return i;       }
   }
}
```

Блокировка на уровне класса

Предотвращает возможность нескольким потокам войти в синхронизированный блок во время выполнения в любом из доступных экземпляров класса. Это означает, что если во время выполнения программы имеется 100 экземпляров класса DemoClass, то только один поток в это время сможет выполнить demoMethod() в любом из случаев, и все другие случаи будут заблокированы для других потоков. Это необходимо когда требуется сделать статические данные потокобезопасными.

```
public class DemoClass{
public synchronized static void demoMethod(){}
}
```

или

```
public class DemoClass{
 public void demoMethod(){
```

```
synchronized (DemoClass.class) {
//other thread safe code
} } }
```

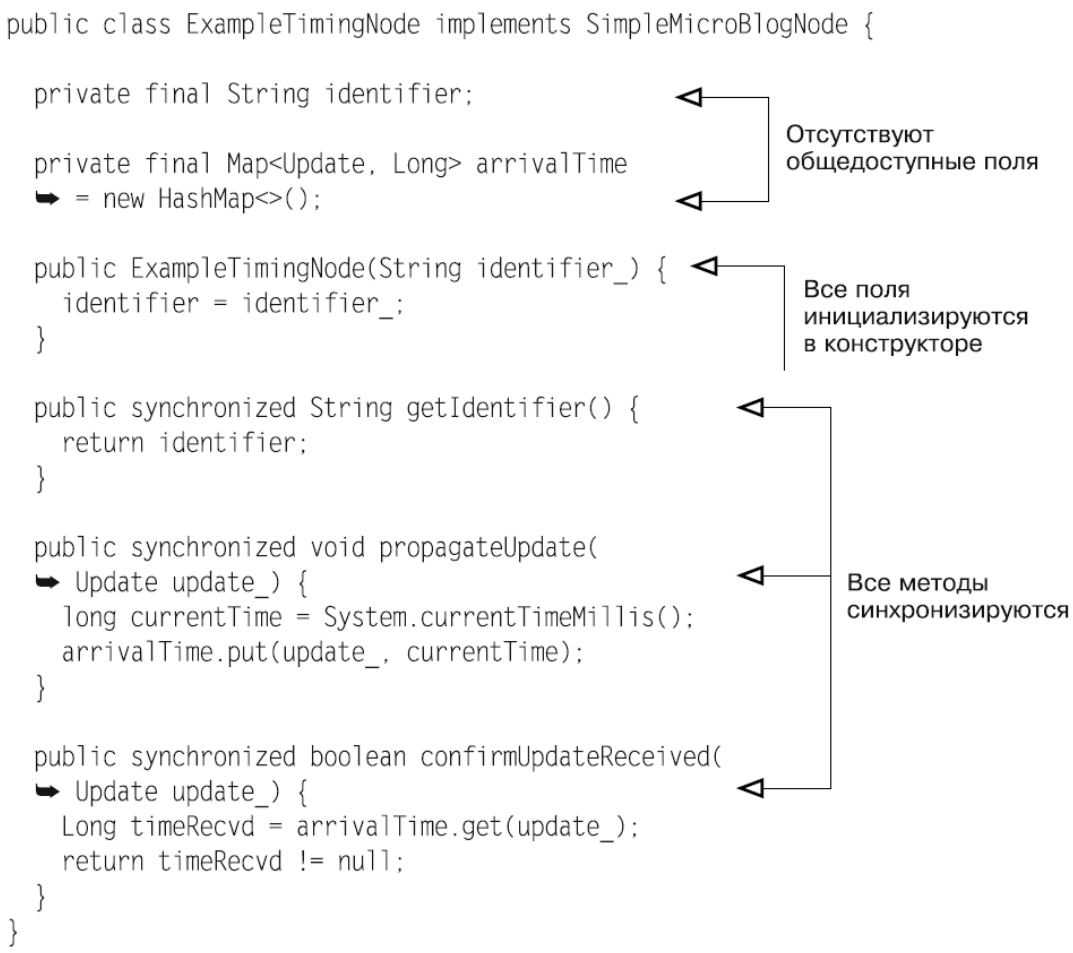
или

```
public class DemoClass{
 private final static Object lock = new Object();
 public void demoMethod(){
 synchronized (lock)  {
//other thread safe code
} } }
```

ПОТОКОБЕЗОПАСНЫЙ КЛАСС

Полностью потокобезопасный синхронизированный класс — это класс, удовлетворяющий следующим условиям:

- 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.



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.

БЕСКОНЕЧНАЯ ОТСРОЧКА (INDEFINITE POSTNEMENT)

Планирование, при котором одна или несколько задач должны ожидать до тех пор, пока не произойдет некоторое событие или не создадутся определенные условия, может оказаться непростым для реализации. Во-первых, ожидаемое событие или условие должно отличаться регулярностью. Во-вторых, между задачами следует наладить связи. Если одна или несколько задач ожидают сеанса связи до своего выполнения, то в случае, если

ожидаемый сеанс связи не состоится, состоится слишком поздно или не полностью, эти задачи могут так никогда и не выполниться. И точно так же, если ожидаемое событие или условие, которое должно наступить, но в действительности не происходит, то приостановленные задачи будут вечно находиться в состоянии ожидания. Если приостановим одну или несколько задач до наступления события (или условия), которое никогда не произойдет, возникнет ситуация, называемая *бесконечной отсрочкой* (indefinite postponement).

В качестве примера может быть рассмотрен 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, реализованный 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 приме 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);}

}
```

ВЗАИМНАЯ БЛОКИРОВКА

Потоки способны перейти в блокированное состояние, а объекты могут обладать синхронизированными методами, которые запрещают использование объекта до тех пор, пока не будет снята блокировка. Возможна ситуация, в которой 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, и никто не может двинуться первым. Такая ситуация называется взаимной блокировкой *(deadlock)*

Пример:

```
public class TestMain {

public static void main(String[] args) {

MyThreadOne t1=new MyThreadOne();

MyThreadTwo t2=new MyThreadTwo();

t1.setThread2(t2);

t2.setThread1(t1);

t1.start();

t2.start();} }

//THREAD NUMBER 1

public class MyThreadOne extends Thread {

Thread t2;

public MyThreadOne() {
```

```
System.out.println(«MyThreadOne create»);
}
public void run() {
System.out.println(«MyThreadOne start»);
try {
sleep(1000);
} catch (Exception e) {
}
try {
System.out.println(«MyThreadOne waiting MyThreadTwo finish…»);
t2.join();
} catch (Exception e) {e.printStackTrace()}
System.out.println(«MyThreadOne finished»);}
public void setThread2(Thread t) {
this.t2 = t;
}}
//THREAD NUMBER 2
public class MyThreadTwo extends Thread {
Thread t1;
public MyThreadTwo() {
System.out.println(«MyThreadTwo create»);
}
public void run() {
System.out.println(«MyThreadTwo start»);
```

```
try {
System.out.println(«MyThreadTwo waiting MyThreadOne finish…»);
t1.join();
} catch (Exception e) {
}
try {
sleep(1000);
} catch (Exception e) {
e.printStackTrace();}
System.out.println(«MyThreadTwo finished»);
}
public void setThread1(Thread t) {
this.t1 = t;}}
```

ТИПОВЫЕ МОДЕЛИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Существует немало параллельных программных приложений, однако в них используется лишь небольшое число моделей решений, или парадигм. В частности, существует пять основных парадигм:

1) итеративный параллелизм;

2) рекурсивный параллелизм;

3) “производители и потребители” (конвейеры);

4) “клиенты и серверы”;

5) взаимодействующие равные.

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. каждый 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).

Производители и потребители – это взаимодействующие процессы. Они часто организуются в конвейер, через который проходит информация. Каждый процесс конвейера является фильтром, который потребляет выход своего предшественника и производит входные данные для своего последователя. Фильтры встречаются на уровне приложений (оболочки) в операционных системах типа ОС Unix, внутри самих операционных систем, внутри прикладных программ, если один процесс производит выходные данные, которые потребляет (читает) другой процесс.

Клиенты и серверы – наиболее распространенная модель взаимодействия в распределенных системах, от локальных до глобальных сетей. Клиентский процесс запрашивает сервис и ждет ответа. Сервер ожидает запросов от клиентов, а затем действует в соответствии с этими запросами. Сервер может быть реализован как одиночный процесс, который не может обрабатывать одновременно несколько клиентских запросов, или при необходимости параллельного обслуживания запросов как многопоточная программа.

Взаимодействующие равные – парадигма взаимодействия, встречающаяся в распределенных программах, в которых несколько процессов для решения задачи выполняют один и тот же код и обмениваются сообщениями. Парадигма «взаимодействующие равные» используется для реализации распределенных параллельных программ, особенно при итеративном параллелизме и децентрализованном принятии решений в распределенных системах.

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.

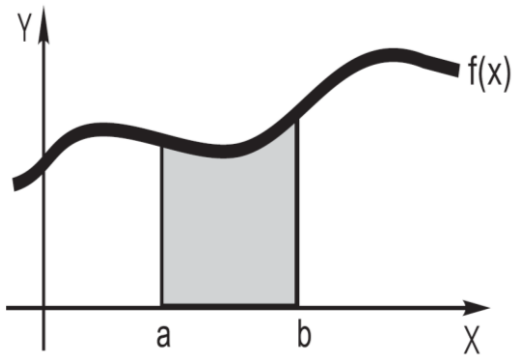
```
public class MatrixMultiThread {
  public static final int NUM_OF_THREADS = 9;
    public static void main(String args[])
   {
      int row;
      int col;
```

```
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];
                              }
}
```

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);   } }
```

ЗАДАЧА «ПРОИЗВОДИТЕЛИ-ПОТРЕБИТЕЛИ». СЕМАФОРЫ.

В задаче "производители-потребители" два процесса совместно используют буфер ограниченного размера. Один из процессов помещает в этот буфер данные, а другой, потребитель, считывает их оттуда. Начинает всегда производитель, и, пока он заполняет буфер, потребитель ожидает. Как только буфер наполнен, производитель сигнализирует потребителю, что можно потреблять, а сам переходит в состояние ожидания. Потребитель, окончив, в свою очередь, сигнализирует производителю, что буфер пуст и его следует наполнить, а сам "засыпает". В общем случае, в задаче участвуют m производителей и n потребителей.

Семафоры используются для того, чтоб перед использованием ресурса проверить его доступность. Примером из жизни может служить тележка (общий ресурс) и два работника (потоки java). Один работник, к примеру, наполняет тележку песком. В это время второй работник, который перевозит груз и затем разгружает, не может взять тележку и отвезти ее. В то же время, если второй работник увез тележку, то первый работник не должен ничего наполнять. Ниже приведен пример программы из 3 классов.

1. Главный класс

```
import java.util.concurrent.Semaphore;
public class SemaphoreExample {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(1);
        new Worker(semaphore, "Adder", true).start();
        new Worker(semaphore, "Reducer", false).start();    } }
```

2. Класс тележки

```
public class Cart {
    private static int weight = 0;
    public static void addWeight(){
        weight--;    }
    public static void reduceWeight(){
        weight++;    }
    public static int getWaight(){
        return weight;    }
}
```

3. Класс работника (boolean переменная определяет это работник по загрузке тележки либо по разгрузке)

```
import java.util.concurrent.Semaphore;
```

```
public class Worker extends Thread {
    private Semaphore semaphore;
    private String workerName;
    private boolean isAdder;

    public Worker(Semaphore semaphore, String workerName, boolean isAdder) {
        this.semaphore = semaphore;
        this.workerName = workerName;
        this.isAdder = isAdder;    }
    @Override
    public void run() {
        System.out.println(workerName + " started working...");
        try {
            System.out.println(workerName + " waiting for cart...");
            semaphore.acquire();
            System.out.println(workerName + " got access to cart...");
            for (int i = 0 ; i < 10 ; i++) {
                if (isAdder)
                    Cart.reduceWeight();
                else
                    Cart.addWeight();

                System.out.println(workerName + " changed weight to: " + Cart.getWaight());
                Thread.sleep(10L);
            }
            semaphore.release();
            System.out.println(workerName + " finished working with cart...");
        } catch (Exception e) {    e.printStackTrace(System.err);        }
    }
}
```

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 к 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.

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 - полезный инструмент для реализации паттерна производитель-потребитель (producer-consumer), потребитель блокируется на получение элемента из очереди пока производитель не добавит его туда, тем самым синхронизируя их выполнение.

ТЕХНОЛОГИЯ Fork/Join

Перспективы аппаратных тенденций таковы, что закон Мура будет выражаться не в увеличении тактовой частоты, а в росте количестве ядер на одной микросхеме. Модель *fork-join framework,* реализованная в языке Java, основывается на модели программирования "разветвление-объединение" (fork-join). Работа программы начинается

с одного потока. Когда программисту требуется добавить в программу параллелизм, выполняется разветвление на несколько потоков, чтобы создать группу потоков. Эти потоки выполняются параллельно в рамках фрагмента кода, который называется параллельным участком. 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).

Ниже приведен пример перемножения матриц, с использованием *fork-join framework.* (источник http://www.java2s.com/Book/Java/Thread-Conncurrent/ForkJoin_Framework.htm)

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Main{
   public static void main(String[] args) {
      Matrix a = new Matrix(2, 3);
      a.setValue(0, 0, 1); // | 1 2 3 |
      a.setValue(0, 1, 2); // | 4 5 6 |
      a.setValue(0, 2, 3);
      a.setValue(1, 0, 4);
      a.setValue(1, 1, 5);
      a.setValue(1, 2, 6);
      Matrix b = new Matrix(3, 2);
      b.setValue(0, 0, 7); // | 7 1 |
      b.setValue(1, 0, 8); // | 8 2 |
      b.setValue(2, 0, 9); // | 9 3 |
      b.setValue(0, 1, 1);
      b.setValue(1, 1, 2);
      b.setValue(2, 1, 3);
      Matrix c = new Matrix(2, 2);
      ForkJoinPool pool = new ForkJoinPool();
      pool.invoke(new Calc(a, b, c));    } }
class Calc extends RecursiveAction {
```

```
    private Matrix a, b, c;
    private int row;
    Calc(Matrix a, Matrix b, Matrix c) {
        this(a, b, c, -1);    }
    Calc(Matrix a, Matrix b, Matrix c, int row) {
        if (a.getCols() != b.getRows()) {
            throw new IllegalArgumentException("rows/columns mismatch");        }
        this.a = a;
        this.b = b;
        this.c = c;
        this.row = row;    }
    @Override
    public void compute() {
        if (row == -1) {
            List<Calc> tasks = new ArrayList<>();
            for (int row = 0; row < a.getRows(); row++) {
                tasks.add(new Calc(a, b, c, row));}
            invokeAll(tasks);        }
         else {
            multiplyRowByColumn(a, b, c, row);        }    }

    void multiplyRowByColumn(Matrix a, Matrix b, Matrix c, int row) {
        for (int j = 0; j < b.getCols(); j++) {
            for (int k = 0; k < a.getCols(); k++) {
              c.setValue(row, j, (int)(c.getValue(row, j) +  a.getValue(row, k)* b.getValue(k, j)));
            }        }    }    }

class Matrix {
 private int[][] doubleArray;
  Matrix(int nrows, int ncols)
 {    doubleArray = new int[nrows][ncols];  }

  int getCols() {    return doubleArray[0].length;  }
  int getRows() {    return doubleArray.length;  }
  double getValue(int row, int col) {    return doubleArray[row][col];  }
```

```
  void setValue(int row, int col, int value) {    doubleArray[row][col] = value;  }
}
```

ТЕСТИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

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 параллельных программах трудно воспроизвести точный контекст параллельных 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.

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.

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. Потенциальные ошибки, связанные с параллелизмом данных и задач
3. Атомарность операций, состояние гонки, проблема видимости
4. Процессы и потоки
5. Потокобезопасные классы.
6. Блокировки. Бесконечное ожидание и взаимная блокировка
7. Парадигмы параллельного программирования.

Литература

1. Гергель В.П. Теория и практика параллельных вычислений. – М.:Интернет-Университет, БИНОМ. Лаборатория знаний, 2007.
2. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления — СПб: БХВ-Петербург, 2002. — 608 с.
3. Немнюгин С., Стесик О. - Параллельное программирование для многопроцессорных вычислительных систем. - СПб. БХВ-Петербург, 2002. – 400с.
4. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М:ДМК-Пресс. 2010, -232с.
5. B. Eckel. Thinking in Java (4th Edition). Prentice Hall, 2006.-1150p.
6. B. Goetiz, T. Peierls. Java concurrency in practice. Addison-Wesley Professional, 2006, -384p.