CS032 2009 Lab 03
Threads

1 Introduction

In this lab you are going to learn about threads and concurrency, which the upcoming assignment, NewsWhere, will focus on.

2 What are Threads?

A thread is one "execution context" within a program. When you run two processes at once, the operating system creates one thread for each process, so both processes can make progress. Similarly, if you wanted to do more than one thing at once from within a single program, you could create a thread for each job to be done and then all jobs would make progress at the same time (in parallel). For example, in a simple game, the graphics engine is often controlled by one thread, while the logic is controlled by another thread.

It's worth noting that on a computer with only one processor, there is never actually more than one thing running at a time. However, the operating system's scheduler rapidly switches between threads (usually every 10-100 ms), so it seems like many things are running at once.

3 Threads in Java

To use threads in Java, you have two options. First, you can make a subclass of the class `java.lang.Thread`. When you make a subclass of `Thread` you should override the `run()` method to do whatever that thread's job will be. If you make a subclass of `Thread`, you should pass an intuitive name to the `Thread` class' constructor. Naming each thread helps you to keep track of what's going on, and yields more useful error stack dumps. If you don't pass a name, the thread will be anonymous, and be known by a unique number instead (not useful for debugging). To get the name of a thread later, use the `getName()` method.

Here's an example of a subclass of `Thread`:

```
public class MyThread extends Thread  {
    public MyThread (String name) {
        super(name);
        // rest of constructor here
    }

    public void run() {
        // ...do stuff...
    }
}
```

The other way to make a new thread is to create a class that implements the `Runnable` interface. This may be more intuitive if your program breaks down into "jobs" more easily than into "threads," or if you want to enhance an existing class by making a multithreaded subclass.[1] To make a new thread out of a `Runnable`, make an instance of that class and then pass it to `Thread`'s constructor (also optionally pass a name for the thread). Here's an example:

```
Thread t = new Thread(new MyRunnableCoolJob(), "Cool Thread");
```

[1] This is necessary because of Java's single inheritance; you cannot inherit both from the superclass and `java.lang.Thread` if the superclass does not inherit from `java.lang.Thread`.

Important note: Creating a thread doesn't run the thread. In order to start your thread, call `start()` on that instance and the thread will begin executing the code in your `run()` method. When `run()` completes, the thread "dies." To summarize, both of the following pieces of code will create a thread object and then run it.

```
MyThread thread = new MyThread("MyFirstThread"); // MyThread extends java.lang.Thread
thread.start();
```

OR

```
Thread t = new Thread(new MyJob(), "MyJob Thread"); // MyJob implements Runnable
t.start();
```

4 Part I: Synchronization

Threads execute in parallel, and they execute independently of each other, but sometimes you need to coordinate their activities. One problem that frequently arises is controlling access to resources shared amongst threads. The solution to the need for access control is called synchronization.

The best way to explain this problem is to show you. You're going to create a simple program with four threads calling `increment()` on the same instance of a simple class `Counter`. Each thread should increment the counter 10,000,000 times. The counter should read 40,000,000 at the end.

1. Create a class `Counter` with one private instance variable, `int count = 0`, and two methods, `increment()` and `getCount()`. The `increment()` function should look like this:

   ```
   count++;
   ```

2. Create a separate `MyThread` class which extends `Thread`.[2] `MyThread` should take a `Counter`, and a String for its name in the constructor. Pass the String up through `super` and store a reference to `Counter` (note that `super` must be called first in the constructor). Override the `public void run()` method of `MyThread` to call `increment()` on the `Counter` reference 10 million times.

3. Write a mainline for `Counter` that instantiates a `Counter` and then instantiates four `MyThread`s using the instantiated `Counter` as a construction argument. Finally, it should `start()` the four threads.

4. `join()` each of the threads after all four have been started. `Thread.join()` causes the current thread (in this case, the main thread of your program itself) to wait for the given thread to finish before it terminates. Joining the four `MyThread`s to the mainline means that the program will not finish before the four MyThreads have finished. Be sure to catch the exception![3]

5. Print the final count after all the threads are done.

6. Compile and then run your code.

Try running the code multiple times. What happened? It didn't work, did it?

You have now reached CHECKPOINT 1.

What went wrong, and what can you do to fix it? Well, when your code is executing, the act of incrementing the counter takes place in three steps (recall CS31): first, the value of the counter is retrieved and stored in a register; second, that value is incremented by 1; third, the new value is stored in counter's location. Let's say thread A is in step 1 and stores 100 in a register. Control then shifts to thread B which also stores 100 in a register, increments it to 101, and stores it in counter's location. Shortly after, control returns back to A which increments 100 to 101 and stores it in counter's location. This results in a "lost" increment. As long

[2] It's actually `java.lang.Thread`, but in this case java automatically recognizes the package, as with anything in `java.lang.*`.
[3] Threads can be `interrupt()`ed, which interrupts execution. This is often used to pass messages between threads.

as you have multiple threads running, there is a small, constant chance that they will lose synchronization with each other. For the first few thousand or possibly even million operations, the second thread may not even start.

So how do you fix this? Programming languages solve this in many different ways. Java makes it relatively simple to achieve safety with the keyword `synchronized`, which can be used in a method declaration. Whenever a thread enters a synchronized method of an object, it locks that object so that no other thread can call any synchronized method on that object. Once the thread leaves the aforementioned synchronized method, other threads are then free to call any synchronized methods on that object.

If we use this on the `increment()` method of `Counter` each thread would have exclusive access to the number in the `Counter` while in the `increment()` method. This prevents the problem stated above, since each thread has to do all three assembly-level steps before another thread can do acccess `increment()`.

Go ahead and open up `Counter.java` and make the `increment()` method synchronized as so:

```
public synchronized void increment()
```

Now run the program again and see that the count is always 40,000,000. Note that it may run more slowly because while one thread is incrementing the counter, the other three must wait.

Welcome to CHECKPOINT 2. Population: 1337.

5 Part II: Producer and Consumer

Now that you know why data synchronization is important let's try to implement the concepts we just learned in a simple producer and consumer example. In this example we will try to make data access synchronized. Copy all the .java files from `/course/cs032/pub/labs/03_threads/src/`, as you will use them in this section. Note that they have no package declaration, and thus should go in your source root (alternatively, you can create a package for them and add the package declaration to the top of all four files).

In this program, there exists a `CubbyHole` which acts as the buffer which multiple threads are attempting to access. There is a `Producer` that puts a number in `CubbyHole`, and a `Consumer` that takes the number out of `CubbyHole`. The desired functionality of the program is to have `Producer` put a number in `CubbyHole` and then have `Consumer` take that same number out, alternating repeatedly (i.e. put 1 in, take 1 out, put 2 in, take 2 out, ...). To run the unsynchronized demo type `/course/cs032/pub/lab03/lab03.jar`.

As you can see, `Producer` might put several numbers in `CubbyHole`, overwriting the last one each time, before `Consumer` has a chance to get a single number from `CubbyHole`. Alternatively `Consumer` might get the same number several times before `Producer` has a chance to update the number.

As a first step towards fixing the program let's make it so that only one thread can access `CubbyHole`'s number at a time. To do this we must lock `CubbyHole`. To apply this to our producer/consumer example we can declare both the `get` and `put` methods of `CubbyHole` synchronized so that only one `Producer` thread or one `Consumer` thread has access to the `CubbyHole`'s number at a time. Do this now.

That takes care of two threads accessing the same data at the same time and prevents the problem we saw when we ran the demo. But how do we control the order in which the two threads access `CubbyHole`? We are going to have to use a flag (in what's called a *spinlock*; a "flag" is sometimes also known as a "semaphore") and make use of two additional methods that deal with threads, `notifyAll()` and `wait()`. Every object has these methods.

First off, we want to make a boolean in `CubbyHole`, `is_available`, and set it initially to false. We are going to use this boolean to keep track of whether a new value has been set and whether it has been retrieved. We do not want to use simple if statements in our `get` and `put` methods because if one was called and the boolean did not evaluate to allow the thread access, then that could would end up doing nothing. Instead, what we want is for threads to wait (looping) in `get` or `put` until the boolean allows the thread to access the number in `CubbyHole`. We can do this with `wait()`. We might use it something like this:

```
public synchronized int get() {
```

```
    while (is_available == false) {
        try {
            wait();
        }
        catch (InterruptedException e) { /* egad!  how rude! */ }
    }

    is_available = false;
    return the_contents;
}
```

`wait()` relinquishes control of the `synchronized` lock that a thread has on the object, in this case `CubbyHole`, so that another thread can obtain the lock and call the synchronized method. When the thread returns from `wait()` it has regained the lock it once had on the object. This way when the `Producer` thread finds the boolean in a state that prevents it from accessing the number it can call `wait()` and hand the lock and control off to another thread (eventually, a `Consumer`), so that when it does regain control and the lock, it will be the producer's turn and the boolean will allow the thread access to the number. The same goes for the reverse.

Now consider when a `Consumer` thread is accessing the number in `CubbyHole` while the `Producer` thread is waiting for the boolean to become a correct value. How do we wake up the `Producer` thread once our `Consumer` thread is done? We can call `notifyAll()`. `notifyAll()` wakes up all[4] threads that are waiting to regain the lock that the current thread has. This way when the thread relinquishes the lock, all the awakened threads can compete to get the lock. Here is an example of how you might want to use `notifyAll()`:

```
public synchronized int get() {
    while (is_available == false) {
        try {
            wait();
        }
        catch (InterruptedException e) { /* well excuse you */ }
    }

    is_available = false;
    notifyAll();
    return the_contents;
}
```

Bienvenidos a CHECKPOINT 3.

Update the `get` method to reflect the changes above. Now you should code the rest of `CubbyHole` (just `put()`), so that you have a fully synchronized producer-consumer program. The `put` method should look remarkably like `get`; however, instead of adding to the buffer, it removes.

Note that `notifyAll()`, `notify()`, and `wait()` can only be called in synchronized sections. If you call it in another circumstance, an `IllegalMonitorStateException` will be thrown. This is because you must synchronize on an object to access Java's list of threads waiting on the object.

CHECKPOINT 4. Get checked off and get out of here!

[4] `notify()` wakes up a single thread, randomly chosen from the list of waiting threads.

6 Checkpoints

1. 25 points TA Initials: _____

   - `Part I` runs.

2. 25 points TA Initials: _____

   - `Part I` runs successfully.

3. 25 points TA Initials: _____

   - `Part II` get is implemented.

4. 25 points TA Initials: _____

   - `Part II` put is implemented and `ProducerConsumerTest` runs successfully.