# CSCI 315 Lab 5 Exercise

February 24, 2010

**Objectives:** This lab gives you the chance to experiment with semaphores and monitors in the context of classic concurrent programming problems such as Bounded Buffer and Readers and Writers.

**References:** Sun's Java Tutorial on threads: synchronization
Silberschatz, Galvin & Gagne: Chapter 6

**Pre-Lab:** Read through Section 6.5 of your text. Copy the file `Semaphore.java` and examine the code. The keyword `synchronized` makes sure that the `acquire()` and `release()` methods are atomic. As usual, the file Semaphore.java is also in the directory ~cs315/Labs10/Lab05.

**There is nothing to hand in for the prelab.** However, lab will be easier and go faster if you do the reading and look at the code.

You should also review Sections 6.6.1 (a semaphore solution to the bounded buffer producer-consumer problem) and 6.8 through 6.8.3 (a solution to the readers-writers problem using Java synchronization) before the start of lab. **Bring your text to lab.**

**Problems:** In **each** of the following exercises, **save and print a copy of your solution** as requested. **Be sure to label each solution with the problem it is for!** Hand in answers to the questions where instructed to.

**A.** Copy the following files for the Bounded Buffer Producers and Consumers problem to your Lab 5 directory: Server.java, Producer.java, Consumer.java, and BoundedBuffer.java. They are also available in the directory ~cs315/Labs10/Lab05/ProdCons.

Remember that there are two kinds of problems that can occur when concurrent threads cooperate: race conditions and synchronization. In the problems that follow there may be bits of code added so that only one problem will appear at one time. When this occurs you will be told. Complete the following activities:

1. Compile and run the program (`main()` is in `Server.java`). Let the program run long enough to observe a problem. Focus on the changes

to the value `count`. What do you see? Look at the code to determine what's going wrong. **Hand in answers to the following questions:** What causes the error to occur? Does this correspond to a problem that can occur in a real system?

Notice that the `yield` loop at the beginning of the methods `enter` and `remove` in `BoundedBuffer.java` are meant to prevent one kind of problem from occurring. Do you see which kind?

2. Using the `Semaphore` class from the prelab, fix the problem in `BoundedBuffer.java` by using a binary (mutual exclusion) semaphore. What is the initial value of such a semaphore? **Add only this one semaphore at this point.** Make sure that the `System.out` statement at the end of each method is inside of the critical section.

   **In your hand-in file, include a listing of your corrected `BoundedBuffer.java` file showing your solution** (be sure to label this code with the problem it solves). Be sure to test your solution to convince yourself that the problem seen earlier doesn't occur anymore.

3. Now that the race condition is taken care of we will expose the other problem. Carry out the following modifications to your current `BoundedBuffer.java` code (that is, the code that solves the previous problem).

   (a) Comment out the `yield` loop at the beginning of the method `enter`.

   (b) Modify `Server` to launch three `Producer` threads. Make sure you give them distinct names so you can distinguish them in the output.

   (c) Add a buffer overflow test to `BoundedBuffer`'s `enter()` method immediately after the code that increments `count`. Print a "Buffer Overflow" message if (and only if) `count` exceeds `BUFFER_SIZE`.

   Now compile and run the program. You may have to let it run for a while; just watch for the "Buffer Overflow" message. **Include an answer to the following question in your hand-in file**: What occurs and why?

   Do the same thing now for the method `remove()` (the message here should be "Buffer Underflow") and test the program to make sure it behaves as you expect. You **don't** have to hand in this part.

4. Correct the problem the previous changes have exposed by adding synchronization (counting) semaphores to insure that overflow and underflow don't occur. **Hand in** a listing (i.e., a printout) of your corrected `BoundedBuffer.java` file. Again, be sure to label the code in your hand-in file.

5. Modify `Server.java` so that it creates two `Consumer` threads and then run the program to see that the race condition and synchro-

nization problems have been solved. **Hand in** a page or so of your output to demonstrate that things don't get messed up.

**B.** Create a new subdirectory in which to work on this part of the lab. Copy into that directory the following files for the critical section problem: Server.java, Worker1.java, and Worker2.java (the files are also in ∼cs315/Labs10/Lab05/Semas). You will also want to use the semaphore class from the previous part: Copy its `.java` file into the new directory as well.

Complete the following activities:

1. Compile and run the programs as is (the `main` method is in `Server.java`). Let the program run until it generates 15-20 lines of output. Consider the output in light of the comments in the worker code indicating the existence of critical sections. If there seems to be a problem describe the problem and how you deduce it from the output. **Hand in the output and your answer.**
2. Now make appropriate use of the semaphore passed to the workers to solve the problem just discovered. Run the resulting program and (**in your hand-in file**) explain how the output seems to indicate a solution.
3. Find **three different ways** to create a deadlock among the workers by modifying the semaphore or its use (see Section 6.7 of your text). The ways should be distinct. Causing a problem in one of the workers, then causing a symmetric problem in the other worker, will not count as different solutions. **For each answer**, turn in your code for the worker classes and the output that shows a deadlock.

**C.** In this part of the lab you will experiment with a solution to the readers-writers problem to see what properties the solution seems to have (see Section 6.6.2 of the course text for discussion of the problem and relevant code). Then you will re-implement the solution using the Java synchronization facility. Again you can find information about this Java feature in Section 6.8.3 of the text.

Before getting started create a new subdirectory in which to work on this part of the lab. Copy into that directory the following files for the critical section problem: Server.java, Database.java, Reader.java, and Writer.java (also in ∼cs315/Labs10/Lab05/ReaderWriter). You will also want to use the semaphore class from the previous part; copy its `.java` file into the new directory as well.
Use this code to carry out the following activities:

1. The program code you copied contains a semaphore solution to the readers-writers problem. You are to run the program and let a couple of screens worth of output accumulate. Can you identify a problem with the solution? Does it seem to provide protected access to the

database? Does it seem to provide fair access to the database? Does increasing the number of *writers* change the situation? Justify your answers by refering to the output, and **include them in your hand-in file**.

2. Java provides a `synchronization` facility to manage process synchronization without using the `semaphores`. Read the course text about the subject and the code example.

   Create a new directory and copy into it your .java files. Work in this new directory to produce a new solution to the problem by modifying the `Database` class to use the Java `synchronization`. (**Note** that you should not copy out the code from your text; the implementation differs from the code you have. Rather, you should use it as a model to change your Database class.) When this is working run the resulting program and answer the questions of the previous part **in your hand-in file**. Contrast the output from this version with that from the semaphore solution.

**Hand in:** Hand in your code and answers and/or discussions by 5 p.m. next Monday.