CSCI 315 Lab 3 Exercise

February 11, 2010

Objectives: In this lab you will work with the Java language and its threads class. A thread is similar in concept to a process and in this lab you will learn about some of the differences.

References: Sun's Java Tutorial on threads.

There are four Java classes that consitute the first example Java program of this lab. Copy the following files to your space (files are also available in the directory ~cs315/Labs10/Lab03): Client.java, Server.java, and Share.java. You are to write the fourth class yourself – call it `Thing.java`. The `Thing` class should be public, contain a single private integer variable (call it `Value`) that is set to zero by the constructor, and three public methods:

- `int getValue()`: This method returns the value of `Value`.
- `void setValue(int)`: This method sets `Value` to the value of the integer parameter.
- `String toString()`: return a `String` object containing the value of `Value`, as shown below:

```
Value = 25
```

Lab Problems: For problems 1 and 2 below, **write your comparisons, explanations, and answers to ALL questions into your hand-in file.**

1. Consider the Java program `Share`.
   (a) Run the program. How does the output differ from what you would expect from a program based on creating processes rather than threads? Assuming the same interleaving of execution, what would you expect the output to be if processes had been forked, rather than threads started?
   (b) If the calls to `start()` were replaced by `fork()` calls, how many times would you expect the '`Parent is Done!`' message to be printed assuming no calls to `exit()` are made by the children?

(c) Does the `Share` program mimic executing the children process with or without the '&' option? **Explain** based on the output.

(d) Add the following code segment to `Share.java` just after the call '`client.start()`':

```
try { server.join(); }
catch (InterruptedException e) {}
```

Explain the effect on the output of the program. Then add `client.join()` within the `try` block after the `server.join()`. **Explain** the resulting effect.

2. Copy the files `Adder.java`, `Sub.java`, `Parent.java` to your working directory, along with the file `Thing.java` that you wrote. Consider the program determined by these files.

(a) Examine the program and determine what answer should be printed. **Explain.**

(b) Add the following line to `Sub.java` just before the line that starts "`T.setValue(v)`":

```
yield();
```

This causes the current thread to yield, allowing another thread of the same priority to run. Note that the scheduling algorithm picks a thread from the set of runnable threads with the same priority; this includes the thread that just yielded. (You can look up the `yield()` method in the Thread class documentation here to see what it does, though the description is not very detailed. Just find the class name in the list on the lower left and click.) Recompile and run the resulting `Parent` several times. **Compare** your results to those of the previous problem and **explain** why there is or is not a discrepency.

(c) Modify the Parent program so that `Sub` starts first rather than `Adder`. Be sure to run `Parent` several times. **Compare** your results to those of the previous two problems and **explain why** there is or is not a discrepency.

(d) What conclusion do you draw from these two problems about threads modifying shared data (objects)?

3. Solve the Conway problem from last week using Java threads. Hint: You can base your answer on last week's solution, using the MessageQueue class discussed in Chapter 4 of the text and demonstrated in lecture. We've provided code files for the Java implementation of the producer/consumer problem (these are in the subdirectory ProdCons): `MessageQueue.java`, `Producer.java`, `Consumer.java`, and `Server.java`. They will be useful as references.

To solve Conway's problem we suggest you create three `Thread` classes (a reader class, a compressor class, and a printer class) to do the work and

a main class to set everything in motion. Use `MessageQueue` objects in place of the pipes in last weeks solution. To help you get started we have prepared partial implementations of two of the class files you will need: `Conway.java`, which contains the main class, and `Reader.java`, which is obviously the reader class.

Here are some hints and advice:

- **`Reader`:** In the file `Reader.java` there are comments indicating what is required for completion.

  You want to put characters in the message queue, but the queue requires objects, not primitive values. Thus you must use the `Character` wrapper class.

- **`Conway`:** This file should be completed in the obvious way: create the necessary message queues and threads, and then start those threads.

- **`Compressor`:** This class will serve as both a consumer and a producer. Draw your inspiration from `Consumer.java` and `Producer.java`.

  In older Java implementations you would have needed to use the `Character` method `charValue()`, which returns the char value contained in the object. Java 6 should provide auto-boxing, which allows you to operate with the `Character` object as if it were a `char` variable.

- **`Printer`:** This class will simply serve as a consumer. Part of your work on `Compressor` can be used as a model for what you need to do here.

For your convenience here is a copy of the `text.txt` input file you used in the previous lab. You should use it to test your program. The output should be identical to what you got from last week's lab.

Hand in: Hand in the answers for problems 1 and 2 along with a listing of the classes for the Conway problem and the output from a sample execution by next Monday. For the Conway problem, put the class listing into a single file and then print that file with a2ps.