CSCI 315 Lab 4 Exercise

February 17, 2010

Objectives: In this lab you will study the behavior of algorithms associated with scheduling Java threads.

References: Sun's Java Tutorial on threads; *Operating System Concepts with Java* - Chapter 5, Sections 3 and 7.

Lab Problems: In the first two parts of this lab you will work with problems that help you understand how thread scheduling works in the Java Virtual Machine. Here is the basic problem you will address: Imagine writing a Java program that will generate lots of threads. For performance reasons you want to manage the scheduling of these threads yourself; you have copied the code for a Java `Scheduler` class. You will work with this scheduler to understand how this `Scheduler` class interacts with the JVM threads scheduler.

Keep in mind that your program (based on the `Scheduler` class) is run as a thread by the JVM. From the perspective of the JVM your program thread, the scheduler thread, and any other threads can all be in the runnable state at the same time. Therefore, even if your program's scheduler thinks it has somehow blocked the execution of a thread, that thread is ultimately managed by the JVM, which is actually in control.

Also, remember that the JVM schedules threads using a preemptive priority scheduling scheme.

A. Copy the files Parent.java, Counter.java, and Scheduler.java to your Lab 4 working directory (these files can also be copied from the ∼cs315/Labs10/Lab04/ directory). Compile and run the program (the `Parent` class has `main()`). Look closely at the output. The `Scheduler` is supposed to set priorities so that only one thread runs during a `Scheduler` time slice (Scheduler time slices are indicated by lines starting with "Starting" and "Stopping"). However, `Scheduler` is running on top of the Java thread scheduler, so some interesting interactions take place. Run Parent a few times to get a good idea of what is happening.

It should be clear to you that the `Scheduler` does not run as desired. Other threads are being scheduled and run by the JVM thread scheduler in addition to the threads that the `Scheduler` class is trying to schedule.

Modify the `Counter` class to prevent this problem. You will want to make sure that the `Counter` thread gives up the CPU if it finds itself running when it shouldn't. **Hint:** How can a thread recognize that it is running when it shouldn't be? How do you get a thread to give up the CPU? (Recall last week's lab, which should *yield* a solution! 8-))

> *When you see how to get a thread to give up the CPU, you will find that you can't do it at a precise time. That's OK.*

Compile your modified code, and run the `Parent` program to verify that your solution works. **Hand in enough of `Counter.java` to show where and what your modification was, and hand in the output that verifies your solution works.** Once your program is working you can try changing the default time slice to see what happens.

B. Modify the `Scheduler` to include 3 queues representing three different priorities (with values of 2, 3, and 4). The `Scheduler` should pick a thread from the highest-priority queue that is not empty and run that thread for a time slice. When all threads in the highest-priority queue are finished, move to the next queue, continuing until all queues are empty. Note that in order to implement this, you will have to change the `Scheduler` so that it sets the priority of the thread it is running to 5, and that it remembers the original priority of the running thread so that it can be returned to the correct queue when the time slice is done. Note that you will also have to modify the `addThread()` method to take a second argument indicating the priority of the thread. You will also probably have to modify the `Counter` class, since your change in Part A was based on having the `Scheduler` run threads at priority 4.

Once you've made the modifications, change `Parent.java` to start up three counters, one with priority 4, one with priority 3, and one with priority 2. Run the program and verify that it does what you expect.

Hand in the code for the modified `Scheduler.java` and the output of your test run.

D. In Section 5.3.2 of the text we discussed the *Shortest Job First* scheduling algorithm and its basic problem: It requires knowledge of future events, in particular, the length of the next CPU-burst. The solution to this problem is to use a predictive method to make a guess of the size of the next CPU-burst for a process. The idea is to base the guess on the lengths of the CPU-bursts the process has already had. This part of the lab allows you to explore this approximation process.

To get started you should copy the following files to your current lab directory: Boss.java, Clock.java, RandomV.java, Timer.java, and Worker.java (they are also in the directory ~cs315/Labs10/Lab04/SJF). Compile them, and then run `Boss` with the following command line arguments:

```
% java Boss 50 0.5
```

What you will see is a sequence of integer pairs, one per line, with the second integer always zero. The first integer on each line is a (randomly generated) CPU-burst length. The second value is meant to be a guess based on the technique described in the book, prediction using *exponential average*.

You are to modify the method `processTics()` in the `Clock` class so that the value of `guess` is correctly calculated using the exponential average, described in Section 5.3.2. Notice that the required values for α and $1-\alpha$ are already available, so your job is made a bit easier.

After making this modification you are to run the `Boss` program three times as follows:

```
% java Boss 100 0.8 > data8.txt
% java Boss 100 0.5 > data5.txt
% java Boss 100 0.2 > data2.txt
```

Each execution generates 100 CPU-bursts and guesses, based on α values 0.8, 0.5, and 0.2, respectively. Each data file generated contains a tab separated list of two data items per line. These data files can be read into Excel, plotted, and printed out. Instructions to do this are at the end of this lab description. If you know how to use MatLab to generate the plot of the graph you may do that instead.

Examine the three graphs that you print and describe how the three values of α allow the algorithm to respond (in terms of making predictions) to short-term changes in CPU-burst length.

Hand in the code for the modified `Clock` class, the three printed graphs, and your analysis of the graphs.

Hand in: Hand in the code and other material specified above by next Monday at 5 p.m.

Using Excel: You can generate the graphs quickly using Excel. When you select `open`, you should tell Excel that you want to open a text file with tab separation. There will appear a series of 3 dialog boxes asking questions about how to process the data: just click on `Next` in the first two boxes and then on `Finish` in the third. You should see your data appear in the spreadsheat.

Now now select the `Insert` tool bar. Choose `Line` from the chart types. The graph should appear.

You can print the graph by selecting the graph (actually, when the graph appears it should already be selected), and selecting the `Print` option from the menu that appears when you click the Office buttonW.

Repeat this operation for each of the three data files. If the printouts are not labeled, then write a label at the top of each, indicating the command which generated the data on the graph.