Week 8 Lab - Introduction to Threads & Multithreaded TCP

COSC244

1 Assessment

This lab is worth 1%. The marks are awarded for the written answers to the preparation questions (worth 0.5%) and completing two programming exercises (worth 0.5%).

The preparation questions should be answered **before** coming to the lab. They should be stored electronically in a plain text file with a .txt suffix in your ∼/244/08 directory. They will be checked by the demonstrators at the start of the lab.

Make sure you get your work signed off by a demonstrator before leaving the lab.

2 Introduction to Threads

In the last lab, you ended up with a program that allowed two-way communication provided the sender and receiver alternated. Pseudocode for the sender and receiver is shown below.

```
1 // Sender pseudocode
2
3 while (oceans have waves) {
4    get a line from the keyboard
5    send the line over the socket
6    get a line from the socket
7    display the line
8 }
```

```
1 //Receiver pseudocode
2
3 while (oceans have waves) {
4    get a line from the socket
5    display the line
6    get a line from the keyboard
7    send the line over the socket
8 }
```

As you can see from the pseudocode, the sender first waits on a line from the keyboard. Even if something were to arrive on the socket, nothing would be done with it until a line were typed on the keyboard and that line sent over the socket. Then the program waits for something from the socket and displays it on arrival. The program loops forever. The opposite sequence of events applies to the receiver.

Notice that lines 4 & 5 of the sender code are identical to lines 6 & 7 of the receiver. Lines 6 & 7 of the sender code are identical to lines 4 & 5 of the receiver.

That was a good first step. However we want to be able to send something to the other end at any time instead of having the 'lock-step' synchronisation between sender and receiver.

The way to accomplish this goal is to use threads. A thread is a unit of program execution. An application can have multiple threads of execution running concurrently. That is, it can start several threads. Each thread is independent of the other threads unless they need to share data.

In our case, we want to be able to read from the keyboard and read from a socket at the same time. We can do so if we use one thread for reading from the keyboard and outputting to a socket and a second thread for reading from a socket and printing to the screen.

3 A Thread Example

Let's begin by looking at a simple threaded application that prints out the name of a fruit at timed intervals. In the lab, you will copy some code which we have provided into your own directory, and run it to see what is happening. For now, we will examine the code.

Here is the code for the *Fruit* class that expects to run as a thread.

```
1  //Fruit.java
2
3  public class Fruit extends Thread {
4
5     private String name;
6     private int pauseSeconds;
7
8     public Fruit (String name, int pauseSeconds) {
9        this.name = name;
10       this.pauseSeconds = pauseSeconds;
11    }
12
13    public void run() {
14       for (int i = 0; i < 10; i++) {
15          try {
16             Thread.sleep(pauseSeconds * 1000); // convert to milliseconds
17          } catch (InterruptedException e) {
18             System.err.println(e);
19          }
20          System.out.printf("%02d ", System.currentTimeMillis() / 1000 % 60);
21          System.out.println(name + " " + i);
22       }
23    }
24 }
```

Lines 8 - 11 are the constructor for the class. Its parameters are a string and an integer. It stores the arguments into two variables declared in lines 5 & 6.

Threads need a `run()` method which is executed when the thread *starts*. We will see how to start a thread soon.

Lines 13 - 23 are the `run()` method for the *Fruit* class. It is a loop that repeats 10 times. The `sleep()` method causes the thread to sleep for the number of seconds specified when the thread is created. It can throw an *InterruptedException* so we enclose the call to `sleep()` in the `try-catch` block.

Once the thread is done sleeping, lines 20 & 21 output a time value, the name of the fruit specified when the thread was created, and the current loop number. After 10 iterations the thread terminates.

Here is the code for a class that will create and start 3 threads. It is called *Example1*

```
1  //Example1.java
2
3  public class Example1 {
4
5     public static void main(String[] args) {
6        Fruit f1 = new Fruit("Banana", 3);
7        Fruit f2 = new Fruit("Apple", 2);
8        Fruit f3 = new Fruit("Orange", 4);
9        f1.start();
10       f2.start();
11       f3.start();
12    }
13
14 }
```

This class has the `main()` method. Lines 6 - 8 create three instances of a *Fruit* class, giving each one a different name and number of seconds to sleep. Lines 9 - 11 call the `start()` method which invokes the `run()` method of the corresponding thread.

When you run this program in the lab, almost always the output will be lines consisting of a number, a fruit name, and the loop number. The fruits will appear in different orders as time goes on. Occasionally a line might appear to be jumbled. Another program, called *Example2.java*, starts six threads and it usually has several jumbled lines. What is happening with the jumbled lines is that one thread wakes up and starts printing its line. During the output, another thread wakes up and pre-empts the first thread. The second thread prints its line (on the same line) and then goes to sleep returning CPU control to the first thread which completes printing its line.

4 Preparation

Let's begin by looking at the *Thread* class.

4.1 Thread

Open the course web page, click on *Resources*, click on *Java API Documentation*, click on *Thread* in the package *java.lang*. Read the introductory material down to where *Nested Class Summary* starts.

Q1 What two ways can you use to create a thread? Explain each in a couple of sentences.

Q2 Whichever way you choose to create a thread, what is the name of the method that must be part of your thread's implementation?

Q3 What method does a program call to start a thread running?

4.2 Multithreaded TCPExample

The basic program from Lab 7, *TCPExample.java*, is given at the end of this document. We will break it into three classes: *Client*, *Server*, and *ReadWriteThread*. The code for the *ReadWriteThread* class is given below.

```
1  // ReadWriteThread.java
2
3  import java.io.*;
4
5  public class ReadWriteThread extends Thread {
6
7     private BufferedReader input;
8     private PrintWriter output;
9
10    public ReadWriteThread(InputStream input, OutputStream output) {
11       this.input = new BufferedReader(new InputStreamReader(input));
12       this.output = new PrintWriter(output, true);
13    }
14
15    public void run() {
16       try {
17          String line;
18          while ((line = input.readLine()) != null) {
19             output.println(line);
20          }
21       } catch (IOException e) {
22          e.printStackTrace();
23       }
24    }
25 }
```

The constructor for this class is given an *InputStream* and an *OutputStream* as parameters. If you don't remember what these are, review them from Lab 7's preparation. The constructor assigns these streams to `input` and `output` as a *BufferedReader* and a *PrintWriter*, respectively. The input could be either *System.in* or the InputStream associated with a socket. The output could be either *System.out* or the OutputStream associated with a socket.

The `run()` method basically reads a line as a *String* from the input and prints it to the output in an infinite loop.

The *Server* code is given below.

```
1  // Server.java
2
3  import java.net.*;
4
5  public class Server {
6
7     public static void main(String[] args) {
8        try {
9           int port = Integer.parseInt(args[0]);
10          ServerSocket serverSocket = new ServerSocket(port);
11          System.err.println("Waiting for a client to connnect");
12          Socket socket = serverSocket.accept();
13          System.err.println("Accepted connection on port " + port);
```

```
14          new ReadWriteThread(System.in, socket.getOutputStream()).start();
15          new ReadWriteThread(socket.getInputStream(), System.out).start();
16       } catch (Exception e) {
17          e.printStackTrace();
18          System.err.println("\nUsage: java Server <port>");
19       }
20    }
21
22 }
```

Compare this code to the code path executed in *TCPExample.java* when a server is started.

Q4 What lines in *Server.java* are different?

Q5 Explain what lines 14 & 15 do.

Q6 Would it matter if we reversed lines 14 & 15? Why or why not?

Q7 Write the code for *Client.java* which will work with *Server.java* using *ReadWriteThread.java*. HINT: Study the code path executed in *TCPExample.java* when a client is started.

5 Programming Exercises

5.1 Trying out the multi-threaded fruit printing programs

Create and/or change to a directory you wish to work in for this lab. Type the following to copy files for this lab.

```
cp -r /home/cshome/coursework/244/start/08 .
```

Don't forget the period at the end. Compile *Example1.java* and *Example2.java* by typing:

```
javac Example1.java
```

```
javac Example2.java
```

Run *Example1* by typing:

```
java Example1
```

Notice how the fruits appear in random order after a period of time. Run *Example2* by typing:

```
java Example2
```

Notice how several lines appear to be jumbled.

Q8 Why is this happening?

5.2 Program 1

Type in the code for *Server.java* and *ReadWriteThread.java* (yes, it's a very good idea to type it in rather than just using copy and paste). Then compile them by typing the command:

```
javac Server.java
```

Now type in the *Client.java* that you created in Question 7 above and compile it.

Start the server first by typing:

```
java Server 7777
```

It should startup with a message about waiting for a client to connect. Start the client by typing:

```
java Client 7777 localhost
```

You should be able to type into the client and server in random order. The lines you type should appear on the other side. If not, you need to debug your client code.

You terminate the client and server type typing Ctrl-C in each window.

5.3 Program 2

Copy *Server.java* to another file called *MultiServer.java*. Modify the code so it corresponds to the following:

```
1  // MultiServer.java
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6
7  public class MultiServer extends Thread {
8
9     private static List<PrintWriter> clients = new LinkedList<PrintWriter>();
10
11    public static void main(String[] args) {
12       try {
13          int port = Integer.parseInt(args[0]);
14          ServerSocket serverSocket = new ServerSocket(port);
15          new MultiServer().start();
16          System.err.println("Waiting for a client to connect");
17          while (true) {
18             Socket socket = serverSocket.accept();
19             synchronized(clients) {
20                clients.add(new PrintWriter(socket.getOutputStream(), true));
21             }
22             System.err.println("Accepted connection on port " + port);
23             new ReadWriteThread(socket.getInputStream(), System.out).start();
24          }
```

```
25      } catch (Exception e) {
26         e.printStackTrace();
27         System.err.println("\nUsage: java MultiServer <port>");
28      }
29   }
30
31   public void run() {
32      Scanner stdin = new Scanner(System.in);
33      while (stdin.hasNextLine()) {
34         String line = stdin.nextLine();
35         synchronized(clients) {
36            for (PrintWriter client : clients) {
37               client.println(line);
38            }
39         }
40      }
41   }
42
43 }
```

Compile and run the program (starting the server first as usual). You should be able to connect more than one client at a time. You will need a different terminal window for each client.

Each line you type into a client appears in the server's window. Each line you type into the server appears in each client's window. You terminate the programs by typing Ctrl-C in each window.

Before leaving the lab, show your programs to a demonstrator.

To be continued next week . . .

Source code for TCPExample.java.

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4
5  public class TCPExample {
6
7     private PrintWriter output;
8     private BufferedReader input;
9
10    public TCPExample(Socket socket) throws Exception {
11       output = new PrintWriter(socket.getOutputStream(), true);
12       input =
13          new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

```
14    }
15
16    public void startReceiving() throws Exception {
17       String line;
18       while ((line = input.readLine()) != null) {
19          System.out.println(line);
20       }
21    }
22
23    public void startSending() {
24       Scanner stdin = new Scanner(System.in);
25       System.err.println("Please enter data here");
26       while (stdin.hasNextLine()) {
27          output.println(stdin.nextLine());
28       }
29    }
30
31    public static void main(String[] args) {
32       Socket socket = null;
33       try {
34          int port = Integer.parseInt(args[0]);
35          if (args.length > 1) {
36             socket = new Socket(args[1], port);
37             System.err.println("Connected to " + args[1] + " on port " + port);
38             TCPExample example = new TCPExample(socket);
39             example.startSending();
40          } else {
41             ServerSocket serverSocket = new ServerSocket(port);
42             System.err.println("Waiting for someone to connect");
43             socket = serverSocket.accept();
44             System.err.println("Accepted connection on port " + port);
45             TCPExample example = new TCPExample(socket);
46             example.startReceiving();
47          }
48       } catch (Exception e) {
49          e.printStackTrace();
50          System.err.println("\nUsage: java TCPExample <port> [host]");
51       }
52    }
53 }
```