ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Федеральное государственное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ЧЕРДЫНЦЕВА М.И.

ПРАКТИКУМ ПО КУРСУ «МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА JAVA»

Ростов-на-Дону
2008

СОДЕРЖАНИЕ

Введение

Начиная с самых первых версий, виртуальные машины Java поддерживают многопоточность, то есть создание нескольких потоков исполнения одновременно. Многопоточные программы развивают идею многозадачности на более низком уровне: отдельные программы способны выполнять «параллельно» несколько задач. Многопоточность имеет очень большое практическое значение. В настоящее время все больше возрастает необходимость разработка программ использующих несколько потоков.

Начиная с версии J2SE JDK 5.0, в языке появились новые возможности, объединенные в пакет java.util.concurrent. Разработчиками были предложены классы и интерфейсы, реализующие те возможности, которых давно не хватало в языке Java.

В учебном пособии приведены описания лабораторных работ по курсу «Многопоточное программирование на Java». Каждая лабораторная работа содержит примеры разобранных решений задач по соответствующим разделам курса, наборы упражнений к каждому заданию, контрольные вопросы для проверки усвоения материала. В лабораторных работах рассмотрены все основные механизмы управления потоками в Java, включая новые возможности, появившиеся в версии J2SE JDK 5.0.

Учебное пособие разработано с учетом кредитно-модульной системы обучения. Подробное описание хода решения задач и полные коды программ позволят использовать учебное пособие не только при проведении практических и лабораторных занятий по курсу, но и для самостоятельного изучения курса студентами.

Навыки и знания, необходимые для выполнения лабораторного практикума.

Студенты, выполняющие лабораторный практикум должны быть знакомы с курсами «Технологии Java», «Многопоточное и распределенное программирование», «Сетевое программирование», знать основные принципы объектно–ориентированного подхода в программировании.

Студенты должны уметь работать в одной из визуальных сред программирования на Java – NetBeans или Eclipse, или уметь использовать инструменты командной строки пакета разработчика – Java Development Kit.

Полученные в результате выполнения практикума знания и навыки могут быть использованы студентами при изучении дисциплин, связанных с разработкой многопоточных и распределенных приложений, в курсах «Проектирование информационных систем», «Технологии баз данных», «Технологии клиент–сервер», при выполнении курсовых и дипломных работ.

График выполнения и сдачи практических работ

Учебное пособие содержит описание 16 лабораторных работ, соответствующих 16 лабораторным занятиям по курсу «Многопоточное программирование на Java», каждое лабораторное занятие рассчитано на 2 часа аудиторной работы и 2 часа самостоятельной работы. Студенты, самостоятельно выполняющие лабораторные работы могут сдавать лабораторные работы и индивидуальные задания, ориентируясь на контрольные сроки отчетности по модулям лабораторных работ:

№ модуля	Лабораторные работы №№	Индивидуальное задание №	Последний срок сдачи заданий	
			Осенний семестр	Весенний семестр
1	1-4		3-я неделя сентября	1-я неделя марта
2	5-8	1	Последняя неделя октября	Последняя неделя марта
3	9-10		2-я неделя ноября	2-я неделя апреля
4	11-15		2-я неделя декабря	2-я неделя мая
6	16		3-я неделя декабря	3-я неделя мая
		2	4-я неделя декабря	4-я неделя декабря

МОДУЛЬ 1

Способы организации потоков, управление потоками

Лабораторная работа 1

Средства для организации потоков в языке Java

Цель работы: Изучить возможность использования класса Thread и интерфейса Runnable. Познакомиться с методами, позволяющими управлять выполнением потоков. Научиться создавать собственные классы–потоки двумя способами и запускать их на выполнение.

Задание 1. Выполните приложение *CurrentThreadDemo.*

```
class CurrentThreadDemo {
   public static void main(String args[]) {
      Thread t = Thread.currentThread();
      System.out. println("current thread: " + t);
      t.setName("My Thread");
      System.out. println("current thread: " + t);
      try {
           for (int n = 5; n > 0; n--) {
                System.out.println(" " + n);
                Thread.sleep(1000);
      } }
      catch (InterruptedException e) {
      System.out.println("interrupted");
      }
} }
```

Замечание. Если при работе не используется никакая визуальная среда разработки, то для компиляции приложения можно воспользоваться командной строкой, в которой необходимо выполнить команду

```
>javac CurrentThreadDemo.java
```

Для корректного выполнения данной команды необходимо, чтобы каталог, в котором расположен файл CurrentThreadDemo.java был текущим (рабочим), а также, чтобы путь к программе–компилятору javac.exe был

включен в состав переменной окружения PATH. В этом случае, если программа не содержит синтаксических ошибок, в текущем каталоге появится файл, содержащий байт–код приложения CurrentThreadDemo.class

Выполнение приложения через командное окно осуществляется командой

```
>java CurrentThreadDemo
```

Задание 2. Рассмотрите два способа создания потоков на примере приложений ThreadDemo и SimpleThread.

Приложение SimpleThread.java

```
public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
}
```

Приложение ThreadDemo.java

```
class ThreadDemo implements Runnable {
Thread t;
ThreadDemo() {
```

```
            Thread ct = Thread.currentThread();
             System.out.println("currentThread: " + ct);
             t = new Thread(this, "Demo Thread");
             System.out.println("Thread created: " + t);
             t.start();
}
public void run() {
try {
              for (int i = 5; i > 0; i--) {
               System.out.println("child thread " +  i);
              Thread.sleep(500);
      }
}  catch (InterruptedException e) {
              System.out.println("child interrupted");
}
System.out.println("exiting child thread");
}
public static void main(String args[]) {
new ThreadDemo();
try {
  for (int i=5; i>0; i--) {
        System.out.println("Main thread " +i);
        Thread.sleep(1000);
       }
}catch(InterruptedException e) {
              System.out.println("Main interrupted");
}
System.out.println("exiting Main thread");
}
}
```

Задание 3. Научитесь получать информацию о потоке и управлять выполнением потока, используя методы класса Thread, приведенные в таблице 1.

Задание 4. Воспользуйтесь справкой Java API Specification, чтобы ознакомиться с другими методами класса Thread.

Замечание. Обратите внимание, что ряд методов отмечен как ***Deprecated*** – устаревшие, не рекомендуемые. Не используйте эти методы при управлении потоками, они могут приводить к ошибкам и сбоям в многопоточных программах.

Таблица 1. Методы класса Thread

Метод	Описание
`long getId()`	Возвращает идентификатор потока
`String getName()`	Возвращает имя потока
`int getPriority()`	Возвращает приоритет потока
`boolean isAlive()`	Проверяет, является ли поток работоспособным
`boolean isDeamon()`	Проверяет, является ли поток демоном
`boolean isInterrupted()`	Проверяет, был ли поток прерван
`void setName(String name)`	Изменяет имя потока
`void setPriority (int newPriority)`	Изменяет приоритет потока
`static void sleep(long millis)`	Заставляет текущий выполняющийся поток заснуть(немедленно прекратить выполнение) на указанное число миллисекунд
`static void yield()`	Заставляет текущий поток прекратить выполнение, чтобы дать возможность выполниться потокам, ожидающим ресурсы процессора
`void join()`	Ожидает когда данный поток завершит выполнение
`void setDaemon(boolean on)`	Устанавливает, будет ли данный поток выполняться как «демон» или как управляемый поток. Назначение потока «демоном» выполняется до старта потока

Вопросы для повторения.

1. Какие способы создания потока есть в Java.
2. В каких состояниях может находиться поток. В результате каких действий поток может переходить из одного состояния в другое.

3. Чем отличается создание потока через реализацию интерфейса Runnable от наследования класса Thread.
4. Можно ли предсказать последовательность выполнения нескольких одновременно запущенных потоков. От чего она зависит.
5. С помощью каких методов можно заставить поток добровольно прервать выполнение.
6. При каких условиях еще может быть осуществлено переключение выполнения нескольких параллельных потоков.
7. Какие из следующих методов определены в классе `Thread`:
   a) `start()`
   b) `wait()`
   c) `notify()`
   d) `sleep()`
   e) `terminate()`
8. Какие методы объявлены в интерфейсе `Runnable`:
   a) `sleep()`
   b) `wait()`
   c) `run()`
   d) `start()`
9. Какой из методов `run()` или `start()` запускает поток на выполнение? Для чего нужен другой метод?
10. Что произойдет, если для потока явно вызвать метод `run()`?
11. Почему метод `sleep()` должен вызываться в блоке, перехватывающем исключительную ситуацию?
12. Может ли один из потоков заставить другой выполняющийся поток заснуть?
13. На что влияет приоритет потока?
14. Почему методы `sleep()` и `yield()` являются статическими? Для какого потока они выполняются?

Лабораторная работа 2

Согласование работы дочерних потоков с главным. Группы потоков

Цель работы: Разобраться, как должны быть согласованы время выполнения приложения (апплета) и время существования потоков данного приложения. Познакомиться с понятием потока–демона в приложении на Java и его отличием от демонов–процессов. Научиться создавать группы потоков и управлять ими.

Рекомендации к выполнению работы: Обратите внимание на то, что главный поток приложения должен всегда завершаться после завершения всех дочерних потоков. Чтобы обеспечить корректность исполнения необходимо, или заставлять главный поток ожидать завершение всех дочерних потоков (используйте метод `join()`), или назначать дочерние потоки демонами, чтобы они завершились принудительно, когда завершится основной поток приложения (используйте метод `setDaemon()`).

Задание 1. Рассмотрите пример приложения, в котором используется примитивный барьер, позволяющий главному потоку приложения дождаться завершения выполнения всех дочерних потоков.

Задание 2. В следующем приложении показано, как создается поток–демон. Попробуйте изменить приложение так, чтобы одновременно запускалось несколько потоков–демонов.

Задание 3. Рассмотрите следующий пример приложения. В этом примере создаются потоки `ThreadTest`. Каждый поток имеет некоторое стартовое значение, передаваемое ему при создании. В методе `run()` это значение последовательно уменьшается. При достижении половины от начальной величины порождается новый поток с вдвое меньшим начальным значением. По исчерпании счетчика поток останавливается. Метод `main()` порождает первый поток со стартовым значением 16. В ходе программы

будут дополнительно порождены потоки со значениями 8, 4, 2. Все дочерние потоки включаются в одну группу. За работой группы потоков наблюдает поток–демон `DaemonDemo`. Этот поток регулярно получает список всех существующих потоков `ThreadTest` и распечатывает их имена для удобства наблюдения.

```
public class ThreadTest implements Runnable {
// используем группу для потоков ThreadTest
   public final static ThreadGroup GROUP =
               new ThreadGroup("Daemon demo");
// Стартовое значение, указывается при создании объекта
   private int start;
   public ThreadTest(int s) {
      start = (s%2==0)? s: s+1;
      new Thread(GROUP, this, "Thread "+ start).start();
   }
public void run() {
      for (int i=start; i>0; i--) {
         try {
            Thread.sleep(300);
         } catch (InterruptedException e) {}
         // По достижении середины порождаем
         // новый поток с половинным начальным значением
         if (start>2 && i==start/2)
       {
           new ThreadTest(i);
         }
      }
   }
public static void main(String s[]) {
   new ThreadTest(16);
   new DaemonDemo();
   }}
public class DaemonDemo extends Thread {
   public DaemonDemo() {
```

```
        super("Daemon demo thread");
        setDaemon(true);
        start();
    }
         public void run() {
        Thread threads[]=new Thread[10];
        while (true) {
            // Получаем набор всех потоков из тестовой группы
            int count=ThreadTest.GROUP.activeCount();
            if (threads.length<count)
                  threads = new Thread[count+10];
            count=ThreadTest.GROUP.enumerate(threads);
            // Распечатываем имя каждого потока
            for (int i=0; i<count; i++) {
               System.out.print(threads[i].getName()+", ");
            }
            System.out.println();
            try {
               Thread.sleep(300);
            } catch (InterruptedException e) {}
        }
    }
}
```

Задание 4. Используя примеры приложений из этой и предыдущей лабораторной работы, создайте свое многопоточное приложение, в котором запускается несколько дочерних потоков. Рассмотрите варианты, когда все потоки одинаковые и когда в приложении запускаются несколько разных потоков.

Вопросы для повторения.

1. Что обеспечивает использование метода `join()`?
2. Почему нельзя допустить, чтобы после основной поток приложения завершился раньше, чем дочерние потоки?
3. Что вкладывается в понятие потока–демона в Java?

4. Можно ли уже запущенный поток сделать демоном?

5. Обязательно ли, чтобы поток – демон выполнялся до тех пор, пока не завершится выполнение основного потока приложения?

6. Что будет выдано в результате компиляции и выполнения следующего кода:

```
class T1 implements Runnable{
public void run(){
System.out.print("T1 ");
}}
class T2 extends Thread{
public void run(){
System.out.print("T2 ");
} }
public class Quest3 {
 public static void main(String[] args) {
      T1 t1= new T1();
T2 t2 = new T2(t1);
t1.start();
t2.start();
 }
}
```

1) T2 T1

2) Ошибка компиляции: метод start() не определен в классе T1

3) Ошибка компиляции: в классе T2 не определен конструктор, принимающий в качестве параметра объект Thread

4) Ничего из перечисленного

5) T1 T2

Лабораторная работа 3

Потоки в приложениях с графическим интерфейсом

Цель работы: Научиться использовать потоки в приложениях с графическим интерфейсом. Научиться управлять элементами пользовательского интерфейса из нескольких потоков. Познакомиться с потоками управления событиями

Задание 1. На примере приложения `PingPong` разберитесь, как можно использовать многопоточность при организации программ с графическим интерфейсом.

```
import java.awt.*;
import java.awt.event.*;
public class PingPong
               extends Frame
               implements Runnable, KeyListener {
     int x=100, y=120;
     int stepx=5, stepy=3;
     int racket=40;
     int rY=40;
     int pause=50;
     int level=1, count=0;
     boolean flag=true;
     public PingPong() {
          super("Level 1");
          addKeyListener(this);
          setSize(400,200);
          setVisible(true);
          Thread myThr = new Thread(this);
          myThr.start();
     }
     public void run() {
          while(flag) {
               try{
                    x+=stepx;
```

```
                    y+=stepy;
                    if ((x<50)&(rY<y)&(rY+racket>y)){
                         stepx=-stepx;
                         count++;
                    }
                    if (count>5) {
                         pause=(int)(pause-pause/100);
                         racket=(int)(racket-racket/100);
                         level++;
                         setTitle("Level "+
                              new Integer(level).toString());
                         count =0;
                    }
                    if (x<0) {
                         flag=false;
                         dispose();
                         System.exit(0);
                    }
                    if (x>getSize().width-20) stepx=-stepx;
                    if ((y<20)|(y>getSize().height-20))
                         stepy=-stepy;
                    repaint();
                    Thread.sleep(pause);
               } catch(InterruptedException e){}
          }
}
public  void paint (Graphics g)
{
      g.fillOval(x,y,20,20);
      g.fillRect(40,rY,10,racket);
}
 public void keyPressed(KeyEvent e) {
       if ((e.getKeyCode()==KeyEvent.VK_UP) & (rY>20)) rY-=10;
       if ((e.getKeyCode()==KeyEvent.VK_DOWN) &
           (rY<getSize().height-racket))
               rY+=10;
```

```
        repaint();
 }
  public void keyReleased(KeyEvent e){}
  public void keyTyped(KeyEvent e){}
 public static void main (String [] args) {
   new PingPong();
}
}
```

Вопросы для повторения.

1. Почему визуальные компоненты могут быть небезопасными для потоков?

2. Какие потоки всегда существуют в приложениях, имеющих оконный интерфейс?

3. В чем заключается правило единственного потока?

4. Что будет выдано в результате компиляции и запуска приложения:

```
class Quest implements Runnable{
 int i=0;
      public int run(){
       System.out.println("i="+ ++i);
       return i;
  }}
public class Quest1 {
 public static void main(String[] args) {
      Quest ob = new Quest();
      ob.run();
}}
```

1) i=1

2) i=0

3) Ошибка компиляции: неправильно определен метод run()

4) Ошибка времени выполнения: поток запускается методом start()

5) Ошибка компиляции: Объект ob нужно создавать, используя конструктор класса Thread.

Лабораторная работа 4

Потоки в апплетах

Цель работы: Научиться использовать потоки в апплетах. Научиться согласовывать работу апплета и поток событий от управляющих компонент, расположенных в окне апплета.

Задание 1. Рассмотрите пример апплета, в котором используется поток для организации баннера.

```
import java.awt.*;
import java.applet.*;

public class SimpleBanner extends Applet implements Runnable{
String msg=" --a simple moving banner-- ";
Thread t=null;
String txt="";
int mouseX=0, mouseY=0;
public boolean mouseDown(Event evtObj,int x, int y){
     mouseX=x;
     mouseY=y;
     txt="Down";
     repaint();
     return true;
}
public void init(){
     setBackground(Color.cyan);
     setForeground(Color.red);
     t=new Thread(this);
     t.start();
}
 public void start(){
}
public void run(){
     char ch;
     for(;;){
```

```
            try{
                  repaint();
                  Thread.sleep(250);
                  ch = msg.charAt(0);
                  msg=msg.substring(1,msg.length());
                  msg+=ch;
            } catch(InterruptedException e){}
      }
}
public void destroy(){
 if (t !=null){
  t.interrupt();
  t=null;
}
}
```

Рубежный контроль модуля предполагает выполнение студентами индивидуальных заданий.

Задания для индивидуальной работы

1. Создать апплет (или приложение), в окне которого выводится движущаяся строка. Программа должна содержать меню, позволяющее осуществлять выбор направления движения (справа налево или наоборот) и вызывать диалоговое окно для ввода пользователем текста отображаемой строки и скорости перемещения строки (задержка в миллисекундах).

2. Создать апплет (или приложение), в окне которого последовательно буква за буквой выводится текст (например, пусть введена строка "АБВГ....ЭЮЯ", тогда последовательность отображения ее символов следующая: "А", "АБ", "АБВ" и т.д.). Программа должна содержать меню, которое позволяет осуществлять выбор шрифта для отображения текста и вызывать диалоговое окно для ввода пользователем текста отображаемой строки и скорости вывода символов (задержка в миллисекундах).

3. Создать апплет (или приложение), в окне которого отображается строка, бегущая изнутри (например, пусть введена строка "АБВГ....ЭЮЯ", тогда последовательность отображения ее символов следующая: "АЯ", "АБЮЯ", "АБВЭЮЯ" и т.д.). Программа должна содержать меню, которое позволяет осуществлять выбор шрифта для отображения текста и вызывать диалоговое окно для ввода пользователем текста отображаемой строки и скорости вывода символов (задержка в миллисекундах).

4. Создать апплет (или приложение), в окне которого отображается строка, бегущая по буквам (например, пусть введена строка "АБВГ....ЭЮЯ", тогда последовательность отображения ее символов следующая: "..............<-А", "А.............<- Б", "АБ...........<- В" и т.д.). Программа должна содержать меню, которое позволяет осуществлять выбор шрифта для отображения текста и его цвета, а также вызывать диалоговое окно для ввода пользователем текста отображаемой строки и скорости вывода символов (задержка в миллисекундах).

5. Создать апплет (или приложение), в окне которого отображается строка. Символы строки отображаются в случайном порядке случайным цветом. После вывода всех символов окно очищается и вывод начинается вновь. Программа должна содержать меню, которое позволяет осуществлять выбор шрифта для отображения текста, а также вызывать диалоговое окно для ввода пользователем текста отображаемой строки и скорости вывода символов (задержка в миллисекундах).

По завершению изучения модуля студент должен уметь:

1. Создавать собственные потоки, используя одну из изученных технологий.
2. Управлять выполнением потоков, используя методы класса Thread.
3. Понимать в каких случаях при выполнении многопоточных приложений могут возникать исключительные ситуации, уметь правильно их обрабатывать.
4. Понимать специфику отладки многопоточных приложений.

МОДУЛЬ 2

Гонки потоков, средства синхронизации

Лабораторная работа 5

Возникновение гонки потоков

Цель работы: Рассмотреть к каким последствиям может приводить гонка потоков. Изучить как на ситуацию гонки могут влиять такие факторы как конвейерное выполнение команд и модель памяти виртуальной машины Java.

Задание 1. На примере приложения изучите возникновение ситуации «гонки» потоков. Проанализируйте условия возникновения гонки и возможные планы выполнения потоков.

```
class OutThread extends Thread{
  private String msg;
  OutThread(String s, String name) {
    super(name); msg=s;
     }
   public void run() {
    for(int i=0; i<5;i++) {
      //try{
      //    Thread.sleep(5);
      //} catch(InterruptedException ie){}
     System.out.print(msg+" ");
     }
     System.out.print("End of "+getName());
   }
     public static void main(String[] args) {
       new OutThread("FIRST","Thread 1").start();
       new OutThread("Second","Thread 2").start();
       new OutThread("Hip","Thread 3").start();
       new OutThread("hop","Thread 4").start();
       System.out.println("END");
}}
```

Упражнения. Выполните несколько экспериментов с приведенным приложением:

a) Уберите комментарии внутри цикла в методе run(). Проверьте, как будет влиять на результат выполнения время засыпания потоков.

b) Увеличьте число повторений цикла. Какая ошибка возникает, если каждый из потоков будет работать достаточно долго. Внесите изменения в приложение, чтобы избежать этой ошибки.

Задание 2. Выполните следующее приложение. Изменяя время засыпания потоков, или изменяя параметры цикла, добейтесь того, чтобы в результате гонки выдаваемые сообщения были бессмысленны.

```
class Callme {
void call(String msg) {
System.out.print("[" + msg);
//try {
//Thread.sleep(1);
//}
//catch(Exception e) {}
for ( int i=0; i<10000; i++)
for ( int j=0; j<2350; j++);
System.out.println("]");
} }
class Caller implements Runnable {
String msg;
Callme target;
public Caller(Callme t, String s) {
target = t;
msg = s;
new Thread(this).start();
}
public void run() {
target.call(msg);
} }
class Synch {
```

```
public static void main(String args[]) {
Callme target = new Callme();
new Caller(target, "Hello!!!");
new Caller(target, "Synchronized");
new Caller(target, "World");
}
}
```

Упражнения. Выполните несколько экспериментов с приведенным приложением:

a) Уберите комментарии внутри цикла в методе run(). Проверьте, как будет влиять на результат выполнения время засыпания потоков.

b) Увеличьте число повторений цикла. Какая ошибка возникает, если каждый из потоков будет работать достаточно долго. Внесите изменения в приложение, чтобы избежать этой ошибки.

Вопросы для повторения.

1. К каким последствиям может приводить гонка потоков?
2. Почему для отладки многопоточных программ не всегда можно использовать промежуточную выдачу результатов с помощью метода `System.out.println()`?
3. Как на гонку потоков может повлиять использование метода `sleep()`? Как, не используя его, добиться возникновения гонки потоков?

Лабораторная работа 6

Гонки потоков (продолжение)

Задание 1. Следующее приложение демонстрирует ситуацию гонки, когда два различных потока обращаются к одним и тем же данным. Данные и методы доступа к ним объединены в класс `Test` и являются статическими. Это позволяет работать с ними не создавая объект данного класса. В методе `main()` класса `TestUnsynched` создаются два потока. У одного из потоков метод `run()` многократно вызывает статический метод `one()` класса `Test`. В этом методе происходит увеличение значений статических переменных `i` и `j`. Второй поток в бесконечном цикле опрашивает значение переменных `i` и `j` через метод `two()`. В зависимости от соотношения значений переменных `i` и `j` изменяются значения переменных–счетчиков `k1`, `k2` и `k3`. Поскольку работа метода `run()` второго потока никогда не завершается, этот поток должен быть демоном. Основной поток приложения ждет, когда завершится выполнение первого дочернего потока, после чего использует статический метод `result()` класса `Test`, чтобы выдать результирующие значения счетчиков.

```
class Test {
     static int i=0, j=0;
     static int k1=0,k2=0,k3=0;
     static void one() {
          i++;
          j++;}
     static void two(){
          if (i==j) k1++;
          if (i<j) k2++;
          if (i>j) k3++;
     }
     static void result(){
          System.out.println("=="+k1+" i<j "+k2+" i>j "+k3);}
     }
```

```
public class TestUnsynched {
     public static void main(String[] args){
          Thread t1=new Thread(){
               public void run(){
                    for (int k=0;k<1000000;k++)
                         Test.one();}};
          t1.start();
          Thread t2=new Thread(){
               public void run(){
                    while(true)
                         Test.two();}};
          t2.setDaemon(true);
          t2.start();
          t1.join();
          Test.result();
}
}
```

Упражнения. Выполните следующие эксперименты с приведенным выше приложением:

a) Реальная ситуация гонки возникнет только тогда, когда выполнение потока `t1` будет прервано и произойдет переключение на поток `t2` и наоборот. Количество таких переключений будет тем больше, чем больше повторений будет в цикле

```
for (int k=0;k<1000000;k++)
```

Подберите экспериментальным путем количество повторений цикла, когда будут получаться наиболее интересные результаты.

b) Проверьте, повлияет ли на результат изменение порядка проверок в методе `two()`

```
if (i==j) k1++;
if (i<j) k2++;
if (i>j) k3++;
```

c) Что изменится, если от какой–нибудь из проверок вообще отказаться?

d) Попробуйте сразу после проверки `if (i<j)` выдать значения переменных `i` и `j`:

```
if (i<j) System.out.println("i="+i+" j="+j);
```

Как можно объяснить получаемые результаты?

e) Обратите внимание, что выполнение операций ввода–вывода происходит в отдельном потоке, причем метод `println()` не допускает во время своего выполнения изменения значений переменных `i` и `j`. Какие можно сделать выводы о возможности следить за выполнением многопоточных программ с помощью выдачи промежуточных результатов?

Вопросы для повторения.

1. К каким последствиям может приводить гонка потоков?
2. Опишите каким ситуациям соответствуют три условия, проверяемые в программе из Задания 3:

```
if (i==j) k1++;
if (i<j) k2++;
if (i>j) k3++;
```

3. Почему в рассматриваемом приложении возможно возникновение ситуации, когда значение переменной `i` меньше, чем значение переменной `j`? Почему при выдаче значений этих переменных всегда `i` больше или равно `j`?
4. Можно ли изменить метод `two()` таким образом, чтобы увидеть когда `i` больше `j`?
5. Какие особенности модели памяти виртуальной машины демонстрирует приведенное выше приложение?
6. Почему для отладки многопоточных программ не всегда можно использовать промежуточную выдачу результатов с помощью метода `System.out.println()`?
7. Как на гонку потоков может повлиять использование метода `sleep()`? Как, не используя его, добиться возникновения гонки потоков?

Лабораторная работа 7

Синхронизация

Цель работы: Научиться использовать синхронизованные блоки и синхронизованные методы. Познакомиться с понятием блокировки объекта, механизмом обмена сообщениями между потоками. Рассмотреть положительные и отрицательные стороны механизма блокировок. Рассмотреть возможности классов блокировки, появившихся в пакете `java.util.concurrent.locks`.

Задание 1. В приводимом ниже примере показано, как избежать гонки потоков, сделав метод `run()` синхронизованным. Настройте сначала цикл или замените его вызовом метода `sleep()`, чтобы обеспечить переключение потоков. Убедитесь, что при выбранных условиях выполнения возникает гонка потоков. После этого сделайте метод `run()` синхронизованным. Убедитесь, что теперь ни при каких условиях гонка потоков невозможна.

```
class ThreadsWrite implements Runnable{
  public void run() {
//  чтобы добиться синхронизации измените заголовок метода
//  run() как показано в следующей строке
// synchronized public void run(){
    System.out.print("Hello,");
    for (int i=0; i<10000; i++)
     for (int j=0; j<2000; j++) ;
 // вместо цикла здесь можно использовать sleep()
    System.out.println("World!!!");
 }
 public static void main (String[] args){
   ThreadsWrite tw = new ThreadsWrite();
   new Thread(tw).start();
   new Thread(tw).start();
   new Thread(tw).start();
   new Thread(tw).start();
```

```
    // можно запустить и больше потоков
}}
```

Задание 2. Используя механизм синхронизованных методов, измените все примеры из лабораторной работы 4, чтобы избежать гонки потоков. Проверьте, влияет ли синхронизация на время выполнения приложения.

Замечание. Для того, чтобы определить время выполнения приложения, или отдельной его части воспользуйтесь методом класса `System`, возвращающем текущее системное время в миллисекундах в виде длинного целого числа:

```
long t =System. currentTimeMillis();
```

Задание 3. Рассмотрите простейшую реализацию задачи типа «производитель–потребитель», использующую в качестве совместно используемых данных целую переменную. Поток–производитель помещает в эту переменную очередное значение, поток–потребитель забирает (читает) значение, помещенное в переменную.

```
class Q {
  int n;
  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out. println("Put: " + n);
} }
class Producer implements Runnable {
  Q q;
  int n;
  Producer(Q q, int count) {
        this.q = q;
        n = count;
```

```
            new Thread(this,"Producer").start();
    }
    public void run(){
       int i = 0;
       while (i<n) {
            q.put(i++);
} } }
class Consumer implements Runnable {
    Q q;
    int n;
    Consumer(Q q, int count) {
       this.q = q;
       n = count;
       new Thread(this,"Consumer").start();
    }
    public void run(){
       int i = 0;
       while (i<n) {
            q.get();
            //System.out.println(”Consumer get=”+q.get());
            i++;
       }
} }
class PC {
    public static void main(String args[]) {
    Q q = new Q();
    new Producer(q,10);
    new Consumer(q,10);
} }
```

Упражнения. Выполните следующие эксперименты с приведенным выше приложением:

а) Убедитесь, что средства синхронизации не обеспечивают правильного чередования работы потоков. Почему синхронизации недостаточно для правильной работы потоков в этом случае?

b) Увеличьте количество производителей и потребителей. Чтобы различать их, используйте соответствующие методы класса `Thread`. Вставьте операторы отладочной печати, как это показано в методе `run()` класса `Consumer`.

c) Создайте приложение, в котором потребитель и производитель выполняются не в отдельных потоках, а в рамках основного потока приложения. Чередование работы осуществляется поочередным вызовом соответствующих методов. Сравните время работы этого приложения и приложения, основанного на отдельных потоках для производителя и для потребителя.

d) Придумайте содержательные интерпретации для модели задачи «производитель–потребитель». Объясните, в каких случаях будет требоваться, чтобы производитель и потребитель выполнялись в отдельных потоках.

Задание 4. В следующем примере приложения, реализующего модель «производитель–потребитель» взаимодействие двух потоков организовано через механизм ожидания – уведомления.

```
class Q {
     int n;
     boolean full = false;
     synchronized int get() {
          while (!full)
               try{
                    wait();
               } catch (InterruptedException e) {
                    System.out.println("Interrupt");
                    return;
          }
          System.out.println("Got: " + n);
          full = false;
          notify();
          return n;
```

```
   }
   synchronized void put(int n) {
     while (full)
          try{
               wait();
          } catch (InterruptedException e) {
               System.out.println("Interrupt");
               return;
          }
          this.n = n;
          full = true;
          System.out. println("Put: " + n);
          notify();
} }
class Producer implements Runnable {
    Q q;
    int rounds;
   Producer(Q q, int rd) {
     this.q = q;
     rounds = rd;
     new Thread(this,"Producer").start();
   }
   public void run(){
   for (int i=0; i<rounds; i++){
     q.put(i);
} } }
class Consumer implements Runnable {
     Q q;
     int rounds;
     Consumer(Q q, int rd) {
          this.q = q;
          rounds = rd;
          new Thread(this,"Consumer").start();
     }
     public void run()   {
          for (int i=0; i<rounds; i++){
```

```
                q.get();
        }
} }
class PC_1 {
public static void main(String args[]) {
        if (args.length==0) {
             System.out.println("Put rounds in params");
             System.exit(1);}
        int rounds = Integer.parseInt(args[0],10);
        Q q = new Q();
        new Producer(q,rounds);
        new Consumer(q,rounds);
} }
```

Упражнение. Проведите эксперименты по сопоставлению времени работы многопоточного приложения и однопоточного, аналогичные упражнению с) из Задания 3.

Вопросы для повторения.

1. Что обеспечивает синхронизованный метод?
2. Что такое монитор в языке Java? Как он работает?
3. Объясните, почему проверка условия ожидания должна выполняться в цикле.
4. Когда нужно применять вместо синхронизованных методов синхронизующие блоки?
5. В каком классе определены методы `wait()` и `notify()`?
6. Чем отличаются методы `notify()` и `notifyall()`?
7. Каковы правила использования методов `wait()` и `notify()`?
8. Почему синхронизация может замедлять выполнение программы?
9. Когда обязательно следует прибегать к средствам синхронизации?

5. Выполнение какого из указанных действий приведет к тому, что поток переходит в состояние «пассивный»?

1) вызов метода stop()
2) окончание выполнения метода run()
3) вызов метода notufyAll()
4) .вызов метода sleep() без параметра
5) вызов метода wait()

6. Что будет выдано в результате компиляции и запуска следующего кода:

```
class Quest5 extends Thread     {
Quest5 () { }
Quest5 (Runnable r) { super(r); }
        public void run() {
            System.out.print("YES ");
        }
            public static void main(String[] args)        {
          Runnable r = new Quest5();  //1
Quest5 t = new Quest5(r);     //2
            t.run();
        }   }
```

1) Ошибка компиляции в строке номер 2.
2) YES
3) YES YES
4) Ошибка компиляции в строке номер 1
5) Ничего из перечисленного

Лабораторная работа 8

Средства синхронизации в пакете java.util.concurrent.locks

Цель работы: Рассмотреть возможности классов блокировки, появившихся в пакете `java.util.concurrent.locks`, ознакомиться с понятиями справедливой блокировки и блокировок чтения и записи. Рассмотреть возможность использовать объекты условий для согласованной работы потоков.

Задание 1. Рассматриваемая ниже программа имитирует работу банка со счетами. В программе случайным образом генерируется транзакция по переводу денег с одного счета на другой. Каждый счет обслуживается отдельным потоком. В процессе транзакции произвольная сумма снимается со счета, обслуживаемого данным потоком, и пересылается на другой счет. Если на счете нет указанной суммы, то перевода не производится.

При выполнении этих операций неизвестно, сколько денег содержится на каждом счете с заданный момент. Однако известна общая сумма, которая должна оставаться неизменной, поскольку транзакции происходят в пределах одного банка.

Запускаемый класс приложения выглядит следующим образом:

```
/* для использования объектов блокировки и объектов условий
   необходимо будет подключить пакет
import java.util.concurrent.locks;
*/

public class SynchBankTest{
  public static void main (String[] args){
   Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
   int i;
   long t = System.currentTimeMillis();
   for (i=0; i< NACCOUNTS; i++) {
         TransferRunnable r =
```

```
                new TransferRunnable (b,i,INITIAL_BALANCE);
            Thread t = new Thread(r);
            t.start();
            t.join();
    }
      long tt = System.currentTimeMillis() - t;
      System.out.println("Total time ="+tt);
 }
 public static final int NACCOUNTS = 10;
 public static final double INITIAL_BALANCE = 1000;
}
```

В классе SynchBankTest создается объект – банк, для которого определяется количество счетов и начальный баланс (сумма) для каждого счета. После этого, для каждого счета создается обслуживающий его объект класса `TransferRunnable`, который должен выполнить несколько операций перевода с данного счета на некоторый другой. Для выполнения такого обслуживания запускается отдельный поток.

Рассмотрим теперь описание класса, моделирующего работу банка.

```
class Bank {
/**
     Конструктор объекта, представляющего банк
*/
  public Bank (int n, double initialBalance) {
   accounts = new double[n];
   for (int i = 0; i<accounts.length; i++)
        accounts[i]   = initialBalance;
/** Для обеспечения синхронизации и согласования работы потоков
     необходимо создать объект блокировки
     bankLock = new ReentrantLock();
     и получить связанный с ни объект условия
     sufficientFunds = bankLock.newCondition();
*/
  }
```

```
/** Метод осуществляющий перевод денег с одного счета на другой
     В представленном варианте не синхронизован
     Для обеспечения синхронизации необходимо заменить
     на описанный ниже вариант
*/
  public void  transfer(int from, int to, double amount){
   if (accounts[from]<amount) return;
   System.out.print (Thread.currentThread());
   accounts[from]-=amount;
   System.out.printf(" %10.2f from %d to %d", amount,from,to);
   accounts[to]+=amount;
   System.out.printf("Total Balance: %10.2f %n",
                    getTotalBalance());
  }
/* измененный вариант метода для перевода денег
public void  transfer(int from, int to, double amount)
     throws InterruptedException
{
// получаем блокировку
     bankLock.lock();
     try {
// далее идет критичный фрагмент кода
          while (accounts[from] < amount)
               sufficientFunds.await();//переход в режим ожидания
          System.out.print (Thread.currentThread());
          accounts[from]-=amount;
          System.out.printf(" %10.2f from %d to %d",
                         amount,from,to);
          accounts[to]+=amount;
          System.out.printf("Total Balance: %10.2f %n",
                    getTotalBalance());
          sufficientFunds.signalAll();//разблокировка
                                      //ожидающих потоков
     }
     finally
     {
```

```
            bankLock.unlock();// операцию разблокирования нужно
                              // выполнить даже при возникновении
                              // исключения
        }
    }
*/
/** Определение суммы средств на всех счетах
     Для корректной работы также требует синхронизации
     Измененный код метода приведен далее
*/
    public double getTotalBalance(){
     double sum=0;
      for (double a:accounts)
       sum+=a;
      return sum;
    }
/** Вариант метода определения суммы средств на всех счетах,
     обеспечивающий синхронизацию
    public double getTotalBalance(){
       bankLock.lock();
       try {
            double sum=0;
            for (double a:accounts)
                 sum+=a;
            return sum;
       }
       finally{
            bankLock.unlock();
       }
}
*/
/** Определение количества счетов в банке
*/
    public int size(){
      return accounts.length;
    }
```

```
  private final double[] accounts;
/* для выполнения блокировок добавляется объект блокировки
     private Lock bankLock;
   для определения условия согласованности переводов со счета на
   счет добавляется объект условие
     private Condition sufficientFunds;
*/
}
```

Приведенное описание класса Bank изначально не содержит никаких средств, обеспечивающих согласованность работы потоков. Выполнив приложение в таком виде можно обнаружить ситуацию гонки потоков, которая будет выражаться как в несогласованной выдаче сообщений, приходящих от разных потоков, осуществляющих перевод средств, так и в неверном балансе суммы всех счетов в банке. Чтобы поучить корректно работающий вариант, следует внести в код исправления в соответствии с приведенными в нем комментариями.

Собственно поток, осуществляющий перевод средств со счета на счет описывается следующим классом:

```
class TransferRunnable implements Runnable{
/**
     Конструктор объекта, реализующего интерфейс Runnable
*/
  public TransferRunnable(Bank b, int from, double max){
    bank = b;
    fromAccount = from;
    maxAmount = max;
  }
  public void run() {
   try {
     for (int j=0; j<MAX_CHANGE; j++){
       int toAccount = (int) (bank.size() * Math.random());
       double amount = maxAmount * Math.random();
       bank.transfer(fromAccount, toAccount, amount);
```

```
        Thread.sleep((int) (DELAY * Math.random()));
       }
     } catch (InterruptedException e ){}
   }
   private Bank bank;
   private int fromAccount;
   private double maxAmount;
   private int DELAY = 100;
   private int MAX_CHANGE=5;
}
```

Упражнения.

a) Выполните приложение, сначала не внося в него изменений, указанных в комментариях? Убедитесь, что имеет место гонка потоков, и периодически возникают неверные выдачи. Рассмотрите, как можно повлиять на гонку, если изменить количество операций со счетами, количество счетов или ввести дополнительные операции засыпания потоков в процессе выполнения операций со счетом.

b) Измените код приложения в соответствии с комментариями. Убедитесь, что после исправления ситуация гонки не возникает ни при каких условиях.

c) Придумайте свое изменение модели работы банка. Убедитесь, что в вашей модели работы банка не возникает ситуация гонки.

d) Сравните затраты времени при выполнении приложения без блокировок и условий ожидания и после использования механизмов синхронизации.

e) Измените код приложения, применяя для синхронизации и определения условий ожидания традиционные средства в языке Java.

f) Модифицируйте код приложения, разделив блокировки в разных методах класса Bank на блокировку чтения и блокировку записи. Проверьте, что это не повлечет возникновения гонки потоков. оцените время выполнения приложения для этого случая.

g) Модифицируйте приложение для задачи о производителях и потребителях так, чтобы для совместной работы использовался массив. Придумайте несколько разных интерпретаций задачи в этом случае.

Вопросы для повторения.

1. Для чего предназначены объекты класса `ReentrantLock`. Каковы правила использования объектов блокировки.
2. Что такое справедливая блокировка?
3. Почему вместо метода `lock()` предпочтительней использовать метод `tryLock()`?
4. Почему при использовании объектов блокировки вызов метода `unlock()` обязательно нужно производить в блоке `finally`?
5. Чем отличаются объекты блокировки чтения и записи от обычных объектов блокировки. Когда их рекомендуется применять?
6. Сколько объектов условий можно связать с одним объектом блокировки?
7. Когда нужно вызывать метод `signalAll()`?
8. Сравните методы `signalAll()` и `signal()`.
9. Что такое тупиковая блокировка? Когда она может возникнуть, какие есть методы предотвращения взаимных блокировок.
10. Какие ограничения имеют неявные объекты блокировки и связанные с ним условия по сравнению с объектами классов `ReentrantLock` и `Condition`?

По завершению изучения модуля студент должен уметь:

1. Распознавать возможность возникновения ситуации гонки потоков.
2. Использовать средства синхронизации для предотвращения гонки потоков.
3. Использовать механизмы ожидания – уведомления для организации согласованной работы потоков.

МОДУЛЬ 3

Изучение принципов разработки многопоточных программ на примерах классических задач

Лабораторная работа 9

Решение задачи о «писателях и читателях»

Цель работы: Рассмотреть на примере классической задачи синхронизации – задачи о читателях и писателях приемы использования уже изученных средств синхронизации.

Задача о читателях и писателях. Совместно используемые данные (базу данных) разделяют два типа процессов – читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, а транзакции писателей и просматривают, и изменяют записи. Каждая отдельная транзакция переводит базу данных из одного непротиворечивого состояния в другое. Для предотвращения взаимного влияния процесс–писатель должен иметь исключительный доступ к базе данных. Если к базе данных не обращается ни один из процессов–писателей, то выполнять транзакции одновременно могут сколько угодно читателей.

Следующее приложение иллюстрирует задачу «писатели-читатели»:

```
class DBase {
   private int data  =0; // «хранилище» в базе данных
   private int nr =0;
   synchronized void startRead(){
               nr++;
   }
   synchronized void endRead(){
               nr--;
              if (nr==0) notifyall();
   }
public void read() {
    startRead();
```

```
   System.out.println("Прочитано: " + data);
   endRead();
}
public synchronized void write() {
   while(nr>0)
//приостановить, если есть активные потоки-читатели
   try{
       wait();
   } catch (InterruptedException e) {
      System.out.println("Interrupt");
                  return;
   }
   data++;
   System.out. println("Записано: " + data);
   notifyAll();
} }
class Writer implements Runnable {
 DBase q;
 int rounds;
 Thread t;
Writer (DBase q, int rd) {
   this.q = q;
  rounds = rd;
   t=new Thread(this,  " Writer "); t.start();
}
public void run(){
for (int i=0; i<rounds; i++){
   q.put();
   try{
     t.sleep(1);
   }
   catch (InterruptedException e) {
        System.out.println("Interrupt");
        return;
   }
} } }
```

```
class Reader implements Runnable {
  DBase q;
  int rounds;
  Thread t;
  Reader (DBase q, int rd) {
       this.q = q;
       rounds = rd;
       t=new Thread(this,  " Reader ");
       t.start();
  }
  public void run()   {
       for (int i=0; i<rounds; i++){
            q.get();
            try{
                  t.sleep(5);
            }
            catch (InterruptedException e) {
                 System.out.println("Interrupt");
                 return;
            }
       }
} }
class ReadersWriters {
  public static void main(String args[]) {
       if (args.length==0) {
            System.out.println("Put rounds in params");
            System.exit(1);
       }
  int rounds = Integer.parseInt(args[0],10);
  DBase q = new DBase();
  new Writer (q,rounds);
  new Reader (q,rounds);
} }
```

Замечание. Внимательно проанализировав код программы можно убедиться, что задача о читателях и писателях аналогично рассматриваемой

ранее задаче производитель–потребитель. Но в отличие от последней, где доступ к данным был взаимоисключающим, в задаче о читателях и писателях разрешены либо параллельные операции чтения, либо одна операция записи. Методы `startRead`, `endRead` и `write` синхронизованы, поэтому в любой момент может выполняться только один из них. Следовательно, когда активен метод `startRead` или `endRead`, поток писателя выполняться не может. Метод `read` не синхронизирован, поэтому его одновременно могут вызывать несколько потоков.

Если поток писателя вызывает метод `write`, когда поток читателя считывает данные, значение `nr` положительно, поэтому писатель перейдет в состояние ожидания. Писатель может работать, когда значение `nr` становится равным нулю.

Упражнения.

a) Увеличьте количество потоков–читателей и проанализируйте, какие возможны ситуации.

b) Перепишите код приложения так, чтобы использовались явные объекты блокировки и условий.

c) Что изменится, если в качестве объекта блокировки использовать блокировку чтения и записи.

d) Проверьте, как повлияет, если в качестве объектов блокировки использовать справедливую блокировку.

e) Измените классы «писатель» и «читатель» так, чтобы у них была разная длительность работы. Может, какие-то из читателей, случайным образом будут переходить в режим бездействия.

f) Увеличьте количество писателей, проследите как это повлияет на разные варианты реализации.

g) Придумайте содержательные задачи, при решении которых можно использовать модель типа «читатели–писатели». Как для решения таких задач придется изменить структуру данных.

h) Как изменить класс, описывающий поток–читатель так, чтобы он мог запускать на чтение только одного читателя.

i) Попробуйте запустить большее количество потоков–писателей, проанализируйте ситуацию.

Лабораторная работа 10

Задача об обедающих философах

Цель работы: Рассмотреть на примере классической задачи об обедающих философах проблемы, которые возникают, когда процессу нужен доступ одновременно более чем к одному ресурсу.

Задача об обедающих философах. Пять философов сидят вокруг круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. При этом каждый из философов пользуется двумя вилками. Между каждой парой философов лежит одна вилка. Философы договорились, что каждый будет пользоваться только теми вилками, которые лежат рядом с ним (слева и справа).

Программа, моделирующая поведение философов должна избегать неудачной ситуации, в которой все философы голодны, но ни один из них не может взять обе вилки. Например, когда каждый из философов держит по одной вилке и не хочет отдавать ее.

Рассмотрим классы, которые будут использованы при решении данной задачи. Класс `Fork` описывает вилку. Для удобства будем считать, что у каждой вилки есть уникальный номер или название. Кроме того, у вилки должен быть признак – занята она или свободна. Два метода `getFork()` и `putFork()` изменяют этот признак.

```
class Fork {
   int numb;
   boolean free =true;
   Fork (int n) {
        numb = n;
   }
```

```
   boolean isFree(){
       return free;
  }
   void getFork(){
      free = false;
   }
    void putFork(){
       free = true;
   }
}
```

Для удобства разобьем вилки на пары по тому, какой из философов может взять их для еды. Каждая из вилок будет принадлежать двум парам. В паре вилки упорядочены (левая и правая) по их расположению относительно берущего пару философа. Для пары вилок определены синхронизованные методы `getForks()` и `putForks()`. В методе `getForks()` проверяется, свободны ли обе вилки из пары. Если хотя бы одна из них занята, то метод переходит в состояние ожидания. Обратите внимание, что для метода `wait()` задан таймаут. Если по истечении этого срока пара не освободится, то ожидание все равно будет прервано и поток сможет продолжить свое выполнение. Почему следует устанавливать таймаут на ожидание, будет объяснено далее.

```
class ForksPair{
     Fork left,right;
     ForksPair(Fork l, Fork r) {
          left = l;
          right = r;
      }
      synchronized void getForks() {
          while ( ! left.isFree() || !right.isFree())
           try{
                 System.out.println("Нельзя взять пару вилок");
                 wait(10);
          }
```

```
            catch(InterruptedException e) {
              System.out.println(e);
              return;
            }
            left.getFork();
            right.getFork();
            notifyAll();
        }
        synchronized void putForks() {
          left.putFork();
          right.putFork();
           notifyAll();
        }
}
```

Класс, описывающий философа, является расширением класса `Thread`. Для каждого философа определяется имя, пара вилок, которую он может брать и выполняется подсчет, сколько раз он успел поесть. В классической задаче о философах считается, что блюдо спагетти неисчерпаемо и процесс еды и размышлений продолжается бесконечно. Поскольку нам интересно проследить за результатами работы потоков, в частности узнать, все ли философы имели возможность поесть, следует в какой-то момент завершить выполнение и подвести итоги. Поэтому введем предположение, что на блюде ограниченное количество порций, например – 50. Для того, чтобы вести учет съеденных порций в классе `Philosof` имеется статическая переменная `timer`. Она управляет продолжением цикла в методе `run()`. Как только один из потоков (философов) съест последнюю порцию, он прекратит выполняться. Завершатся и все остальные потоки, только если они не находятся в состоянии ожидания освобождения вилок, эти потоки могут привести к зависанию приложения, чтобы этого не произошло, в методе `getForks()` предусмотрен таймаут ожидания.

```
class Philosof  extends Thread {
    static int timer =0;
```

```
String name;
int kol=0;
ForksPair p;
Philosof (String nm, Fork l, Fork r) {
       name = nm;
       p = new ForksPair(l,r);
}
 public void run() {
     while (timer <50){
          try {
              Thread.sleep(1);
          }catch(InterruptedException e) {
                System.out.println(e);
                    return;
           }
          System.out.println("Философ "+name+
                " собирается поесть");
          p.getForks();
          System.out.println("Философ "+name+" взял вилки");
           try {
               Thread.sleep(1);
           }catch(InterruptedException e) {
                  System.out.println(e);
                      return;
            }
           p.putForks();
           System.out.println("Философ "+name+
                                " положил  вилки");
           kol++;
           timer++;
          System.out.println("Философ "+name+" размышляет");
           try {
               Thread.sleep(1);
           }catch(InterruptedException e) {
                  System.out.println(e);
                      return;
```

```
            }
      }
}
      int get_kol() {
         return kol;
      }
}
```

Основной класс приложения создает массив из пять вилок и массив из пяти философов. После этого запускается поток для каждого из философов. По завершении их работы выдается информация о том, сколько раз поел каждый философ. Чтобы обеспечить корректную выдачу результатов, основной поток приложения ожидает завершения всех потоков – филисофов.

```
public class TestPhilosof {
  public static void main ( String [] args){
    Fork f[] = new Fork[5];
    String names[]= {"ph1","ph2","ph3","ph4", "ph5"};
    Philosof ph[] = new Philosof[5];
    for (int i=0;i<5;i++){
       f[i] = new Fork(i);
    }
    for (int i=0;i<5;i++){
      if (i==0)
         ph[i] = new Philosof(names[i],f[4],f[0]);
     else
         ph[i] = new Philosof(names[i],f[i-1],f[i]);
    }
    for (int i=0; i<5;i++)
        ph[i].start();
    for (int i=0;i<5;i++)
        try {
           ph[i].join();
        }
        catch(InterruptedException e) {
            System.out.println(e);
```

```
            return;
          }
    for (int i =0; i<5;i++){
        System.out.println("Философ "+names[i]+
              " поел "+ph[i].get_kol()+" раз");
   }
}}
```

Упражнения.

a) Проверьте, что будет происходить, если в методе ожидания убрать таймаут. Чтобы прервать выполнение зависшего приложения необходимо нажать Ctrl+C. Если вместо этого нажать Ctrl+Break, то можно увидеть информацию о возникшей проблеме с потоками.

b) Перепишите класс `Philosof`, использовав вместо расширения класса Thread, реализацию интерфейса `Runnnable`.

c) Выполните реализацию, использовав вместо неявных объектов блокировки и условий ожидания средства классов `ReentrantLock` и `Condition`.

d) Предположим, что все вилки кладутся на середину стола и философы могут брать из них любые две свободные. Реализуйте решения для такого варианта. Может оказаться полезным вести объект «стол», который бы управлял выдачей вилок.

В задаче об обедающих философах каждому из процессов было необходимо наличие двух ресурсов. Видоизменим условие. Пусть процессу нужен только один ресурс, но его можно выбирать из нескольких доступных.

e) Имеется два принтера и несколько заданий, использующих их для печати. Когда заданию необходимо вывести данные на печать, оно ожидает, пока какой–либо из принтеров не освободится. После использования принтера заданием он становится свободен и может быть использован любым другим процессом.

f) Видоизмените предыдущую задачу следующим образом. Пусть принтеры разные, например один из них лазерный или широкоформатный. Все задания можно разбить на группы – те задания, которым подходит любой из принтеров и те, для которых нужен конкретный.

По завершению изучения модуля студент должен уметь:

Применять классические решения многопоточных задач к реальным проблемам многопоточного программирования

МОДУЛЬ 4

Применение средств пакета java.util.concurrent

Лабораторная работа 11

Блокирующие очереди

Цель работы: Познакомиться с возможностями появившихся в пакете java.util.concurrent классов блокирующих очередей. Блокирующие очереди – полезный инструмент для координации действий нескольких потоков. Некоторые потоки могут помещать результаты своей работы в очередь. Если очередь заполнена, то потоки приостанавливают свою работу, пока из очереди не начнут удаляться элементы. Другие потоки извлекают данные из очереди. Если очередь пуста, то их работа приостанавливается. Таким образом, сама блокирующая очередь следит, чтобы потоки работали согласованно и не требует программировать ожидание условий.

Задание 1. Программа, приводимая далее, демонстрирует использование блокирующих очередей для управления набором потоков. Программа ищет файлы в каталоге и его подкаталогах. В найденных файлах ищутся строки, содержащие заданное ключевое слово.

Поток – поставщик выбирает все файлы из всех подкаталогов и помещает их, конечно, не сами файлы, а ссылки на них, в блокирующую очередь. Эта операция выполняется достаточно быстро.

Одновременно запускается большое количество поисковых потоков. Каждый поток извлекает файл из очереди, построчно анализирует и выводит все строки, содержащие ключевое слово. Затем поток переходит к следующему файлу.

В этом примере в качестве механизма синхронизации выступает очередь.

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

public class BlockingQueueTest {
   public static void main (String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Введите базовую директорию :");
        String directory = in.nextLine();
        System.out.print("Введите искомое слово:");
        String keyword = in.nextLine();
        final int FILE_QUEUE_SIZE = 10;
        final int SEARCH_THREADS = 100;
        BlockingQueue <File> queue =
                new ArrayBlockingQueue <File> (FILE_QUEUE_SIZE);
        FileEnumerationTask enumerator =
           new FileEnumerationTask(queue, new File (directory));
// запускаем поток на просмотр каталога
        new Thread(enumerator).start();
// запускаем потоки для анализа найденных файлов
        for (int i=1; i<= SEARCH_THREADS; i++)
             new Thread (new SearchTask(queue,keyword)).start();
  }
}
/**
 Данный поток помещает все файлы из каталога и подкаталога в
блокирующую очередь
*/
```

```
class FileEnumerationTask implements Runnable {
   public FileEnumerationTask (BlockingQueue <File> q, File dir)
 {
      queue = q;
      startingDirectory = dir;
   }
   public void run() {
      try {
          enumerate(startingDirectory);
          queue.put(DUMMY);
      }
      catch (InterruptedException e) {}
   }
   public void enumerate (File directory)
                    throws InterruptedException {
      File[] files = directory.listFiles();
      for (File f : files ) {
          if (f.isDirectory()) enumerate (f); //рекурсивный вызов
      else queue.put(f);//если очередь заполнена,
                        // поток приостановится
      }}

   public static File DUMMY = new File("");
   private BlockingQueue <File> queue;
   private File startingDirectory;
}
/**
  Потоки для поиска по ключесвому слову
*/
class SearchTask implements Runnable {
   public SearchTask (BlockingQueue <File> q, String key) {
       queue = q;
       keyword = key;
   }
   public void run() {
       try {
```

```
      boolean done = false;
      while ( !done) {
         File f = queue.take(); //если очередь пуста,
                                // поток приостановится
         if (f == FileEnumerationTask.DUMMY) {
            queue.put(f);
            done = true;
                   }
        else search(f);
      }
        }
        catch (IOException e) { e.printStackTrace(); }
         catch (InterruptedException e) {}
   }
   public void search (File f) throws IOException {
        Scanner in = new Scanner(new FileInputStream(f));
        int lineNumber = 0;
        while (in.hasNextLine()) {
          lineNumber++;
          String line = in.nextLine();
          if (line.contains(keyword))
           System.out.println(f.getPath()+"  "+
                    lineNumber+"   "+ line);
        }
        in.close();
   }
   private BlockingQueue <File> queue;
   private String keyword;
}
```

Упражнения.

a) Разработайте решение задачи «писатели – читатели», используя для обмена информацией блокирующую очередь.

b) В приведенном решении практически отсутствует промежуточная печать, которая позволяла бы следить за ходом выполнения.

В частности, за тем, как происходит заполнение и опустошение очереди, когда возникают ситуации блокировки из-за того, что очередь пустая или заполненная. Чтобы было легче анализировать, сделайте выдачу этой информации в файл.

c) Используйте блокирующую очередь для решения задачи о неделимой рассылке. Имеется один процесс производитель, который помещает свои сообщения в ограниченную очередь. Несколько процессов–потребителей должны все прочесть каждое из сообщений. Когда сообщение прочитано всеми процессами, оно удаляется из очереди.

d) В приложении создается много потоков, анализирующих файлы, как по окончании работы проанализировать, сколько из них было реально задействовано. Обрабатывали какие-то из них несколько файлов?

Лабораторная работа 12

Классы–синхронизаторы. Семафоры

Цель работы: Познакомиться с возможностями появившихся в пакете java.util.concurrent классов–синхронизаторов. В данной лабораторной работе рассматривается класс Semaphore.

Задание 1. Рассмотрим еще один вариант решения задачи о читателях и писателях. В этом приложении обмен информацией между читателями и писателями осуществляется через циклический массив заданного размера. В качестве механизма синхронизации используются семафоры.

Один из семафоров управляет писателями, он рассчитан на количество разрешений, равное размеру массива. В дальнейшем, разрешения будут выдавать читатели, после прочтения информации из массива.

Второй семафор управляет читателями. Он рассчитан на одно разрешение. После, того как писатель помещает данные в массив, он выдает очередное разрешение для семафора читателей.

Класс, который организует хранилище данных в виде массива, выглядит так:

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

public class SmartQ {
    private SmartQRealization impl;
    private Semaphore sWrite;
    private Semaphore sRead;
    public SmartQ(int n) {
        sWrite = new Semaphore(n);
        sRead = new Semaphore(1);
        impl = new SmartQRealization(n);
     try{
//семафор сразу имеет разрешение, заберем его, чтобы читатель не
//смог воспользоваться им раньше, чем писатель запишет хотя бы
//одно значение
        sRead.acquire();
        } catch(InterruptedException e){};
    }
    public void push(int i) throws InterruptedException {
        sWrite.acquire();//запрос разрешения на запись
        synchronized(impl) {
            impl.push(i);
            sRead.release();//выдано разрешение на чтение
        }
    }
    public int pop() throws InterruptedException {
        sRead.acquire();//запрос разрешения на чтение
        synchronized(impl) {
            int tmp = impl.pop();
            sWrite.release();//выдано разрешение на запись.
            return tmp;
        }

    }
```

Классы писателя и читателя выглядят следующим образом:

```
public class Reader implements Runnable {
```

```
    private SmartQ q;
    private String name;//для отладки
    private int readTime;
    private boolean stopFlag;
    public Reader(String name, SmartQ q, int readTime) {
        this.name = name;
        this.q = q;
        this.readTime = readTime;
        stopFlag = false;
    }
    public void stop() { stopFlag = true; }
    public void run() {
        try {
            while (!stopFlag) {
                System.out.println("   "+name+
                        "       get ="+q.pop() );
                Thread.sleep(readTime);
            }
        } catch(Exception e) {}
    }
public class Writer implements Runnable {
    private SmartQ q;
    private int writeTime;
    private boolean stopFlag;
    private int base, step;
    public Writer(SmartQ q, int writeTime, int base, int step) {
        this.q = q;
        this.writeTime = writeTime;
        stopFlag = false;
        this.base = base;
        this.step = step;
    }
    public void stop() { stopFlag = true; }
    public void run() {
        try {
            int i = base;
```

```
            while (!stopFlag) {
               System.out.println("put="+i);
                q.push(i);
                i += step;
                Thread.sleep(writeTime);
            }
        } catch(Exception e) {}
}}
```

Наконец рассмотрим класс, который создает хранилище данных, создает и запускает потоки. В этом классе создается несколько писателей и несколько читателей. Они организуются в списки на базе массива.

```
import java.util.*;
public class Main {
    static final int nWriters = 10;
    public static void main(String[] args) {
        SmartQ q = new SmartQ(10);
        List<Reader> listR = new ArrayList<Reader>();
        List<Writer> listW = new ArrayList<Writer>();
        for ( int i = 0;  i < nWriters;  ++i) {
            Reader r = new Reader("#"+i,q, 10);
            new Thread(r).start();
            listR.add(r);
            Writer w = new Writer(q, 10, i, nWriters);
            new Thread(w).start();
            listW.add(w);
        }
        try{
            Thread.sleep(100);
        } catch(Exception e) {}
        for(Reader r : listR) {
            r.stop();
        }
        for(Writer w : listW) {
            w.stop();
        }
```

```
    }
}
```

Упражнения.

a) Проведите эксперименты с семафорами. Попробуйте изменить количество допустимых разрешений у семафора для читателей. Что произойдет? И у семафора для писателей. Чтобы анализировать состояние семафоров, воспользуйтесь соответствующими методами класса Semaphore.

b) Измените время засыпания потоков, как это отразится на работе всего приложения.

c) Введенное в классе Reader имя позволяет следить, кто из читателей выполняет операцию. Добавьте аналогичную возможность для класса Writer.

Лабораторная работа 13

Классы–синхронизаторы. Барьеры

Цель работы: Рассмотреть еще несколько классов–синхронизаторов из пакета java.util.concurrent. В данной лабораторной работе используется циклический барьер.

Задание 1. Чаще всего для демонстрации использования барьеров рассматривают задачи обработки матриц. При этом потоки обрабатывают отдельные части матрицы – строки, полосы, блоки. По окончании работы всех потоком выполняется какая–то итоговая обработка. После этого процесс может быть повторен итеративно. В приводимом ниже примере выполнятся параллельное суммирование строк треугольной матрицы. Перед запуском потоков создается циклический барьер, ожидающий их завершения. Для барьера определено действие, которое выполнится, когда все потоки достигнут барьера.

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

   public class Summary {
     private static int matrix[][] = {
```

```
        {1},
        {2, 2},
        {3, 3, 3},
        {4, 4, 4, 4},
        {5, 5, 5, 5, 5}
      };
      private static int results[];

      private static class Summer extends Thread {
        int row;
        CyclicBarrier barrier;
        Summer(CyclicBarrier barrier, int row) {
          this.barrier = barrier;
          this.row = row;
        }
       public void run() {
        int columns = matrix[row].length; //размер строки матрицы
        int sum = 0;
        for (int i=0; i<columns; i++) {
            sum += matrix[row][i];
        }
        results[row] = sum; //готовим для итоговой суммы
        System.out.println(
             "Results for row " + row + " are : " + sum);
// ожидаем завершения остальных потоков
        try {
            barrier.await();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        } catch (BrokenBarrierException ex) {
            ex.printStackTrace();
        }
      }
      }

      public static void main(String args[]) {
```

```
        final int rows = matrix.length;
        results = new int[rows];
//определяем действие барьера, которое выполнится, когда все
//потоки достигнут барьера
        Runnable merger = new Runnable() {
          public void run() {
            int sum = 0;
            for (int i=0; i<rows; i++) {
              sum += results[i];
            }
            System.out.println("Results are: " + sum);
          }
        };
//циклический барьер создается по числу запускаемых потоков
// в данной задаче число потоков равно числу строк в матрице
        CyclicBarrier barrier = new CyclicBarrier(rows, merger);
//запускаем отдельный поток для каждой строки
        for (int i=0; i<rows; i++) {
          new Summer(barrier, i).start();
        }
        System.out.println("Waiting...");
      }
    }
```

Упражнения.

a) Измените размер матрицы. Как это отразится на выдаваемых результатах?

b) Чтобы повторно использовать циклический барьер, вызовите для него методы reset() и повторно запустите потоки на выполнение. Что изменится в результатах выдачи?

c) Попробуйте управлять приоритетами потоков. Как это повлияет на порядок выдачи результатов?

d) Создайте многопоточное приложение с циклическим барьером, выполняющее любую обработку матрицы итерационно. Определите, какие

действия, и с какими частями матрицы можно выполнять параллельно. Определите также, необходима ли какая-то подготовка перед следующей итерацией.

е) Изучите возможности синхронизатора CountDownLanch. Разработайте приложение, демонстрирующее его использование.

Лабораторная работа 14

Асинхронные вычисления

Цель работы: Познакомиться с интерфейсами Callable и Future, возможностью создания на их основе асинхронных вычислений. Изучить на примере использование класса–оболочки FutureTask.

Задание 1. Приводимая ниже программа предназначена для решения задачи, аналогичной рассмотренной в лабораторной работе 11. Но чтобы отразить ее вычислительный характер, поиск ключевого слова сопровождается подсчетом количества файлов, в которых оно найдено. Подсчет ведется отдельно для каждого каталога и общим итогом по всем файлам и подкаталогам.

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

public class FutureTest {
   public static void main (String[] args) {
       Scanner in = new Scanner(System.in);
       System.out.print("Введите базовую директорию :");
       String directory = in.nextLine();
       System.out.print("Введите искомое слово:");
       String keyword = in.nextLine();
      MatchCounter counter =
               new MatchCounter (new File(directory), keyword);
      FutureTask <Integer> task =
               new FutureTask <Integer> (counter);
      Thread t = new Thread (task);
```

```
        t.start();
        try {
           System.out.println(task.get() + "  найдено файлов");
        }
        catch (ExecutionException e) {
                e.printStackTrace();
        }
        catch (InterruptedException e) {}
  }
}
/**  Данный класс подсчитывает файлы, находящиеся в каталоге и
его подкаталогах, в которых имеется ключевое слово.
*/
class MatchCounter implements Callable<Integer> {
   public MatchCounter ( File dir, String key) {
      directory = dir;
      keyword = key;
   }
//Метод call() интерфейса Callable аналогичен методу run()
// интерфейса Runnable, но возвращает результат
   public Integer call() {
        count = 0;
         try {
           File[] files =directory.listFiles();
           ArrayList <Future <Integer>> results =
                     new ArrayList <Future <Integer>>();
           for (File file : files)
                     if (file.isDirectory()) {
                          MatchCounter counter =
                               new MatchCounter (file,keyword);
                          FutureTask <Integer> task =
                               new  FutureTask <Integer>(counter);
           results.add(task);
           Thread t = new Thread(task);
           t.start();
         }
```

```
        else {
          if (search(file)) count++;
        }
      for (Future <Integer> result : results)
          try {
          count += result.get();
        }
        catch (ExecutionException e) {
          e.printStackTrace();
        }
        }
        catch (InterruptedException e) {}
        return count;
    }
    public boolean search (File f) {
        try {
           Scanner in = new Scanner(new FileInputStream(f));
           boolean found = false;
          while (!found && in.hasNextLine()) {
               String line = in.nextLine();
               if (line.contains(keyword))
                    found = true;
          }
          in.close();
          return found;
        }
        catch (IOException e) {
              return false;
        }
    }
    private File directory;
    private String keyword;
    private int count;
}
```

Упражнения.

a) Измените задачу так, чтобы каждый поток считал для файла количество строк, в которых встречается ключевой слово.

b) Разработайте многопоточное приложение, которое подсчитывает, сколько раз встретилось ключевое слово в заданном файле. Потоки должны получать для анализа отдельную строку.

Лабораторная работа 15

Пулы потоков

Цель работы: Научиться повышать эффективность использования потоков путем создания пула потоков. Познакомиться с методами класса Executors.

Пул содержит ограниченное количество потоков, готовых к выполнению. Помещаемый в пул объект Runnable будет выполнен одним из свободных потоков или поставлен в очередь до тех пор, пока не появится свободный поток.

Задание 1. В примерах из лабораторных работ 11 и 14 создавалось большое количество потоков (по одному на каждый файл). Каждый из потоков выполнял относительно небольшую работу и уничтожался. Формирование нового потока и завершение работы существующего потока связано с определенными затратами из–за взаимодействия с операционной системой. Кроме того, большое количество одновременно запущенных потоков (если оно существенно превышает количество доступных процессоров) существенно снижает производительность программы и может даже разрушить виртуальную машину. С учетом этих замечаний код программы из лабораторной работы 14 доработан так, чтобы использовать пул потоков.

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

```
public class ThreadPoolTest {
   public static void main (String[] args)  throws Exception{
       Scanner in = new Scanner(System.in);
       System.out.print("Введите базовую директорию :");
       String directory = in.nextLine();
       System.out.print("Введите искомое слово:");
       String keyword = in.nextLine();
//Организуем пул потоков с возможностью его увеличения
      ExecutorService pool = Executors.newCachedThreadPool();
      MatchCounter counter =
          new MatchCounter (new File(directory), keyword,pool);
      Future <Integer> result = pool.submit (counter);
     try {
               System.out.println(result.get() + "   файлов");
     }
     catch (ExecutionException e) {
               e.printStackTrace();
     }
     catch (InterruptedException e) {}
     pool.shutdown();
     int largestPoolSize =
               ((ThreadPoolExecutor) pool).getLargestPoolSize();
       System.out.println ("Наибольший размер пула ="+
                                  largestPoolSize);
  }
}
/**
 Данный класс подсчитывает файлы, находящиеся в каталоге и его
подкаталогах, в которых имеется ключевое слово.
*/
class MatchCounter implements Callable<Integer> {
  public MatchCounter ( File dir, String key, ExecutorService p)
{
      directory = dir;
      keyword = key;
      pool = p;
```

```
}
   public Integer call() {
       count = 0;
       try {
               File[] files =directory.listFiles();
               ArrayList <Future <Integer>> results =
                         new ArrayList <Future <Integer>>();
     for (File file : files)
      if (file.isDirectory()) {
          MatchCounter counter =
                    new MatchCounter (file,keyword,pool);
         Future <Integer> result = pool.submit(counter);
          results.add(result);
      }
      else {
          if (search(file)) count++;
     }
     for (Future <Integer> result : results)
      try {
          count += result.get();
        }
        catch (ExecutionException e) {
          e.printStackTrace();
        }
        }
        catch (InterruptedException e) {}
        return count;
   }
   public boolean search (File f) {
     try {
      Scanner in = new Scanner(new FileInputStream(f));
      boolean found = false;
      while (!found && in.hasNextLine()) {
         String line = in.nextLine();
         if (line.contains(keyword))
           found = true;
```

```
      }
      in.close();
      return found;
    }
    catch (IOException e) {
       return false;
    }
   }
   private File directory;
   private String keyword;
   private int count;
   private ExecutorService pool;
}
```

Упражнения.

a) Добавьте в приложение возможность подсчитывать количество рассмотренных файлов. Сравните это количество с количеством потоков в пуле. Всегда ли эти значения совпадают.

b) Сделайте пул ограниченным и сравните по времени выполнение приложения, когда требуется количество потоков большее, чем размер пула.

c) Сделайте упражнения к предыдущей лабораторной работе, используя ограниченный пул потоков.

Вопросы для повторения.

1. Каким должно быть соотношение между количеством одновременно выполняемых потоков и количеством процессоров для эффективной работы многопоточного приложения?

2. Почему большое количество одновременно запущенных потоков замедляет выполнение приложения?

3. Какие проблемы могут возникать, если одновременно выполняется мало потоков, может быть только один, но в приложении много потоков создается и запускается.

4. Что такое пул потоков? Как организована работа пула?

5. Что происходит с запущенным потоком, если ему нет места в пуле?

По завершению изучения модуля студент должен уметь:

1. Использовать разнообразные средства управления потоками, синхронизации, а также наборы данных для безопасной работы с потоками, появившиеся в версии JDK1.5.
2. Уметь создавать приложения для асинхронных вычислений.
3. Уметь эффективно управлять потоками с помощью пула потоков.

МОДУЛЬ 5

Создание многопоточного сервера

Лабораторная работа 16

Использование потоков при разработке серверов

Цель работы: Научиться использовать средства многопоточности при программировании серверов. Основное назначение серверов – одновременно предоставлять услуги нескольким клиентам. Чтобы сервер мог одновременно обслуживать несколько клиентов, он должен выделять для взаимодействи с каждым клиентом поток.

Задание 1. Рассмотрим простейшую организацию сервера, использующего протокол TCP/IP. Для реализации сетевого соединения используем сокеты. Основная работа данного сервера заключается в том, что он возвращает сообщение, полученное от клиента назад, снабдив его дополнительным префиксом. Сервер ждет, пока клиент пришлет ему сообщение о завершении работы (END). Сам сервер работает бесконечно. Чтобы завершить выполнение сервера нужно нажать CTRL+C.

```
// Сервер, использующий многопоточность
// для обслуживания любого числа клиентов.
import java.io.*;
```

```
import java.net.*;
/** Данный класс реализует взаимодействие сервера с одним
клиентом, создавая для него поток и  завершая его,
после получения от клиента сообщения о завершении сеанса работы
*/
class ServerOneThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServerOneThread (Socket s) throws IOException {
    socket = s;// получение сокета для обмена данными
    in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(
        new BufferedWriter(new OutputStreamWriter(
            socket.getOutputStream())), true);
// Если какой либо, указанный выше класс выбросит исключение
// вызывающая процедура ответственна за закрытие сокета
// В противном случае нить(поток) закроет его.
    start();
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) {
               out.println(”close”);
               Thread.currentThread().interrupt();
        }
        System.out.println("Echoing: " + str);
        out.println(”ECHO:”+str);
      }
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
```

```
        System.out.println("closing...");
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
    }
  }
}
/** Теперь создаем многопоточный сервер
*/
public class MultiServer {
  static final int PORT = 8080;
  public static void main(String[] args)throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
// Останавливаем выполнение, до нового соединения:
        Socket socket = s.accept();
        try {
          new ServeOneThread(socket);
        } catch(IOException e) {
// Если неудача - закрываем сокет,
// в противном случае нить закроет его:
          socket.close();
        }
      }
    } finally {
      s.close();
    }
  }
}
```

Клиент может быть любым, если он соединяется по протоколу TCP/IP. Например, можно соединиться готовым клиентом telnet введя команду

> open localhost 8080

После этого можно будет вводить в консоли telnet любые строки и видеть, как их принимает сервер и отсылает обратно, сопроводив префиксом “ECHO:”.

Однако мы рассмотрим собственный вариант клиента, чтобы в дальнейшем провести с ним ряд экспериментов.

```
// Пример клиента для доступа к серверу
import java.net.*;
import java.io.*;
import java.util.*;

public class Client {
  public static void main(String[] args) throws IOException {
// Установка параметра в null в getByName()
// возвращает специальный IP address - "Локальную петлю",
// для тестирования на одной машине без наличия сети
    InetAddress addr =
      InetAddress.getByName(null);
// Альтернативно Вы можете использовать адрес или имя:
// InetAddress addr = InetAddress.getByName("127.0.0.1");
// InetAddress addr = InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket = new Socket(addr, 8080);
     boolean cont = true;
     Scanner input = new Scanner(System.in);
     try {
      System.out.println("socket = " + socket);
      BufferedReader in = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));
      PrintWriter out = new PrintWriter( new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())),true);
      while (cont) {
         System.out.print(">>");
         String phrase = input.nextLine();
         out.println(phrase);
         String str = in.readLine();
```

```
            System.out.println(str);
            if (str.equals("close") )
              cont=false;
          }

        } finally {
          System.out.println("closing...");
          socket.close();
        }
      }
}
```

Указание. После запуска сервер продолжает работать бесконечно, если не возникнет исключительная ситуация. Для того, чтобы запустить программу для клиента на том же компьютере, необходимо открыть еще одно консольное окно. Чтобы отсоединиться от сервера, клиент должен послать сообщение “END”. Чтобы остановить сервер, нужно нажать CTRL+C.

Упражнения.

a) Выполните эксперименты с подключением к серверу одновременно нескольких клиентов. Это могут быть несколько запущенных в разных окнах приложений Client. Или программа telnet. Убедитесь, что сервер работает одновременно со всеми клиентами.

b) Измените клиента так, чтобы он сообщал серверу свое имя, а сервер при ответе добавлял это имя в качестве префикса.

c) Если есть возможность, подключитесь клиентами к серверу с разных компьютеров по сети.

d) Измените приложения клиента и сервера так, чтобы подсоединившийся клиент передавал серверу имя файла, а сервер возвращал клиенту файл построчно или передавал код ошибки, если файл не найден. Клиент может выдавать файл построчно на экран, а может сохранять его на диске.

е) Измените сервер так, чтобы он использовал для запуска клиентов ограниченный пул потоков. Проверьте, что будет, если подсоединится клиентов больше, чем размер пула.

Рубежный контроль модуля предполагает выполнение студентами индивидуальных заданий.

Примеры индивидуальных заданий

1. Создайте сервер, который по запросу клиента находит файл с заданным именем и передает его построчно клиенту.
2. Создайте сервер, который подсчитывает, сколько файлов в заданном каталоге содержат ключевое слово. Клиент должен передавать имя каталога и ключевое слово, сервер возвращать количество найденных файлов.
3. Создайте сервер, который по запросу клиента сообщает ему, сколько в данный момент времени клиентов подключено к серверу и их имена.
4. Создайте сервер, который по заданному имени файла, переданному клиентом, и команде клиента выполняет некоторую обработку этого файла и передает результат клиенту. В качестве возможных обработок могут быть: подсчитать количество строк в файле, определить количество слов в файле, вычислить длину максимальной строки и т.п.

По завершению изучения модуля студент должен уметь:

1. Создавать многопоточные серверы, использующие сокетное соединение.

2. Разрабатывать и реализовывать протокол взаимодействия клиента и сервера.

Вопросы для повторения

1. Что такое процесс и поток в современных операционных системах.
2. В чем отличие многопоточного и параллельного программирования.
3. Для чего необходима синхронизация в многопоточных программах.
4. Зачем в язык Java введены средства создания и управления потоками.
5. Какие способы создания потока есть в Java.
6. В каких состояниях может находиться поток. В результате каких действий поток может переходить из одного состояния в другое.
7. Что такое приоритет потока, как следует управлять приоритетами.
8. Когда необходимо применять синхронизованные методы.
9. Для чего предназначены блоки синхронизации, чем они отличаются от синхронизованных методов.
10. Почему синхронизованные методы не полностью реализуют идею мониторов.
11. Какие методы позволяют осуществить обмен сообщениями между потоками.
12. Что такое объект блокировки, и какие средства имеются для явного выполнения операций блокировки/разблокировки.
13. Что такое блокировки чтения – записи. Для чего они используются.
14. Как объекты условий организуют взаимодействие потоков.
15. В чем преимущества использования наборов данных из пакета `java.util.concurrent`.
16. Как осуществляется работа с блокирующими очередями.
17. В чем особенность эффективных очередей и хеш-таблиц.

18. для чего предназначены классы `Callable` и `Future`.

19. Что такое пул потоков, как с ним организуется работа.

20. Какие синхронизаторы имеются в пакете `java.util.concurrent`.

21. Каковы правила использования методов wait() и notify().

22. Объекты блокировки, методы `lock()`, `unlock()`, `tryLock()`. Правила использования.

23. Почему методы `sleep()` и `wait()` должны выполняться в блоке `try{…}catch(…)`.

24. Почему некоторые методы являются «deprecated». Какие вместо них нужно использовать средства.

25. Какие средства позволяют снизить риск возникновения мертвых блокировок.

26. Как в Java определяется понятие поток-демон. Какие правила определения потоков-демонов.

ЛИТЕРАТУРА

1. Хабибулин И.Ш. Самоучитель Java 2. – СПб.: БХВ-Петербург, 2005. – 720 с.
2. Хорсман К., Корнелл Г. Java 2. Библиотека профессионала, том II. Тонкости программирования. 7-е изд. – М.: Издательский дом «Вильямс», 2007. – 1168 с.
3. Эккель Брюс. Философия Java. Библиотека программиста. 3-е изд. – Спб.: Питер, 2003. – 971 с.
4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003. – 512 с.
5. Нотон П.. Шилдт Г. Полный справочник по Java. – Киев, «Диалектика», 1997. или последующие издания.

6. Кен Арнольд, Джеймс Гослинг Язык программирования Java.- Издательство "Питер-Пресс", 1997.
7. Майкл Эферган Java: справочник.- Издательство "Питер Ком", 1998.
8. Джо Вебер Технология Java в подлиннике.- Q"BHV-Санкт-Петербург",1997 .
9. Джейсон Мейнджер. Java: Основы программирования.- М.:Издательская группа BHV, Киев,1997.
10. И.Ю.Баженова Язык программирования Java.- АО "Диалог-МИФИ", 1997 .
11. Джон Родли Создание Java-апплетов.- Издательство НИПФ "ДиаСофт Лтд.",1996
12. Шилдт Г., Холмс Д. Искусство программирования на Java.–М.: Издательсикй дом «Вильямс», 2005.

Ресурсы в сети Интернет

13. http://java.sun.com/docs/
14. http://www.citforum.ru
15. http://www.rusdoc.ru
16. http://www.osp.ru
17. http://javable.com
18. http://javaworld.com