Threads in Java

Objectives:

1. Explore communication between Java threads.
2. Explore coordination between Java threads.
3. Use of `synchronized` keyword in Java.
4. Use of `wait` and `notify` keywords in Java.

Reference: Chapter 23 in *Java: How to Program* by Deitel and Deitel, sixth edition.

Laboratory Assignment:

Exercise 1: Communication Between Threads

Communication between threads can be accomplished by passing values in a *shared variables* or a shared data area. In C or C++, several threads could share a global variable. However, Java does not have global variables. In Java, each thread's local variables in methods are private to it because each thread has its own private activation stack. However, threads are not given private copies of static class variables or object instance variables. Therefore, you might ask what's a good way to share a variable in Java?

Copy the following file `~cs355/Lab9/ProducerConsumer.java` into one of your own directories. Note: this an example program we used in class.

Study the program and explain how the two threads share an object.

Compile and run the program many times to observe its behavior. This program should inhibit *non-deterministic* behavior. If it doesn't, try running on a loaded compute server like `linuxcomp3`.

What is *non-deterministic* behavior? Why is it usually undesirable behavior?

Whenever, two or more threads (or processes) share a variable (or data area), any computation on the shared variable by one thread must be protected from interference by the other threads. Only by careful analysis can we identify these *critical regions* and then use a *synchronization primitive* to protect them. In Java, this synchronization primitive is a Hoare-style *monitor* **associated with the object**. The monitor guarantees that at most one thread can execute within a critical region *for that object* at any time.

Java provides the `synchronized` keyword for programmers to designate these critical regions *for an object*. The keyword `synchronized` may be placed on a method of a class or surrounding a block of code where in the latter case the programmer must specify the object. It is easy to do this wrong!

By analyzing the program and inserting `synchronized` keywords, fix the program. Where did you insert them?

Exercise 2: Coordination Between Threads

In the previous exercise, we focused on the correctness of communication between threads. A second important but different issue is the *coordination* between threads. Multiple threads need to alter their behavior depending on what other threads are doing. If you are not careful, your multi-threaded program will run with each thread acting like an independent-minded cat. Proper coordination is difficult, i.e., like herding a dozen cats! If the programmer is not careful, a thread can *starve*, i.e., because of unfair coordination a thread never gets a chance to run. Also, a group of threads may *deadlock* where all the threads in the group are waiting for an event which can only be produced by one of the group.

A simple example of coordination is the interaction between a producer and a consumer. For example, the consumer should wait if there is nothing to consume. And, the producer should wait if the shared data structure becomes full.

Assume we desire the value of the shared variable n to always be 0, 1, 2, 3, or 4. Alter your solution of exercise 1 to require the consumer to wait if n has a value of 0 and require the producer to wait if n has 4. Use the `wait()` and `notify()` methods to accomplish this task. See Java text for help.

Exercise 3: Using the `Runnable` Interface

A class in Java can only **extend** one class, i.e, a class can only inherit from one class. Multiple inheritance is not allowed. However, most of the benefits of multiple inheritance, can be achieved in Java by *implementing* several *interfaces*.

If you have a class that already inherits a superclass by `extends` and you want threads, you must use the `Runnable` interface.

Change the program of Exercise 2 to *not* use `extends Thread`. Create the two threads by using the `Runnable` interface technique.

Hand In:

Hand in a copy of the listing of the Java code and sample output for Exercises 2 and 3. Also, hand in the answers to the questions of Exercise 1.