|  |
| --- |
| https://www.dit.ie/media/logo/DIT_logocol2013.jpg |
| **MULTITHREADING IN JAVA** |
| **Distributed Systems** |
| **Networking Applications and Services** |
| **Junaid Malik** |
|  |

**C11452772**

|  |
| --- |
|  |

**08/Oct/2014**

**Introduction**

Threads in a distributed system a very important they provide concurrent execution within a process and help reduce the impact of blocking operations in a network. An important characteristic is a blocking call which prevents an entire process being blocked. This exploits a parallelism to attain a high performance. A typical design is to organize the server as a single dispatcher with multiple threaded workers.

In a multithreaded program, threads are obtained from the pool of available ready-to-run threads and run on the available system CPUs. The OS can move threads from the processor to either a ready or blocking queue, in which case the thread is said to have "yielded" the processor. Alternatively, the Java virtual machine (JVM) can manage thread movement -- under either a cooperative or preemptive model -- from a ready queue onto the processor, where the thread can begin executing its program code.

**Objective**

Aim of this lab was to study and examine the use of concurrent programming and design to develop java multi-threaded applications

**Part 1 Exercise: Extending Thread class**

Modify **ExtendingThreadTest.java** as follows:

1. Create and start another thread.
2. Set the name of the thread as "My Own"
3. Build and run the application.
4. In your lab report explain why the threads are interleaved.
5. When you have finished part 4 of this lab come back to this code and add the necessary code to prevent each thread's run method interleaving (hint: use thread synchronization). Outline in your report what code you added to synchronize the methods.

a) b)

System.out.println("Creaate FirstThreadedClass object instance..");

FirstThreadedClass myOwn = new FirstThreadedClass("My Own");

System.out.println("\nCalling start() method of " + myOwn.getName() + " thread");

myOwn.start();

(c)

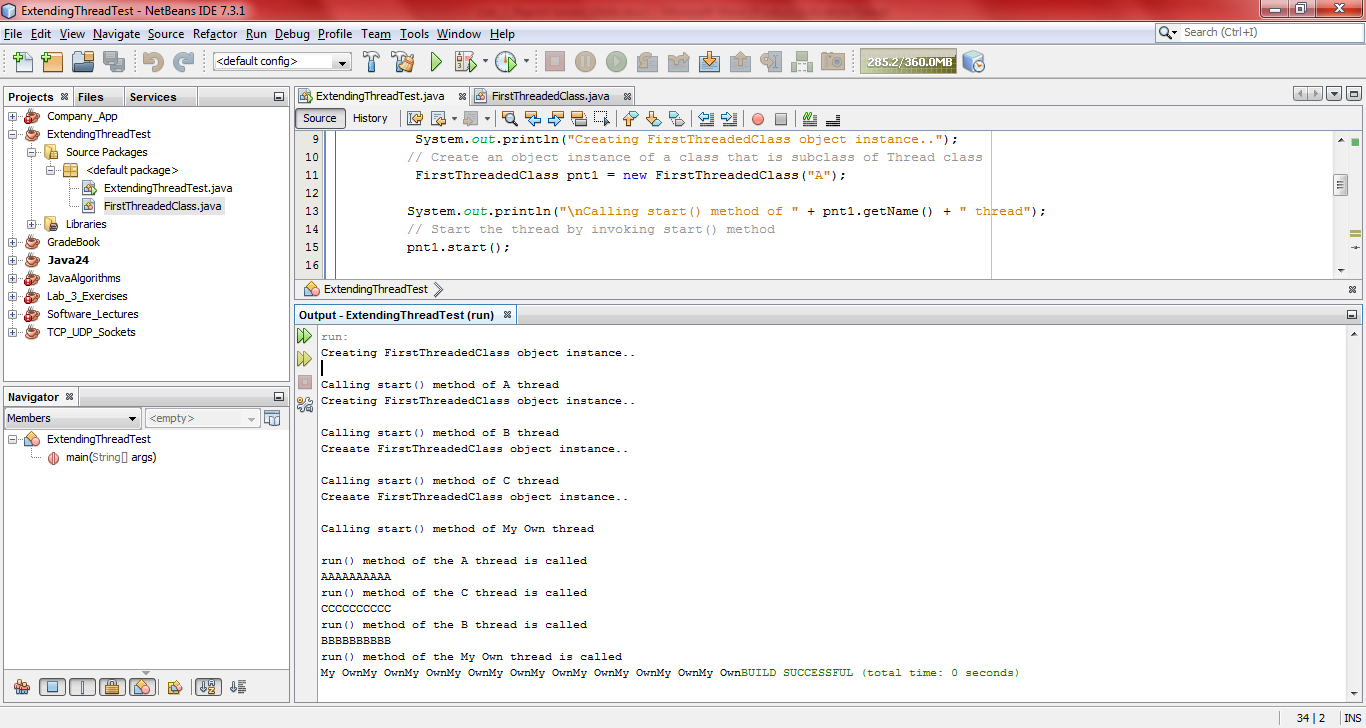


Figure : This is the built application with the interleaving threads

(d). The above interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap one another.

*(f)When you have finished part 4 of this lab come back to this code and add the necessary code to prevent each thread's run method interleaving (hint: use thread synchronization(). Outline in your report what code you added to synchronize the methods.*

Threads communicate primarily by sharing access to fields and the objects reference fields that they refer to. This form of communication is very efficient, but makes two kinds of errors are made possible: 1) *thread interference* and 2) *memory consistency errors*. The tool needed to prevent these errors is synchronization. A Java synchronized code will only be executed by one thread at a time the code is modified by adding synchronize after the class access modifier. The synchronized keyword tells the java virtual machine to only run one thread at a time.

@Override

public synchronized void run() {

System.out.println("\nrun() method of the " + this.getName() + " thread is called" );

for (int i = 0; i < 150; i++) { System.out.print(this.getName());

}

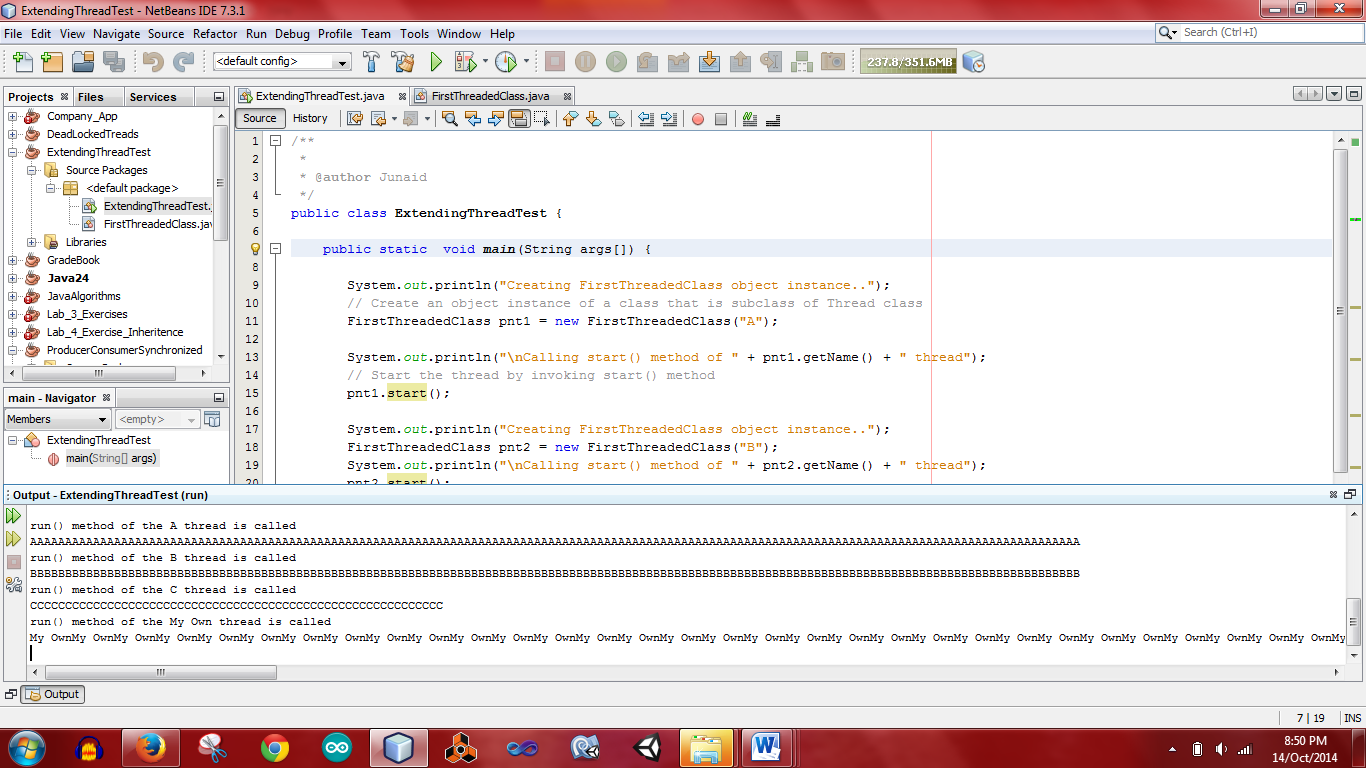


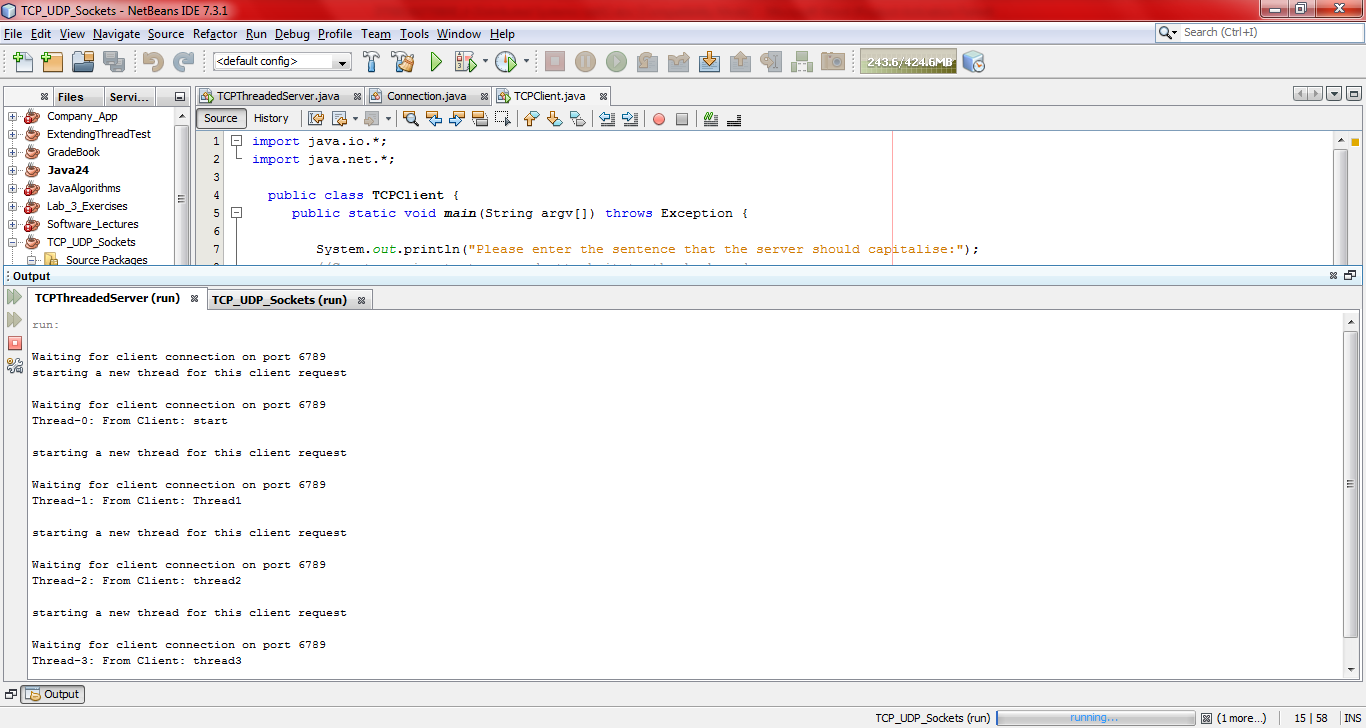
Figure : This is the output of the threads as they are now synchronized: Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors

**Part 2 Exercise: TCP Multithreaded Server**

(a).Explain in your lab report why you think it is a good idea to have a multi-threaded server to handle client requests.

Multi-threaded servers allow us to increase throughput. If we assume a server has a pool of threads each of which repeatedly removes a request from a queue of received requests and processes it. If a single thread has to perform all processing. Then the time for handling each request will be high and if any new request messages arrive while server is handling a request it is then queued at the server port. If we consider a server with two threads the then it can process a second request from client while one thread is blocked. Throughput can also be increased by using a shared-memory multiprocessor to decrease processor bottleneck. A shared execution environment can be implemented in shared memory and the multiple threads can be scheduled to run on the multiple processors.

It is therefore a good idea to have a multithreaded server as it includes improved performance and better structure. It is easier to start a new thread then to start a new process also having a single-thread server prohibits simple scaling to a multiprocessor system. Network latency is hidden from the user as requests are anticipated while a previous is being applied. Structure is also improved as servers have high input/output demands. Threading reduces impact of blocking calls. Multithreaded programs tend to be smaller and easier to understand due to simplified flow control.



The above demonstrates Multithreaded Server using TCP Client to send messages to the TCP server which in turn created multiple threads of each message that was received. The thread number was incremented on the server output for each new client application that was created.

## Part 3: Synchronization of threads

In the third part of this lab exercise I learned to do synchronization among threads to prevent thread interference and memory consistency errors.

1. Explain in your lab report what a synchronized method is and how it works.
2. What code did you have to add to prevent the threads interleaving?

(a )

The synchronized method allows only one thread to execute the method block at a time making it thread safe. Threads communicate primarily by sharing access to fields and the objects reference fields that they refer to. This form of communication is very efficient, but makes two kinds of errors are made possible: (1) thread interference and (2) memory consistency errors. The tool needed to prevent these errors is synchronization. A Java synchronized code will only be executed by one thread at a time The code is modified by adding synchronize after the class access modifier. The synchronized keyword tells the java virtual machine to only run one thread at a time. Synchronized methods enable a simple method to prevent thread interference and memory consistency errors, if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

(b) The print method of TwoStrings class was marked as synchronized. If a method is specified with the keyword synchronized, it is only executed by one thread at a time.  
Synchronized methods are useful in these situations where methods are executed concurrently, so that these can be intercommunicate manipulate the state of an object in ways that can corrupt the state.

In this program, the method **print(String str1, String str2)**  is synchronized and will be shared by both thread's objects at the time of the program execution. Therefore only one thread can access the method and process it until all statements of the method are executed.

public class TwoStrings {

// This method is synchronized

static synchronized void print(String str1, String str2) {

System.out.print(str1);

try { Thread.sleep(500);}

catch (InterruptedException ie) {}

System.out.println(str2);

}

}

## Part 4: Inter-thread communication

**Part 4 Exercise**

1. Explain in your lab report how the producer and consumer have been synchronized to overcome the problem encountered in part 4.1.

Wait and notify methods in Java are used for inter-thread communication if one thread tells something to another thread It uses notify() and notifyAll() these belong to the object class. In our case we have producer and consumer where one thread produces and tells another thread that there is a shared object, the consumer then picks this object to do its job. There is a shared queue, once a object is produced consumer thread has to be notified and similarly after consumption producer thread needs to be notified. This inter-thread communication is achieved using wait and notifies methods.

The following commands are used to make inter-thread communication possible.

|  |  |
| --- | --- |
|  | public void wait() Causes the current thread to wait until another thread invokes the notify(). |
|  | public void notify() Wakes up a single thread that is waiting on this object's monitor. |
|  | public void notifyAll() Wakes up all the threads that called wait( ) on the same object. |

public synchronized int get() {

while (dataAvailable == false) {

try {

wait();

} catch (InterruptedException e) { }

}

System.out.format("Consumer got: %d%n", data);

notifyAll();

dataAvailable = false;

return data;

## Part 5: Thread Deadlock

In this exercise, I learned how deadlock can occur in multithreaded applications. Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because thesynchronizedkeyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

**Part 5 Exercise**

1. Explain in your lab report what the problem is with the code.
2. Change the code to fix the problem. (Hint: the order of the locks is important to prevent deadlock).

(a) The problem with this code is that Thread 1 is holding resource T1 and needs another resource T2 to finish execution but T2 is locked by thread 2, it holds an exclusive lock that the other threads needs. Therefore they become stuck and are deadlock. The deadlock is occurring because the threads are trying to acquire the same locks in different orders. To avoid the deadlock the program I arranged the resources in a well-defined order. Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

(b) I changed the order of the locks to a sequential order to avoid a deadlock situation.

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (**Lock1**) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (**Lock1**) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (**Lock2**) {

System.out.println("Thread 2: Holding lock 2...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 1...");

synchronized (**Lock2**) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

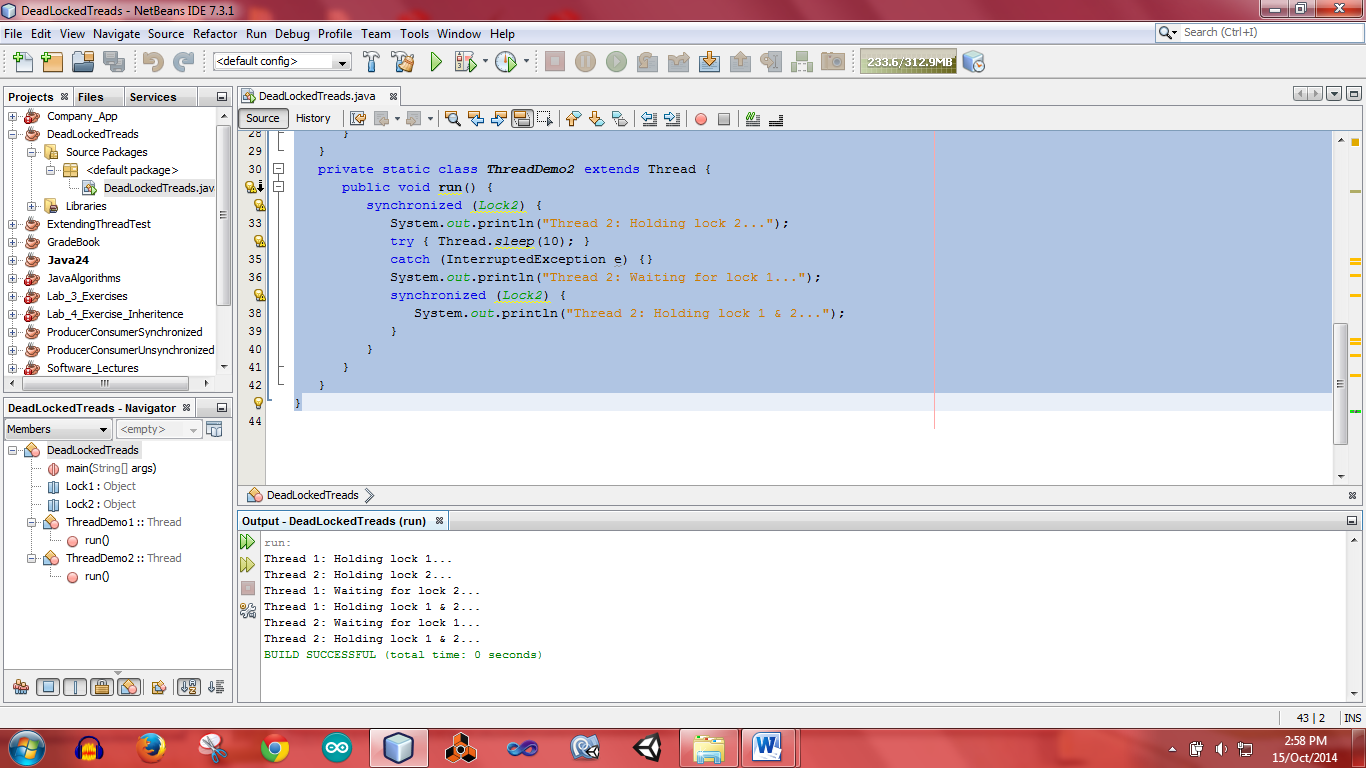


Figure 3 The above figure display output of the program with arranged locks so no deadlock is occurring.