МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра программной инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
О.Г. Локтионова
«01» 03 2024 г.

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ.

Методические указания к выполнению
лабораторных работ
для студентов направления 02.03.03

Курск 2024

УДК 004.2

Составители: В.В. Ефремов, И.Н. Ефремова

Рецензент
Кандидат технических наук, доцент А.*В.Малышев*

Параллельное программирование: методические указания к выполнению лабораторных работ для студентов направления 02.03.03/ Юго-Зап. гос. ун-т; сост.: В.В.Ефремов, И.Н. Ефремова. Курск, 2024. - 23 с.

Содержат описание трех лабораторных работ по дисциплине «Параллельное программирование».

Предназначены для студентов направления 02.03.03

Текст печатается в авторской редакции

Подписано в печать Формат 60х84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ . Бесплатно
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94.

Содержание

Лабораторная работа №1.

Программирование сетевых приложений с помощью интерфейса сокетов.

Содержание отчёта

1. Титульный лист.
2. Формулировка задачи.
3. Листинг программы.
4. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое сокеты?
2. Роль сокетов?
3. Какие операционные системы и среды разработки позволяют работать с сокетами?
4. Основные принципы работы с сокетами?

Задание

Разработать клиент-серверное приложениие с использованием UDP протокола и сокетов на языке Java.

Теоретические и справочные сведения

Сокет´ (socket) — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться на одном компьютере или на различных компьютерах, связанных между собой сетью.

Сокет — абстрактный объект, представляющий конечную точку соединения.

Слово «socket» можно перевести как «розетка», поэтому образно сокеты можно представить себе как две розетки, в которые включен кабель для передачи данных через сеть.

По своей сути программный интерфейс сокетов представляет собой набор функций, которые позволяют программисту решать задачи, связанные с передачей информации между ЭВМ по сети. Если локальная или глобальная сеть построена на основе протокола TCP/IP, то на прикладном уровне взаимодействие между узлами сети можно реализовывать при помощи технологии сокетов.

Изначально интерфейс сокетов разрабатывался в рамках работы над операционной системой Unix (стандарт Berkeley Sockets). Позднее, фирма Microsoft доработала указанный интерфейс (например, добавила в него некоторые необходимые для ОС Windows функции), что привело к появлению интерфейса Windows Sockets или WinSock. Функции, составляющие современную версию интерфейса WinSock, собраны в динамически подключаемой библиотеке WS2_32.DLL.

Основы сетевого взаимодействия с помощью языка Java [1]

Java был разработан как язык сетевого программирования для обеспечения возможности создания клиент/серверных приложений, которые взаимодействуют друг с другом в сети. Java обеспечивает обширную библиотеку сетевых классов, которые позволяют быстро получать доступ к сетевым ресурсам. Основные сетевые возможности Java реализуются в классах и интерфейсах пак та java.net, посредством которых Java обеспечивает потоковое взаимодействие, позволяющее приложениям рассматривать соединение, как поток данных.

Существуют два механизма, предназначенных для сетевого взаимодействия программ, - это сокеты датаграмм, которые используют пользовательский датаграммный протокол (User Datagram Protocol) (UDP) без установления соединения, и сокеты, использующие Протокол управления передачей / Межсетевой протокол (Transmission Control Protocol/Internet Protocol) (TCP/IP), устанавливающий соединение.

Датаграмма - пакет данных, отправленный по сети, прибытие которого, время прибытия и содержание не гарантировано. Не гарантируется также и порядок доставки пакетов. При передаче пакета UDP по какому-либо адресу нет никакой гарантии того, что он будет принят, а также, что по этому адресу вообще существует потребитель пакетов. Аналогично, когда вы получаете датаграмму, у вас нет никаких гарантий, что она не была повреждена в пути следования или что отправитель ожидает подтверждения получения датаграммы. Использование UDP может привести к потере или к дублированию пакетов, что приводит к дополнительным проблемам, связанным с проверкой ошибок и обеспечением надежности передачи данных. Если вам необходимо добиться оптимальной производительности, и вы готовы сократить затраты на проверку целостности информации, пакеты UDP могут оказаться весьма полезными.

При использовании потоковых сокетов, программа устанавливает соединение с другим сокетом и, пока соединение установлено, поток данных протекает между программами, и говорят, что потоковые соке-ты обеспечивают обслуживание на основе установления соединения. TCP/IP является потоковым протоколом на основе установления двунаправленных соединений точка-точка между узлами Интернет, и взаимодействие между компьютерами по этому протоколу предназначено для реализации надежной передачи данных. Все данные, отправленные по каналу передачи, получаются в том же порядке, в котором они переда-вались. В отличие от датаграммных сокетов, сокеты TCP/IP реализуют высоконадежные устойчивые соединения между клиентом и сервером.

Сетевые классы содержат методы для выполнения таких задач как открытие и закрытие соединения с удаленным хостом, переда-

ча и получение пакетов данных, и доступ к ресурсам сети. Ниже представлены некоторые из классов, представленных в пакете java.net:

• DatagramPacket: Представляет объект датаграммного пакета (datagram packet), который содержит данные с информацией об адресе источника и адресе назначения, номера портов источни-ка и назначения. Эта информация используется для передачи датаграммного пакета от одного компьютера к другому по се-ти.

• DatagramSocket: Представляет объект датаграммного сокета, который может посылать и принимать пакеты датаграмм. Пакеты, посланные объектом DatagramSocket, могут приходить к получателю в любом порядке.

• MulticastSocket: Создает мультикастовый (multicast) объект датаграммного сокета, который используется для отправки и приема пакетов датаграмм для групп. IP адреса класса D (IP адреса между 224.0.0.0 и 239.255.255.255) используются для создания групп. Когда сообщение посылается на IP-адрес клас-са D, все клиенты, присоединенные к группе, получают от-правленное сообщение. Этот класс содержит методы joinGroup() и leaveGroup(), которые дают возможность клиен-там объединяться и покидать определенную группу.

• InetAddress: Создает объект, который содержит информацию, состоящую из IP-адреса и имени хоста (host name).

• ServerSocket: Создает объект сокета сервера, который слушает запросы клиента. Объекты ServerSocket используют номер порта для получения запросов клиента.

• Socket: Создает объект сокета клиента, который соединяется с объектом класса ServerSocket для посылки запросов на сервер.

• URL: Представляет объект, который может представлять файл или иной ресурс в Интернет.

Пакет java.net также содержит классы для обработки исключительных ситуаций, которые позволяет обрабатывать сетевые ошибки во время выполнения. Ниже представлены несколько классов

обработки исключительных ситуаций, содержащихся в пакете java.net:

• BindException: Этот объект исключительных ситуаций генерируется, когда возникает ошибка при попытке связаться с сокетом локального адреса или порта. Например, когда не может быть доступен локальный адрес или запрашиваемый порт уже используется.

• ConnectException: Этот объект исключительных ситуаций генерируется, когда возникает ошибка во время соединения сокета с удаленным IP адресом и портом. Например, когда уда-ленное соединение не может быть установлено.

• MalformedURLException: Этот объект исключительных ситуаций генерируется, когда URL содержит недействительный протокол или если ссылка на URL не может быть успешно обрабо-тана.

• UnknownHostException: Этот объект исключительных ситуаций генерируется, когда IP-адрес хоста не может быть определен или не существует.

Сервер UDP представляет собой сетевое приложение использующее протокол UDP для обслуживания запросов клиентских приложений. Для создания сервера UDP используется объект DatagramSocket, который принимает объекты DatagramPacket от клиентов. Для создания сервера UDP необходимо выполнить следующие шаги:

• Создать сокет, используя объект DatagramSocket.

• Создать объект класса DatagramPacket и использовать метод receive() для получения сообщения клиента.

• Создать объект класса DatagramPacket и использовать ме-тод send() для отправки сообщение клиенту.

• Запустить сервер, вызывая конструктор класса сервера UDP в методе main().

Клиент UDP представляет собой приложение, которое использует протокол UDP для отправки запросов на сервер и получения ответов от серверного приложения. В клиентском UDP-приложении, необходимо создать объект класса DatagramSocket, который прини-

мает сообщения от сервера UDP, для чего необходимо выполнить следующие шаги:

1. Создать сокет, использующий объект класса DatagramSocket для установки соединения с сервером.

2. Создать объект класса DatagramPacket и использовать метод send() для отправки сообщения на сервер.

3. Создать объект класса DatagramPacket и использовать метод receive() для получения сообщений, отправленных сервером.

Следующий фрагмент кода можно использовать для создания объекта DatagramSocket:

```
try
{
DatagramSocket socket = new DatagramSocket(1501);
}
catch(SocketException se)
{
System.out.println("Error");
}
```

Объект DatagramPacket, который получает датаграммный пакет, содержит буфер для хранения датаграмм. Следующий фрагмент кода можно использовать для создания объект DatagramPacket, который принимает датаграммные пакеты:

```
try
{
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
}
catch(Exception e){
System.out.println("Error");
}
```

Следующий фрагмент кода можно использовать для отправки объекта DatagramPacket на заданный адрес:

```
try
{
DatagramPacket packet = new DatagramPacket(buffer, length, address, port);
socket.send(packet);
}
catch(Exception e)
{
System.out.println("Error");
}
```

Следующий фрагмент кода можно использовать для запуска сервера UDP:

```
public static void main(String args[]) throws Exception
{
/* Запуск сервера */
new UDPServer();
```

```
}
```

Следующий код позволяет создать UDPServer, который отображает полученные сообщения от клиента и отправляет клиенту ответные сообщения:

```
import java.io.*;
import java.net.*;
public class UDPServer
{       /* Объявляются переменные */
        DatagramSocket socket = null;
        BufferedReader in = null;
        String str = null;
        byte[] buffer ;
        DatagramPacket packet;
        InetAddress address;
        int port;
        /* Конструктор класса UDPServer */

        /* Создается объект DatagramSocket, который получает запросы клиента
        на номер порта 1501 */
        socket = new DatagramSocket(1501);
/* Вызывается метод call()*/
call();
}
public void call()
{
try
{
while (true)
{
buffer= new byte[256];
/* Инициализируется объект DatagramPacket */
packet = new DatagramPacket( buffer, buffer.length);
/* Посылается пакет датаграмм, используя метод receive()
класса DatagramSocket */
socket.receive(packet);
if(packet == null) break;
System.out.println("Request string for sending to client ");
try
{
/*Создается входной поток, который считывает данные с консо-ли*/
in = new BufferedReader(new InputStreamReader(System.in));
}
catch(Exception e)
{
System.out.println("Error : " + e);
}
str = in.readLine();
buffer = str.getBytes();
address = packet.getAddress();
port = packet.getPort();
packet = new DatagramPacket(buffer, buffer.length, address, port);
/* Посылается датаграммный пакет */
socket.send(packet);
```

```
}
/* Закрывается поток и сокет */
in.close();
socket.close();
}
catch(Exception e)
}
```

Следующий фрагмент кода можно использовать для создания объекта класса DatagramSocket для клиентского приложения:

```
try
{
DatagramSocket socket = new DatagramSocket();
}
catch(Exception e)
{
System.out.println("Error");
}
```

Следующий фрагмент кода используется для отправки объекта DatagramPacket на заданный сервер:

```
try
{
DatagramPacket packet = new DatagramPacket(buffer, length, address, port);
socket.send(packet);
}
catch(Exception e)
{
System.out.println("Error");
}
```

Следующий фрагмент кода можно использовать для создания объекта DatagramPacket, который принимает пакеты датаграмм от сервера:

```
try
{
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
str = new String(packet.getData());
System.out.println("Принятое сообщение : "+str);
}
catch(Exception e){
System.out.println("Ошибка");
}
```

Следующий код используется для создания класса UDPClient, который посылает и принимает сообщения от UDPServer:

```
import java.io.*;
import java.net.*;
public class UDPClient
{
/* Объявляются переменные */
static DatagramSocket socket;
static InetAddress address;
static byte[] buffer;
```

```
static DatagramPacket packet;
static String str, str2;
static BufferedReader br;
public static void main(String arg[]) throws Exception{
/* Создается входной поток, который читается с консоли */
br = new BufferedReader(new InputStreamReader(System.in));
while(true)
{
/* Создается новый объект DatagramSocket и связывается с портом по умол-чанию */
socket = new DatagramSocket();
address = InetAddress.getByName("127.0.0.1");
buffer = new byte[256];
packet = new DatagramPacket(buffer, buffer.length, address, 1501);
/* Посылается DatagramPacket на сервер */
socket.send(packet);
System.out.println("Sending request ");
packet = new DatagramPacket(buffer, buffer.length);
/* Принимается DatagramPacket от сервера */
socket.receive(packet);
/* Принимаются данные от объекта пакета датаграмм и*/
str = new String(packet.getData());
System.out.println("Received message : "+str.trim());
System.out.println("Do you want continue (Yes/No) : ");
str2 = br.readLine();
/* Выход из цикла while */
if(str2.equals("No")) break;
}
/* Закрывается объект сокет */
socket.close();
}
}
System.out.println("Error : " + e);
}
}
public static void main(String args[]) throws Exception
{
/* Запускается сервер */
new UDPServer();
}
}
```

Лабораторная работа №2.
Потоки и процессы в языке JAVA.

Содержание отчёта

1. Титульный лист.
2. Формулировка задачи.
3. Листинг программы.
4. Ответы на контрольные вопросы.

Контрольные вопросы

1. Понятие потока, процесса?
2. Принципы разбиения программы на процессы?
3. Как выполняются многопоточные вычисления?
4. Способы и средства записи параллельных программ?
5. Какие бывают проблемы при выполнении параллельных программ?
6. Какие способы решения вопроса 5 существуют ?

Задание

Разработать многопоточную программу на языке JAVA.

Теоретические и справочные сведения

Реализация многозадачности в Java [2].

Для создания многозадачных приложений Java необходимо воспользоваться классом **java.lang.Thread**. В этом классе определены все методы, необходимые для создания потоков, управления их состоянием и синхронизации.

Есть две возможности использования класса Thread.

Во-первых, можно создать собственный класс на базе класса Thread и переопределить метод run(). Новая реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, создаваемый класс, может реализовать интерфейс Runnable и реализовать метод run(), который будет работать как отдельный поток.

Создание подкласса Thread. При использовании этого способа для потоков определяется отдельный класс, например:

```
class myThread extends Thread  {
  public void run()   {
    // здесь можно добавить код, который будет
    // выполняться в рамках отдельного потока
  }
  // здесь можно добавить специализированный для класса код
}
```

Метод run**()** должен быть всегда переопределен в классе, наследованном от Thread. Именно он определяет действия, выполняемые в рамках отдельного потока. Если поток используется для выполнения циклической работы, этот метод содержит внутри себя бесконечный цикл.

Метод run() получает управление при запуске потока методом **start()** класса Thread**.** В случае апплетов создание и запуск потоков обычно осуществляется в методе start() апплета.

Остановка работающего потока раньше выполнялась методом **stop()** класса Thread**.** Обычно остановка всех работающих потоков, созданных апплетом, выполняется в методе stop() апплета. Сейчас не рекомендуется использование этого метода. Завершение работы потока желательно проводить так, чтобы происходило естественное завершение метода run. Для этого используется управляющая переменная в потоке.

Пример. Многопоточное приложение с использованием наследников класса Thread

```
// Поток для расчета координат прямоугольника
class ComputeRects extends Thread  {
   boolean going = true;
```

```
// конструктор получает ссылку на создателя объекта - апплет
   public ComputeRects(MainApplet parentObj)    {
      parent = parentObj;
   }
   public void run()    {
      while(going)   {
         int w = parent.size().width-1, h = parent.size().height-1;
         parent.RectCoordinates
         ((int)(Math.random()*w),(int)(Math.random()*h));
      }
   }
   MainApplet parent;              // ссылка на создателя объекта
}
// Поток для расчета координат овала
class ComputeOvals extends Thread   {
   boolean going = true;
    public ComputeOvals(MainApplet parentObj)    {
      parent = parentObj;
   }
   public void run()    {
      while(going)   {
         int w = parent.size().width-1, h = parent.size().height-1;
         parent.OvalCoordinates
         ((int)(Math.random()*w),(int)(Math.random()*h));
      }
   }
   MainApplet parent; // ссылка на создателя объекта
}

public class MainApplet extends JApplet   {
   ComputeRects m_rects = null;
   ComputeOvals m_ovals = null;
   int m_rectX = 0;    int m_rectY = 0;
   int m_ovalX = 0;   int m_ovalY = 0;
   // Синхронный метод для установки координат
   // прямоугольника из другого потока
   public synchronized void RectCoordinates(int x, int y)    {
      m_rectX = x; m_rectY = y;
      this.repaint();
   }
   // Синхронный метод для установки координат овала
   // из другого потока
   public synchronized void OvalCoordinates(int x, int y)    {
      m_ovalX = x; m_ovalY = y;
      this.repaint();
   }
   @Override
   public void start()    {
      super.start();
      // Запускаем потоки
      if (m_rects == null) {
            m_rects = new ComputeRects(this);   m_rects.start();
      }
      if (m_ovals == null) {
           m_ovals = new ComputeOvals(this);   m_ovals.start();
      }
   }
   @Override
   public void stop()    {
      super.stop();
```

```
        // Останавливаем потоки
        if(m_rects != null) m_rects.going = false;
        if(m_ovals != null) m_ovals.going = false;
    }
    public void paint(Graphics g)    {
        int w = this.getWidth(), h = this.getHeight();
        g.clearRect(0, 0, w, h);
        g.setColor(Color.red);
        g.fillRect(m_rectX, m_rectY, 20, 20);
        g.setColor(Color.blue);
        g.fillOval(m_ovalX, m_ovalY, 20, 20);
    }
    public static void main(String[] args)    {    }
}
```

Реализация интерфейса Runnable. Если нет возможности расширять класс Thread, то можно применить второй способ реализации многозадачности. Допустим, уже существует класс MyClass, функциональные возможности которого удовлетворяют разработчика. Необходимо, чтобы он выполнялся как отдельный поток.

```
class MyClass implements Runnable   {
  // код класса - объявление его элементов и методов
  // этот метод получает управление при запуске потока
  public void run()   {
    // здесь можно добавить код, который будет
    // выполняться в рамках отдельного потока
  }
}
```

Пример. Доработанное многопоточное приложение

```
// Поток для расчета координат линии
class ComputeLines implements Runnable  {
    boolean going = true;
    public ComputeLines(MainApplet parentObj)    {
        parent = parentObj;
    }
    public void compute()    {
        int w = parent.size().width-1, h = parent.size().height-1;
        parent.LineCoordinates
              ((int)(Math.random()*w),(int)(Math.random()*h),
              (int)(Math.random()*w), (int)(Math.random()*h));
    }
    MainApplet parent; // ссылка на создателя объекта
    public void run()    {
        while(going) { compute(); }
    }
}

public class MainApplet extends JApplet {
   . . .              // скопируйте из предыдущего примера
    ComputeLines m_lines = null;
    int m_lineX1 = 0, m_lineX2 = 0, m_lineY1 = 0, m_lineY2 = 0;
    // Синхронный метод для установки координат
    // прямоугольника из другого потока
    public synchronized void RectCoordinates(int x, int y)    {
        m_rectX = x; m_rectY = y;
```

```
        this.repaint();
    }
    // Синхронный метод для установки координат овала
    // из другого потока
    public synchronized void OvalCoordinates(int x, int y)    {
        m_ovalX = x; m_ovalY = y;
        this.repaint();
    }
    // Синхронный метод для установки координат линии
    // из другого потока
  public synchronized void LineCoordinates(int x1, int y1, int x2, int y2)  {
        m_lineX1 = x1; m_lineX2 = x2; m_lineY1 = y1; m_lineY2 = y2;
        this.repaint();
    }
    @Override
    public void start()    {
        super.start();
        // Запускаем потоки
        . . .        // скопируйте из предыдущего примера
        if (m_lines == null) {
            m_lines = new ComputeLines(this);
            new Thread(m_lines).start();
        }
    }
    @Override
    public void stop()    {
        super.stop();
        // Останавливаем потоки
        . . .        // скопируйте из предыдущего примера
        if(m_lines != null) m_lines.going = false;
    }
   public void paint(Graphics g)    {
       . . .        // скопируйте из предыдущего примера
       g.setColor(Color.green);
       g.drawLine(m_lineX1, m_lineX2, m_lineY1, m_lineY2);
   }
    public static void main(String[] args)    {   }
}
```

Класс Thread содержит несколько конструкторов и большое количество методов для управления потоков.

Некоторые методы класса Thread:

currentThread() - возвращает ссылку на выполняемый в настоящий момент объект класса Thread;

sleep() - переводит выполняемый в данное время поток в режим ожидания в течение указанного промежутка времени ;

start() - начинает выполнение потока. Это метод приводит к вызову соответствующего метода run();

run() - фактическое тело потока. Этот метод вызывает после запуска потока;

stop() - останавливает поток (устаревший метод);

isAlive() - определяет, является ли поток активным (запущенным и не остановленным);

suspend() - приостанавливает выполнение потока (устаревший метод);

resume() - возобновляет выполнение потока (устаревший метод). Этот метод работает только после вызова метода suspend();
setPriority() - устанавливает приоритет потока (принимает значение от MIN_PRIORITY до MAX_PRIORITY);
getPriority() - возвращает приоритет потока;
wait() - переводит поток в состояние ожидания выполнения условия, определяемого переменной условия;
join() - ожидает, пока данный поток не завершит своего существования бесконечно долго или в течении некоторого времени;
setDaemon() - отмечает данный поток как поток-демон или пользовательский поток. Когда в системе останутся только потоки-демоны, программа на языке Java завершит свою работу;
isDaemon() - возвращает признак потока-демона.

Во время своего существования поток может переходить во многие состояния, находясь в одном из нижеперечисленных состояний:

- Новый поток
- Выполняемый поток
- Невыполняемый поток
- Завершенный поток

Новый поток. При создании экземпляра потока этот поток приобретает состояние “Новый поток”:

```
Thread myThread=new Thread();
```

В этот момент для данного потока распределяются системные ресурсы; это всего лишь пустой объект. В результате все, что с ним можно делать - это запустить: myThread.start**()**;

Любой другой метод потока в таком состоянии вызвать нельзя, это приведет к возникновению исключительной ситуации.

Выполняемый поток. Когда поток получает метод start**()**, он переходит в состояние “Выполняемый поток”. Процессор разделяет время между всеми выполняемыми потоками согласно их приоритетам.

Невыполняемый поток. Если поток не находится в состоянии “Выполняемый поток”, то он может оказаться в состоянии “Невыполняемый поток”. Это состояние наступает тогда, когда выполняется одно из четырех условий:

1. *Поток был приостановлен*. Это условие является результатом вызова метода suspend(). После вызова этого метода поток не находится в состоянии готовности к выполнению; его сначала нужно “разбудить” с помощью метода resume(). Это полезно в том случае, когда необходимо приостановить выполнение потока, не удаляя его. Поскольку метод suspend не рекомендуется к использованию, приостановка потока должна выполняться через управляющую переменную.

2. *Поток ожидает***.** Это условие является результатом вызова метода sleep(). После вызова этого метода поток переходит в состояние ожидания в течении некоторого определенного промежутка времени и не может выполняться до истечения этого промежутка. Даже если ожидающий поток имеет доступ к процессору, он его не получит. Когда указанный промежуток времени пройдет, поток переходит в состояние “Выполняемый поток”. Метод resume() не может повлиять на процесс ожидания потока, этот метод применяется только для приостановленных потоков.
3. *Поток ожидает извещения*. Это условие является результатом вызова метода wait(). С помощью этого метода потоку можно указать перейти в состояние ожидания выполнения условия, определяемого переменной условия, вынуждая его тем самым приостановить свое выполнение до тех пор, пока данное условие удовлетворяется. Какой бы объект не управлял ожидаемым условием, изменение состояния ожидающих потоков должно осуществляться посредством одного из двух методов этого потока - notify() или notifyAll(). Если поток ожидает наступление какого-либо события, он может продолжить свое выполнение только в случае вызова для него этих методов.
4. *Поток заблокирован другим потоком*. Это условие является результатом блокировки операцией ввода-вывода или другим потоком. В этом случае у потока нет другого выбора, как ожидать до тех пор, пока не завершится команда ввода-вывода или действия другого потока. В этом случае поток считается невыполняемым, даже если он полностью готов к выполнению.

Завершенный поток. Когда метод run() завершается, поток переходит в состояние “Завершенный поток”.

Приоритеты потоков. В языке Java каждый поток обладает приоритетом, который оказывает влияние на порядок его выполнения. Потоки с высоким приоритетом выполняются чаще потоков с низким приоритетом. Поток наследует свой приоритет от потока, его создавшего. Если потоку не присвоен новый приоритет, он будет сохранять данный приоритет до своего завершения. Приоритет потока можно установить с помощью метода setPriority(), присваивая ему значение от MIN_PRIORITY до MAX_PRIORITY (константы класса Thread). По умолчанию потоку присваивается приоритет Thread.NORM_PRIORITY.

Потоки в языке Java планируются с использованием алгоритма планирования с фиксированными приоритетами. Этот алгоритм, по существу, управляет потоками на основе их взаимных приоритетов, кратко его можно изложить в виде следующего правила: в любой момент времени будет выполняться “Выполняемый поток” с наивысшим приоритетом. Как выполняются потоки одного и того же приоритета, в спецификации Java не описано.

группы потоков. Все потоки в языке Java должны входить в состав группы потоков. В классе Thread имеется три конструктора, которые дают возможность указывать, в состав какой группы должен входить данный создаваемый поток.

Группы потоков особенно полезны, поскольку внутри их можно запустить или приостановить все потоки, а это значит, что при этом не потребуется иметь дело с каждым потоком отдельно. Группы потоков предоставляют общий способ одновременной работы с рядом потоков, что позволяет значительно сэкономить время и усилия, затрачиваемые на работу с каждым потоком в отдельности.

В приведенном ниже фрагменте программы создается группа потоков под названием genericGroup (родовая группа). Когда группа создана, создаются несколько потоков, входящих в ее состав:

```
ThreadGroup genericGroup=new ThreadGroup("My generic group");
Thread t1=new Thread(genericGroup,this);
Thread t2=new Thread(genericGroup,this);
```

Если при создании нового потока не указать, к какой конкретной группе он принадлежит, этот поток войдет в состав группы потоков main (главная группа). Иногда ее еще называют текущей группой потоков. В случае апплета main может и не быть главной группой. Право присвоения имени принадлежит Web-браузеру. Для того чтобы определить имя группы потоков, можно воспользоваться методом **getName**() класса ThreadGroup.

Для того, чтобы определить к какой группе принадлежит данный поток, используется метод **getThreadGroup()**, определенный в классе Thread. Этот метод возвращает имя группы потоков, в которую можно послать множество методов, которые будут применяться к каждому члену этой группы.

Лабораторная работа №3.

Принципы организации грид-систем на платформе BOINC.

Содержание отчёта

1. Титульный лист.
2. Формулировка задачи.
3. Листинг программы.
4. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое GRID-вычисления?
2. Назначение системы BOINC?
3. Структура системы BOINC.
4. Как можно провести требуемые в задании рассчеты?
5. Какие подобные платформы существуют?

Задание

Используя источники:
http://boincstats.com
http://boinc.berkeley.edu
http://boinc.ru
Установите на компьютер программное обеспечение BOINC Manager. Подключитесь к проектам распределенных вычислений. Определите объем входящего и исходящего трафика. Определите время вычисления, затраты оперативной памяти и жесткого диска. Определите средний объем очков, выдаваемых 1 ядру процессора за 1 час работы. Определить объем вычислений, выполненный пользователем в рамках выбранного проекта.

Теоретические и справочные сведения

BOINC (***Berkeley Open Infrastructure for Network Computing***) — открытая программная платформа (университета Беркли для GRID вычислений) — некоммерческое межплатформенное ПО для организации распределенных вычислений. Используется для онганизации добровольных вычислений.

Состоит из серверной и клиентской частей. Платформа является доступной для сторонних проектов. На сегодняшний день BOINC является универсальной платформой для проектов в области математики, молекулярной биологии, медицины, астрофизики и климатологии. BOINC даёт исследователям возможность задействовать огромные вычислительные мощности персональных компьютеров со всего мира.

Платформа работает на различных операционных системах. BOINC распространяется под лицензией GNU Lesser General Public License, как свободное программное обеспечение с открытым исходным кодом.

Серверная часть состоит из HTTP-сервера с веб-сайтом проекта, базы данных MySQL и набора демонов (генератор заданий, планировщик, валидатор, ассимилятор результатов.

BOINC-клиент позволяет участвовать одновременно в нескольких проектах с помощью одной общей программы управления (boinc или boinc.exe). Для визуализации процесса управления BOINC-клиентом можно использовать поставляемую по умолчанию официальную программу-менеджер (boincmgr или boincmgr.exe).

Список используемых источников

1. Дубаков А.А. Сетевое программирование: учебное пособие / А.А. Дубаков– СП: НИУ ИТМО, 2013. – 248 с.
2. Технология разработки объектно-ориентированных программ на JAVA: учебно-методическое пособие / И. А. Васюткина. - Новосибирск : НГТУ, 2012. - 150 с.
3. Биллиг, В. А. Параллельные вычисления и многопоточное программирование [Электронный ресурс] : учебное пособие / В. А. Биллиг. - 2-е изд., испр. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 311 с.- Режим доступа: http://biblioclub.ru
4. Левин М. П. Параллельное программирование с использованием OpenMP [Электронный ресурс] : учебное пособие / Михаил Петрович Левин. - М.: Бином. Лаборатория знаний : Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. – 120с. //Режим доступа – http//biblioclub.ru
5. Борзов Д. Б. Параллельные вычислительные системы (архитектура, принципы размещения задач) [Текст] : монография / Д. Б. Борзов, В. С. Титов ; Курский государственный технический университет. - Курск : КурскГТУ, 2009. - 159 с.
6. Воеводин, В. В. Параллельные вычисления [Текст] : учебное пособие / В. В. Воеводин, Вл. В. Воеводин. - СПб. : БХВ-Петербург, 2002. - 608 с.