LAB 13 – Concurrent Programming

Background

You are familiar now with the flow of program execution: the processor follows a path through your program--starting with your main method--executing statements sequentially, following control structures, method calls, and returns, until execution is complete. This serial flow of control or "thread" is at the heart of most algorithms.

While many programs are written simply as a single thread of instructions, computer systems actually allow multiple, simultaneous threads to be executing concurrently and concurrent programming is needed to fully exploit modern multicore computer systems. Carefully written algorithms can dramatically improve the performance of many applications, as well as make interactive programs, such as games, highly responsive to user input.

These threads have shared, concurrent access to data in your program, which can cause serious complications and makes concurrent programming quite challenging.

Introduction

In this lab, you will investigate various aspects of concurrent programming by using the Thread and Runnable classes in Java.

For each step in this lab, you are to create a Java class in its own file. Include a Javadoc header as shown below, but you do not need to provide additional Javadoc-style comments. At the end of the lab, you will turn in these files for grading.

```
/**
 * Name of Java Class
 * Optional description/comment
 *
 * @author your name (your login)
 *
 * @recitation section number and recitation instructor's name
 *
 * @date date of completion
```

```
 */
```

If the output from one of your programs is unclear or you can't explain an error message you're getting, be sure to ask the TA or one of your fellow students. It is important that you understand both why things work the way they do, but also why things don't work.

1. Problem MyThread

Summary: Create Java thread using the Thread class.

There are two direct ways to create a Thread in Java: (1) by defining a subclass of Thread and creating an instance of this new class, and (2) by implementing the Runnable interface and creating a generic Thread object that uses it.

For this problem, you are to use the first method: Define a class named MyThread that is a subclass of ("extends") the standard Thread class.

As usual, create a main method where execution begins. Note that the execution of this main method is itself "in a thread". To find the thread object associated with your running main method, you can call the currentThread() static method in class Thread. You can then "print" this object, which calls the toString() method and generates information about the thread.

Your main method should look something like:

```
public static void main(String[] args) {
    Thread t1 = Thread.currentThread();
    System.out.println(t1);
}
```

Compile and run this program. Your output should look something like:

```
Thread[RMI TCP Connection(1)-127.0.0.1,5,RMI Runtime]
```

The three values inside the square brackets are the thread name, priority, and group. You don't need to worry about the details yet, but the thread name can be useful to identify threads.

Next, create a new thread by creating an instance of your MyThread class by adding this line to your main method:

```
MyThread t2 = new MyThread();
```

As before, use println to print this thread object. You should now see output

similar to this:

```
Thread[RMI TCP Connection(1)-127.0.0.1,5,RMI Runtime]
Thread[Thread-2,5,RMI Runtime]
```

Java has created a new thread and named it “Thread-2”. If your output is (significantly) different, or you have a question, be sure to ask your TA or a fellow student. Note that the new thread is not yet running.

Class MyThread is now complete.

2. Problem MyRunnable

Summary: Create a Java thread using the Runnable interface.

For this problem, you are to create a Thread by using the Runnable interface. Create a class named MyRunnable that implements the Runnable interface. You need to create a “public void run()” method to satisfy the requirement of the interface. (Why didn’t you need a run method in the previous problem?) Have the method print “hello runnable thread”.

To start a thread, you must call its start() method. Your main method needs to do three things: (1) create an instance of your MyRunnable class, (2) create an instance of the Thread class, passing your MyRunnable instance as an argument to the constructor (see public Thread(Runnable target)), and (3) start the thread by calling the Thread start() method.

Compile and run this program. Does the output make sense?

Class MyRunnable is now complete.

3. Class Interleave

Summary: Illustrate thread interaction by allowing two threads to output simultaneously to the console.

In this problem, you are to run two threads concurrently and observe their interaction. Specifically, each thread prints a character repeatedly to the standard output. Since the two threads are running concurrently and there is only one output stream, these characters will be intermixed, producing output something like “aaabbabbbabbabbabbbabb…”.

Create a class named Interleave that implements the Runnable interface. Your class needs a char member variable, which will be the value to be printed by threads using that runnable. Create a constructor that takes a char as argument

and stores the character in the member variable.

In the main method, create two instances of the Interleave class, one with ‘a’ and the other with ‘b’ (or, use two different characters of your choice!). For each of those objects, create and start a corresponding Thread object.

The run method in your class consists of a main loop that prints the character member variable, say, 100 times to the output.

Run your program. You will probably be disappointed to observe that all of one letter comes out first, followed by all of the other letter. This behavior is due to the fact that printing (with the internal buffering) is so fast that the first thread to start finishes before the second one gets going.

One way to see the threads overlap is to slow them down. For example, in your main run loop, insert a subloop that calculates something repeatedly (computing Math.sin(j) 1000 times works). After inserting this delay in your loop, run your program again. If the output is interleaved, give a high five to the nearest person in the lab. Otherwise, think about it some and ask for help if necessary.

You can try different numbers of iterations for the calculation subloop. Too few iterations (e.g., 10) and you’ll see no interleaving; lots of iterations (e.g., 10,000) and the two threads will run in lockstep, producing “abababab…”.

Class Interleave is now complete.

4. Class Slowdown

Summary: Create a method that runs slowly.

In preparation for the following problem, (“Class Speedup”), you will write this program: class Slowdown.

Create a method “`computeRange(int lower, int upper)`” that performs some computation `upper-lower` times. For example, let `i` range from `lower` to `upper-1` and compute…

```
Math.hypot(Math.sin(i), Math.cos(i));
```

Your main method should call `computeRange(...)` with some range of values that takes a few seconds to execute. Try 100 thousand, 1 million, and 10 million.

To easily compute and display the execution time, use the method System.currentTimeMillis() . This method returns the current time of day in milliseconds. By calling `currentTimeMillis()` before and after computeRange(...), and subtracting the two values returned, you can get a pretty accurate estimate on the amount of time (in milliseconds) taken by computeRange. Note that `currentTimeMillis()` returns a `long`.

When class Slowdown reports about 4000 to 10000 milleseconds to compute, it is complete.

5. Class Speedup

Summary: Use multiple threads to speed the computation of `computeRange(...)`.

Implement class `Speedup` as a subclass of `Thread`. Divide the range of values to be computed in to `n` blocks, where `n` is the number of threads to be used (given as a command-line parameter). Create an array of `n` threads and initialize it to `n` `Speedup` objects, each one parameterized by its block number. The `run` method of each thread calls `computeRange(...)` giving upper and lower bounds corresponding to its block number (e.g., block `i` starts at `i*LIMIT/n` and goes to `(i+1)*LIMIT/n`). For `LIMIT`, use the range calculated in Class Slowdown. Ignore round-off if `n` does not evenly divide `LIMIT`.

Use the timer to calculate the begin and end times as in the previous problem, but note that the calculation is not complete until the last thread terminates. Use Thread.join() on each thread in your thread array to wait for that thread to terminate. You'll need to catch (and ignore, in this case) the exception thrown by `join()`.

Run your program with different numbers of threads (e.g., 1, 2, 4, and 8) and see how much the performance improves. Use ssh to login to mc17.cs.purdue.edu and try your experiments there. That machine has 24 cores!

Class Speedup is now complete.

6. Class Synchronization

Summary: Create a class that illustrates synchronization between two threads.

Make a copy of your `Speedup` class and call it `Synchronization`. Be sure to go through the code and replace all instances of "Speedup" with "Synchronization".

Create a `static` member variable named "`counter`" and initialize it in your `main`

method to 0. Create an `addOne()` method that simply adds 1 to `counter`; call this method from within the `computeRange(...)` loop. That is, each time you calculate `Math.hypot(...)` you add one to the counter variable. At the end of your `main` method, print the number of times the calculation should have been done (`LIMIT`) and the number of times you counted it being done (counter).

Run your program. If you run it with more than one thread, you should see that `counter` is somewhat less than `LIMIT`. This shortage occurs because the increment of `counter` in one thread was interrupted by another thread, which also incremented `counter`. These two increments overlap in time and only one of them is counted.

To fix this problem, create a synchronization variable: a `static` member variable named `object` of type `Object` will do. In your `main` method, initialize `object` to `new Object()`. In your `addOne()` method, insert a

```
synchronized (object) { … }
```

block around the increment of `counter`. This synchronization ensures that only one thread synchronizing on `object` is incrementing `counter` at a time; other threads that try are temporarily suspended until the block is free.

Class `Synchronization` is now complete.

Turn In

Before turning in your labs, please remove all class files from your lab13 directory. To do that, first verify that you are in the lab13 directory, if you are not in this directory, navigate there now.

```
$ pwd
/homes/your_login/cs180/lab13
```

Now remove all class files.

```
$ rm *.class
```

Finally, move to the parent directory of lab13, and submit your code via turnin.

```
$ cd ..
$ turnin -v -c cs180=XXX -p lab13 lab13
```

Note: The **XXX** stands for your section id. Use the following table to determine it:

Lab Day	Lab Time	Section-id to use in the turnin command

T	9:30	L01
W	9:30	L03
W	11:30	LM3
W	1:30	L02
F	11:30	LM2
F	1:30	LM1
F	3:30	L04

Grading

You will be graded on the following items.

- **Part 1: MyThread 10%**
  - Prints two Thread information blocks
- **Part 2: MyRunnable 10%**
  - Implements Runnable
  - Prints “hello runnable thread”
- **Part 3: Interleave 15%**
  - Implements Runnable
  - Prints interleaved ‘a’s and ‘b’s
- **Part 4: Slowdown 20%**
  - Prints duration of calculations
  - Calculations take between 4000 - 10000 ms
- **Part 5: Speedup 25%**
  - Prints duration of calculations
  - Calculations take between 4000 - 10000 ms with one thread
  - Calculations take less time with each additional thread up to at least four threads
- **Part 6: Synchronization 10%**
  - Performs according to guidelines for Speedup
  - Prints values of LIMIT and counter
  - addOne method called after each calculation
  - addOne method synchronized