Lab 5 – Introduction to Threads

(Written by Dr. Scott Barlowe*)

In traditional environments, programs are executed in order and at one step at a time. Complex or computationally intense programs would occupy resources for too long. One solution to this dilemma is to make programs multi-threaded. Architectures that support multi-threaded programs allow independent parts of a program to be executed simultaneously. The program is separated into individual threads which are allocated computational work. Threads can work as an isolated unit. Threads can also report to another thread (such as a main thread) and its results can be placed together with the reports from other threads. A thread has an ID, a program counter, a register set, and a stack. A thread can also share resources with other threads belonging to the same process. Perhaps the most difficult aspect of multithreaded program design is that we can not usually predict in what order the threads will execute. Thread execution order is determined by the thread pool scheduler.

There are two primary ways of creating threads in Java. One way is to subclass Java's Thread class. However, this creates a rigid hierarchy which may become problematic as the programs become more complex. The other way is to implement Java's **Runnable** class and use that to create threads. This lab will use the latter.

The thread capabilities required by these labs and the last programming assignment are introductory and are not intended as a comprehensive coverage of threads. You will learn much more about threads and concurrency in CS 370, Operating Systems.

The following shows an example of how to create a thread where **Worker** implements **Runnable.** The class implementing Runnable is instantiated and then passed to the **Thread**'s constructor.

Worker w0 = new Worker(some arguments could go here);

Thread th0 = new Thread(w0);

To begin the thread, call the thread's **start()** method.

th0.start();

The **start()** method invokes the **run()** method that should have been implemented in the class implementing **Runnable**. After the **run()** method has finished executing, the thread terminates.

A request can be made to interrupt the thread before the **run()** method has finished. This request is made with the **interrupt()** method. This is sometimes used along with the thread's **isInterrupted()** method to control a variable length loop (perhaps inside of the thread's run() method). An example of the **interrupt()** method being called is shown below.

th0.interrupt();

The **sleep()** method pauses execution of the calling object for a specified amount of time. This gives other threads the opportunity to execute.

th0.sleep(1000); // Sleep the thread for 1000 msecs.

Based on program conditions and/or the methods called, threads can occupy one of the following states: new, runnable, running, non-runnable, and terminated.

Build the Program

This program will create and start multiple threads. Each thread will be responsible for summing one row in a two-dimensional matrix and then summing the individual results of the threads together. We will add functionality so that the program ensures that the last thread has finished before summing the individual thread results.

*Some introductory material taken from Operating System Concepts with Java by Silberschatz, Galvin, and Gagne (6th edition).
Some parts of the code were inspired by an example found at https://www.cs-book.com/OS9/java-dir/4.pdf

```
wget https://agora.cs.wcu.edu/~sbarlowe/cs465/threadOne/MatrixAdder.java

wget https://agora.cs.wcu.edu/~sbarlowe/cs465/threadOne/SumAccumulator.java

wget https://agora.cs.wcu.edu/~sbarlowe/cs465/threadOne/AdderWorker.java
```

MatrixAdder.java and **AdderWorker.java** are partially completed files which will be finished in the steps below. The **SumAccumulator** class is responsible for accumulating (or adding) the results of each individual thread into one sum. This class is complete.

AdderWorker.java

1. Modify the **AdderWorker** class so that it implements **Runnable**.

2. Create private variables with the following data types:

```
SumAccumulator  // a variable to add the sums as the threads finish

int[ ]          // an array to hold a row of the matrix created in MatrixAdder.java

int             // an integer to hold an assigned ID
```

3. Create a constructor that accepts arguments to set the private fields and initializes the fields to those values.

4. Create a public **run()** method that accepts zero parameters and has a **void** return type. Within this method

   -Create a local integer variable and set it to zero.

   -Use a loop to sum the values in the array and store the result in the local integer.

   -Print a message in the form *Thread 22 added 41759 nums* → *41759* where *22* is our thread ID, the first *41759* is the length of the array given to thread 22, and last *41759* is the sum of the array. (The number of elements added and the sum are the same because we populated the matrix with ones. You may change how you populate the array if you wish.)

   -Call the **SumAccumulator** field's **setSum()** method and pass to it the sum of the array elements found in the previous steps.

MatrixAdder.java

The code already in **MatrixAdder.java** creates a 2d array and populates it with all ones. Complete the following steps after the code that is already there.

1. Create an **ArrayList** of type **Thread**.

2. Create a new **SumAccumulator** object.

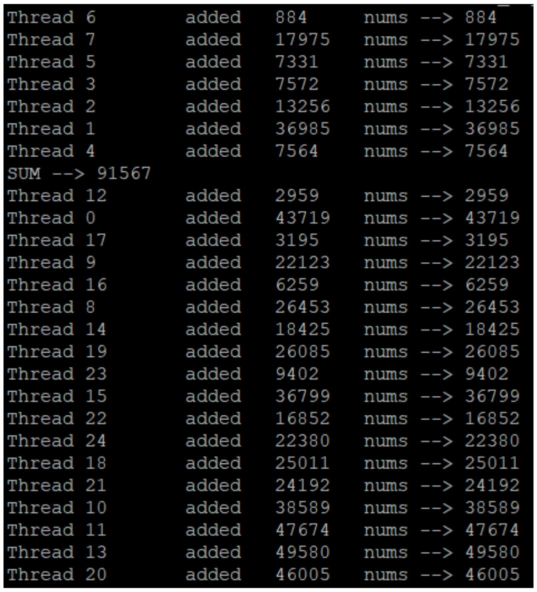
3. Create a loop that iterates once for each row in the matrix. Inside the body of the loop, instantiate a new **Thread** for each matrix row and store the new **Thread** in the **ArrayList**. Pass a new **AdderWorker** object to each new **Thread** constructor. Each **AdderWorker** should have its ID set to the loop iteration number, have the array set to the corresponding row in the matrix, and the **SumAccumulator** object set to the one created earlier (i.e. pass the Sum object created in step 2 to each constructor).

4. Create a new loop that will start each thread.

5. Place a single **System.out.println** that outputs the value returned by the SumAccumulator's **getSum()** method.

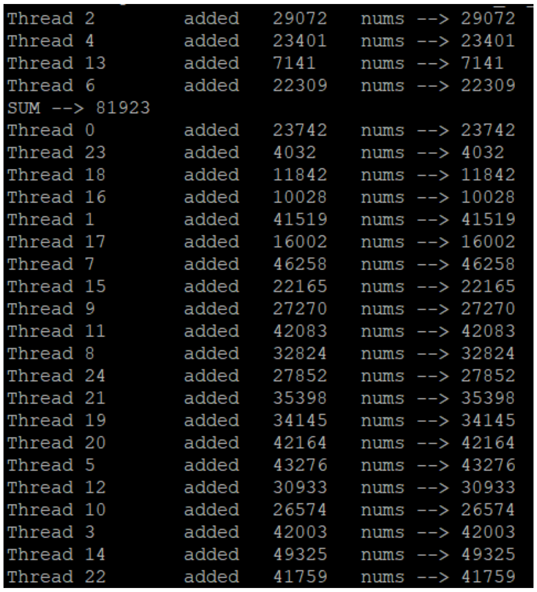
6. Run the program 10-15 times. You should get an output similar to that shown below, but varying with each program execution. Note the change in the order of thread execution and output of the sum.

One Execution



```
Thread 6        added   884     nums --> 884
Thread 7        added   17975   nums --> 17975
Thread 5        added   7331    nums --> 7331
Thread 3        added   7572    nums --> 7572
Thread 2        added   13256   nums --> 13256
Thread 1        added   36985   nums --> 36985
Thread 4        added   7564    nums --> 7564
SUM --> 91567
Thread 12       added   2959    nums --> 2959
Thread 0        added   43719   nums --> 43719
Thread 17       added   3195    nums --> 3195
Thread 9        added   22123   nums --> 22123
Thread 16       added   6259    nums --> 6259
Thread 8        added   26453   nums --> 26453
Thread 14       added   18425   nums --> 18425
Thread 19       added   26085   nums --> 26085
Thread 23       added   9402    nums --> 9402
Thread 15       added   36799   nums --> 36799
Thread 22       added   16852   nums --> 16852
Thread 24       added   22380   nums --> 22380
Thread 18       added   25011   nums --> 25011
Thread 21       added   24192   nums --> 24192
Thread 10       added   38589   nums --> 38589
Thread 11       added   47674   nums --> 47674
Thread 13       added   49580   nums --> 49580
Thread 20       added   46005   nums --> 46005
```

Another Execution



```
Thread 2        added   29072   nums --> 29072
Thread 4        added   23401   nums --> 23401
Thread 13       added   7141    nums --> 7141
Thread 6        added   22309   nums --> 22309
SUM --> 81923
Thread 0        added   23742   nums --> 23742
Thread 23       added   4032    nums --> 4032
Thread 18       added   11842   nums --> 11842
Thread 16       added   10028   nums --> 10028
Thread 1        added   41519   nums --> 41519
Thread 17       added   16002   nums --> 16002
Thread 7        added   46258   nums --> 46258
Thread 15       added   22165   nums --> 22165
Thread 9        added   27270   nums --> 27270
Thread 11       added   42083   nums --> 42083
Thread 8        added   32824   nums --> 32824
Thread 24       added   27852   nums --> 27852
Thread 21       added   35398   nums --> 35398
Thread 19       added   34145   nums --> 34145
Thread 20       added   42164   nums --> 42164
Thread 5        added   43276   nums --> 43276
Thread 12       added   30933   nums --> 30933
Thread 10       added   26574   nums --> 26574
Thread 3        added   42003   nums --> 42003
Thread 14       added   49325   nums --> 49325
Thread 22       added   41759   nums --> 41759
```

In some cases, the sum may have been output before all of the threads had finished their calculations. That's not what we want. A solution to this is to use the **join()** method. The join method pauses the current thread (not the referenced thread) until the referenced thread is finished executing.

6. Insert the following code just before the **System.out.println()** method in **MatrixAdder.java** (substitute your variable names for the ones below):

```
try{
        for(int i = 0; i < length of the arraylist; i++){
                threadList.get(i).join();
        }
}
catch(InterruptedException e ){
}
```

Execute the Program

7. Compile the program and run it multiple times. Although the order of the threads may still be out of order, the sum should always be retrieved last – after all threads have finished.

Teams

You may work in teams of one or two people. If working in pairs, make sure each partner's name is in the file header comments of ALL files. No other documentation is needed. You are not allowed to share code with any other teams. No credit will be given for programs with no or little functionality.

Submit

Submit your work as program 500 by 11:59 pm on November 22 with the command below. You must also work with the same partner as you are for project 3. You must also demo this lab at the same time as the November 22 milestone for project 3.

handin.465.1   500   *.java

Lab 6 – Interrupting Threads

Download **AnotherAdder.java** and **AnotherAdderWorker.java**. These are partially complete files which will be finished in the steps below. This program will run two threads, which will execute a while loop until they are interrupted by the main program.

Build the Program

To get started, download the two files with the commands below.

```
wget https://agora.cs.wcu.edu/~sbarlowe/cs465/threadTwo/ThreadInterrupted.java

wget https://agora.cs.wcu.edu/~sbarlowe/cs465/threadTwo/AnotherAdderWorker.java
```

AnotherAdderWorker.java

1. Add a private field of type **Thread**.

Make the following additions in the **run()** method.

2. Call **Thread**'s **currentThread()** method and assign the return value to the private **Thread** field. **currentThread()** returns a reference to the thread being executed at that time.

3. Create a while loop that adds 1 to the private field sum during each iteration. The while loop should continue as long as the thread is not interrupted.

4. After the while loop terminates, output the value of the private field sum to the console.

ThreadInterrupted.java

Make the following additions to the main method:

5. Create two new instances of **AnotherAdderWorker**. Give the first instance an id of 0 and the second instance an id of 1.

6. Create two new threads and pass the **AnotherAdderWorker** objects as arguments. Start the threads.

7. Create a try-catch block that handles an **InterruptedException** exception.

8. Place a loop in the try-catch block that iterates 5 times. Put each thread to sleep for an amount that varies with each iteration. (Example: t1.sleep(10*i))

9. Outside of the try-catch, interrupt each thread by calling their **interrupt()** method.

Execute the Program

10. Compile and execute the program. The output should consist of two numbers. Each number represents how many times the loop is executed in each thread's **run()** method before being interrupted by the main program.

Submit

Submit your work as program 600 by 11:59 pm on November 22 with the command below. You must also work with the same partner as you are for project 3. You must also demo this lab at the same time as the November 22 milestone for project 3.

handin.465.1   600   *.java