Lab 5: Racing Threads

Purpose:

This lab introduces the concept of a *race condition* that arises when two or more threads contend for a shared resource. The lab uses the Producer/Consumer problem to illustrate a race condition.

Background:

The Producer/Consumer problem is the first of several classical synchronization problems you will encounter through the Java labs. (You can find a discussion of the Producer/Consumer Problem in the Stallings text beginning on page 222.) In the simple form of the problem there are two processes (or threads), a *producer* and a *consumer*. The producer thread "produces" data that it deposits in a shared buffer. The consumer thread retrieves data from the buffer and "consumes" it. The two threads execute (and access the shared buffer) concurrently. Ideally one wants the consumer to retrieve the data in the same order that the producer deposits the data. That is, the shared buffer should behave as a FIFO buffer. As you will see in the lab, this does not work as desired without providing synchronized access to the buffer.

The Java program you will be working with in this lab is structured as follows. There are four classes defined (each in a separate file):

- `Producer.java` – the Producer class defines the behavior of a producer
- `Consumer.java` – the Consumer class defines the behavior of a consumer
- `CubbyHole.java` – the CubbyHole class is a one-item buffer that is shared by the producer and consumer threads
- `PCTest.java` – the PCTest class is the main program that creates the buffer and creates and starts the threads

Instructions:

1. Compile the four classes and execute the program in the class PCTest.

   Run the program three times. Do you get the same results each time? Include the results of one execution with your lab.

   Find and write up a definition of *race condition*. Do you observe a race condition?

   Change the order in which the two threads are started and run the program another three times. Describe any differences in the results obtained. (Again include the results of one execution.)

2. Modify the program by changing the number of iterations the Consumer thread executes to four and adding two more Consumer threads to the code.

   Run the modified program three times. What observations can be made?

   Hand in a copy of the results of an execution and hand in a copy of the modified code.

3. In this step we will make a series of modifications to the CubbyHole class to correct the behavior of the program.

   The idea behind the modifications is to make the Consumer wait if there is nothing currently in the CubbyHole (the shared buffer). Similarly the Producer is required to wait if there is something in the CubbyHole.

   Introduce a boolean instance variable named *empty* to the CubbyHole class. You will need to add a constructor to the CubbyHole class which initializes this variable. Add the modifier *synchronized* to both the put() and the get() methods, i.e.

```
public synchronized void put (int value)
```

   In the put() method add a statement to wait if the CubbyHole is not empty. (Use the wait() method from the Object class.) After placing the value into the CubbyHole you will need to modify *empty* accordingly and then notify any waiting process (with the notify() method from the Object class).

   Make similar (symmetric) modifications to the get() method.

   Modify the program so that there is only one Consumer thread and one Producer thread and both of them process 10 items. Run the program three times. Does it behave correctly? Explain. Print the results of one execution.

   Modify the program by switching the order in which the Consumer and Producer threads are started. Run the program three times. Does it behave correctly? Print the results of one execution.

   Modify the program to have the Consumer thread consume 4 items and to have the Producer thread produce 12 items, then create three Consumers and one Producer. Run the program three times. Does it behave correctly? Print the results of one execution.

   Make one final modification to the program. Introduce a delay at the end of both the get() and put() methods of CubbyHole. You can force a delay with the statement

```
Thread.sleep(n);
```

   where *n* is the number of milliseconds. Try 100 msec. Run the program three times. Does it behave correctly now? Print the results of one execution.

Hand in:

Hand in the code requested in Steps 2 and 3 of the instructions. Hand in the results of the executions from Steps 1, 2, and 3. Hand in a write-up of the observations and discussions requested in Steps 1, 2, and 3.

Email the final versions of your Java files to your instructor.

Looking ahead:

If you want to start on the next step of these exercises, replace the CubbyHole class with a BoundedBuffer class. A BoundedBuffer will have the same methods as a CubbyHole, however it will be capable of holding up to *n* integers instead of just a single integer. (The value of *n* should be a parameter of the constructor.) The BoundedBuffer will be implemented as a circular array.