Computer Science 252
Problem Solving with Java
The College of Saint Rose
Spring 2016

Topic Notes: Concurrency

We introduced active objects early in the course as a way of providing objects which behaved independently of user actions. Our `ActiveObject` class encompassed what are usually termed *threads of control*.

Java includes a class named `Thread` to support this idea. Rather than starting directly with "raw" Java threads, however, we started out with the `ActiveObject` class, which interacted well with our `WindowController` class and provided a number of methods that allowed you to better control threads. Now, you know enough to be able to use Java threads directly, and we encourage you to browse the Java documentation about threads. Our examples for this topic use only standard Java – no Objectdraw graphics, and our "Main" class extends `JApplet` and our "active object" extends `Thread`. More on this later.

Our main focus for the moment is on complications that arise with threads and provide some indications of how to handle them.

Interference

We've talked about how an `ActiveObject` activates an extra "brain" for your program. Imagine your body was controlled by two brains. Suppose one brain gives instructions to your muscles to start running, the other to sit down. The muscles will receive multiple messages, and the result is not likely to be good. Back to "brains" in our programs, what happens if those brains give your program contradictory messages?

An important class of problems can arise with concurrency when there are several threads that might try to update the same variable at the same time. Many of the important problems with parallel, concurrent, and distributed programs boil down to this problem.

Consider an example of a bank with two ATMs which can be used to deposit and withdraw money.

See Example: ATM1

One of the ATMs will repeatedly withdraw $100 from the account while the other will repeatedly deposit $100 into the account (see the difference in parameters in the constructors for the ATM's). When the user pushes the button, the `actionPerformed` method repeats the construction and execution of the ATM objects.

The main items of interest here are the `getBalance` and `setBalance` methods. They do the obvious things.

The `run` method repeatedly deducts `change` from the account by first executing

```
account.getBalance()
```

to get the balance and then executing

```
account.setBalance(balance+change)
```

to update the balance.

The final balance in the account should be $1000, the same amount started with, as one of the ATM objects withdraws $100 ten times, while the other deposits $100 ten times. If you run this code enough, however, you will discover that the answer does not always turn out to be $1000! What is causing the problems? Look at the program to see if you can determine what is going wrong before reading further.

The error occurs because two different threads (objects of type `Thread`) are updating the same variable, `balance`. Each gets the balance from the bank, adds in its change, and then tells the bank the new balance. However, it can happen that both ATMs get the balance before either of them has the opportunity to update the balance.

For example, suppose ATM1 gets the balance of $1000, while ATM2 "simultaneously" gets the balance of $1000 (they aren't actually happening simultaneously if there is only one processor, but for our purposes it can be helpful to think that way). Now ATM1 adds $100 to the balance and updates the balance to $1100. ATM2 then removes $100 from the balance that it originally got ($1000) and updates the balance to $900. Thus if the interleaving of operations of ATM1 and ATM2 are such that both get balances before either registers the new balance, the final balance will not reflect one of the two operations. This is called *interference* and is an example of a *race condition*, as the two threads are essentially racing each other to query and update the balance, and whoever updates last has their value remain.

Clearly this is a problem, yet we would like to have the operations of the two ATMs interleaved. (We could just run ATM1 to conclusion before starting up ATM2, but this does not model the usage of ATM's properly.)

We would like to ensure that if ATM1 queries the balance with the intent to change it and set a new balance, that ATM2 does not read the original balance. It is when both read the old balance and both update that one of the transactions is lost. We attempt to remedy this by replacing the `setBalance` method with a `changeBalance` method:

See Example: ATM2

Now rather than having separate methods to get and set the balance, we have a single method which takes the amount of change and updates the balance. Because the getting and setting are no longer separated by distinct method calls, the chances of interference are not as great. However there is still the opportunity of interference between the calculation of the new balance and the update of the value. We have artificially increased the chances of this by adding the `pause` between the two.

Even if we remove that, we reduce the time between the calculation of the old balance and setting of the new balance, but still allows the (at least theoretical) possibility of interference between the calculation of `balance+change` and the assignment of that value to `balance`. To be absolutely safe, we must ensure that only one thread at a time can execute the method `changeBalance`. In

general, this idea is called *process or thread synchonization*. We can do this in Java by using the keyword `synchronized`.

If we attach the keyword `synchronized` to methods in a class, then Java will ensure that only one thread at a time will be executing any of those methods. For example we can label both `getBalance` and `changeBalance` as `synchronized`.

See Example: ATM3

Now if a thread associated with one ATM object is executing either of these methods, then no other thread can execute either of the methods. For example, if `ATM1` is executing `changeBalance`, then `ATM2` will not be allowed to execute either `changeBalance` or `getBalance`. Instead it will wait until `ATM1` has finished executing that method and then execute the desired method. (The operating system is given the responsibility of scheduling the threads' access to the processor.)

A thread executing either `changeBalance` or `getBalance` has no impact on another thread's attempts at executing any of the non-synchronized methods of the program. Thus the user-interface thread can by executing the `actionPerformed` or `startATMs` method while `ATM1` is executing `changeBalance`.

Because of the use of `synchronized`, neither thread can interfere with the other, ensuring that the final answer is the correct one. However, there is one disadvantage of using synchronized methods – they cut down on the amount of concurrency in the system. This may slow down the execution of the program, as one thread may be waiting for an operation to complete (e.g., a write to the screen or a read from a file), while another might be ready to do something. The second thread may be ready to use the processor, but if it is ready to execute a synchronized method and the other thread is executing a synchronized method of the same object, then it may be blocked from executing.

This example may seem a bit contrived – we carefully made sure the pause times for the two ATMs were the same and added a random pause inside the methods that change the balance to increase the chances of interference. However, the interference could happen in each of our cases (except the one with the `synchronized` modifiers) even without the careful attempts to increase the chances. Has anyone ever had some big program, even maybe a commercial program, crash in an unexpected and non-reproducable manner? Perhaps a browser or even Windows itself? There's a pretty good chance that a lot of those kinds of crashes are the result of concurrency not being dealt with carefully enough. Most of the time, things are fine – but once in a while just the right combination of things is happening and there you go. Crash and burn, and in the worst case, read that literally.

There are many other complications involved in the use of concurrency – too many to go into detail here. Concurrency can be quite challenging, and inattention to details may result in programs that don't work as expected. Most of the programs that we have had you write which involve active objects have been designed so that no interference is possible. But it is important to this about the possibilities of interference when you use active objects and threads. Advanced Computer Science courses study concurrency in much more detail, including other problems that may arise and the techniques to deal with them.