CS 341
Software Design
Minilab 2
Race conditions
December 4, 2007

Interacting Threads

Today we will continue working with the water tank problem. You will need to download a new version of the program to start with. This version is very similar to the version you ended with last week except:

1. The emptying thread automatically stops when the the tank becomes empty.
2. There are labels displaying the starting volume of water, the total that has been added, the total removed, and the current volume.

Step 0: Observe a race

A race condition occurs in a program when two threads use a shared variable and at least one of those threads modifies the variable. If thread execution interleaves in an unfortunate way, it is possible for the reading thread to see the object in an inconsistent state. This is the example of the bank account that is updated from multiple concurrent transactions.

In this program, both the filling and emptying threads change the current volume of water. Start the program as it is currently written, starting both a filling and an emptying thread (quickly, the pause time has been greatly reduced so you don't need to wait as long for the race condition to happen). Let it run for a minute or so and then stop the threads (or they may have both stopped themselves by then anyway). Add starting volume to total added, then subtract total removed. Does the result equal current volume? If it does, you got lucky! (Or unlucky, run it again for a longer time.) If they are not equal, this is an indication that there is a race condition involving the updating of the current volume of water.

Step 1: Eliminate the race

If you look in the addWater method, you will see code that looks like this:

```java
int newVolume = currentVolume + 1;
Thread.yield();
currentVolume = newVolume;
```

This is functionally equivalent to currentVolume++. It has been written in this peculiar way to increase the likelihood that a context switch will occur between the reading of the currentVolume variable and its update, the situation that makes the race condition apparent. A properly synchronized program will not exhibit the race condition even if it is written
in this bizarre way. Your first job, then, is to add synchronization to this program to remove
the race condition. (Leave the peculiar coding in place!) To test it, let it run for a while and
verify that the numbers work correctly. (Testing concurrent programs is not nearly so con-
vincing an exercise as testing sequential programs!)

**Step 2: Wait and notify**

Last time we used a shared variable to allow two threads to communicate. Another way they
can interact is using `wait()` and `notify()` (or `notifyAll()`). `wait` is used to allow a thread
to give up a lock if it cannot proceed with its operation. `notify` is used to inform other
threads that the state of the object has changed and it may be possible for a waiting thread
to continue. These methods can only be called while the thread holds a lock (typically mean-
ing inside a synchronized method or block). Also, `wait` should always be inside a while loop,
because it is possible the object's state will change again between the time that the other
thread called `notify` and this thread is allowed to proceed.

To practice using `wait` and `notify`:

- change Filler's run method so that it doesn't end when the tank is full,
- add a call to `wait` so that the filling thread waits when the tank is full,
- add a call to `notify` when water is added to the tank so a waiting Filler wakes up

When doing this, remember that `wait` and `notify` can only be called when a thread holds a
lock. That is, they can only be called from synchronized methods or from methods called by
synchronized methods.

Test this by:

- starting the Filler
- letting the tank completely fill
- starting the Emptier

When the emptier starts, the water level should go up and down a bit but stay near the full
level.

Make analogous changes so that the Emptier waits when the tank becomes empty rather
than stopping.