OS Week 5 - Lab

Synchronization and Deadlock

Introduction

Last week's lab illustrated *mutual exclusion*, which is a type of *synchronization* used to control interference between threads. If we use synchronization carefully, it helps to eliminate some types of non-deterministic behaviour in concurrent programs ("Race Conditions"). But, as discussed in the Week 4 lecture, synchronization brings with it another kind of problem peculiar to concurrent systems - the possibility of *deadlock*

Before experimenting with deadlock in Java, we discuss one of the "other kinds of synchronization" (i.e. besides mutual exclusion), discussed in last week's lecture.

Notification

In the latter part of this lab, we will be modelling the locking of resources, and our implementation technique will be similar to the use of semaphores for mutual exclusion in last week's lab. But mutual exclusion is only one possible kind of synchronization between threads. Another important synchronization discussed in the week 4 lecture was *notification synchronization*.

Some pseudocode at the start of the week 4 lecture (slide 7) looked like this:

```
Thread i      Thread j

.             .
.             .
A             P(S)
V(S)          B
```

where `S` is a semaphore *initialized to 0* (in contrast, a semaphore used for implementing mutual exclusion is typically initialized to 1).

Remember that the *P* operation *decreases* the semaphore, *but not below 0*. If the semaphore still had the value 0 at the time the *P* was executed, the thread would be *blocked* until another thread increased the semaphore. Increasing a semaphore can only be done using the *V* operation.

So, in the example above, if thread *j* hits the line `P(flag)` before thread *i* hits the line `V(flag)`, then thread *j* must wait. In other words, the code section *A* is *guaranteed* to be executed before the code section *B*.

Convince yourself that this kind of synchronization *is* different to mutual exclusion. In Java, for example, there is no direct way of implementing such synchronization *just* by using synchronized methods - unless you use a CPU-intensive wait loop, or new Java primitives we haven't discussed in these labs. What is going on here is more like a kind of *notification* between threads. One thread notifies the other when it may proceed.

Experiment 1

The following Java code illustrates this pattern:

```
import java.util.concurrent.*;

public class Notification {
    public static void main(String args []) throws Exception {
        MyThreadI thread1 = new MyThreadI();
        MyThreadJ thread2 = new MyThreadJ();

        thread1.start();
        thread2.start();

        thread2.join();
        thread1.join();
    }
}

class MyThreadI extends Thread {
    static Semaphore flag = new Semaphore(0); // initialize to 0
    void delay() {
    int delay = (int) (1000 * Math.random());
    try {
        Thread.sleep(delay);
    }
    catch(Exception e) {
    }
}
```



```
    public void run() {
        try {
            delay();

            System.out.println("Thread I signalling") ;

            flag.release(); // V(flag)
        }
        catch(Exception e) {}
    }

}

class MyThreadJ extends MyThreadI {
    public void run() {
        try {
            delay();

            System.out.println("Thread J starting to wait") ;
            flag.acquire(); // P(flag)
            System.out.println("Thread J finished waiting") ;
        }
            catch(Exception e) {}
    }

}
```

Here the two different threads are defined in two different classes called `MyThreadI` and `MyThreadJ`. Some common properties like static variables are defined in the base class `MyThreadI` and *inherited* by `MyThreadJ`. If you aren't familiar with object oriented programming and inheritance, don't worry about how this works - the upshot is just that the two classes can both access the same semaphore, and use the method `delay`. Concentrate on the code inside the `run` methods for the two threads (highlighted in yellow).

Run the program above several times. Sometimes the first thread wins and the message:

```
Thread I signalling
```


appears first. Sometimes the second thread wins, and the message:

```
Thread J starting to wait
```

appears first. But in either case, the message:

```
Thread J finished waiting
```

cannot appear until after the first thread has signalled. (Your lab book should record evidence of both scenarios.)

Deadlock with Two Resources

Experiment 2

This rather long example illustrates a deadlock scenario with two resources:

```
import java.util.concurrent.*;

public class Deadlock2 {
    public static void main(String args []) throws Exception {
        MyThreadA thread1 = new MyThreadA();
        MyThreadB thread2 = new MyThreadB();

        thread1.start() ;
        thread2.start();

        thread2.join() ;
        thread1.join();
    }
}

class MyThreadA extends Thread {

    static Semaphore semR = new Semaphore(1);
    static Semaphore semS = new Semaphore(1);
    void delay() {
        int delay = (int) (1000 * Math.random());
        try {
            Thread.sleep(delay);
        }
        catch(Exception e) {
        }
    }

    public void run() {
        try {
            while(true) {
```



```
                delay() ;

                System.out.println("Thread A waiting for R");
                semR.acquire();
                System.out.println("Thread A acquired R");
                 delay();
                System.out.println("Thread A waiting for S");
                semS.acquire();
                System.out.println("Thread A acquired S");
                delay();
                semS.release();
                System.out.println("Thread A released S");

                semR.release();
                System.out.println("Thread A released R");
            }
        }
        catch(Exception e) {}
    }

}

class MyThreadB extends MyThreadA {

    public void run() {
        try {
            while(true) {
                delay() ;

                System.out.println("Thread B waiting for S");
                semS.acquire();
                System.out.println("Thread B acquired S");
                delay();
               System.out.println("Thread B waiting for R");
                 semR.acquire();
                System.out.println("Thread B acquired R");
                delay();
                semR.release();
                System.out.println("Thread B released R");

                semS.release();
                System.out.println("Thread B released S");
            }
        }
        catch(Exception e) {}
    }

}
```


Thread *A* acquires resource *R* and resource *S* (in that order). These resources are modelled by the semaphores semR and semS. Thread *B* acquires the same resources but in the opposite order.

Both threads loop indefinitely until the system "eventually" deadlocks. Because of the way average delay periods work out here, more often than not you may see the deadlock in the first iteration. The output may look something like this:

```
Thread B waiting for S
Thread B acquired S
Thread A waiting for R
Thread A acquired R
Thread B waiting for R
Thread A waiting for S
```

The system is now deadlocked because both threads are waiting for resources the other thread has acquired.

When a program deadlocks, you can usually kill the program using NetBeans. A more extreme way to recover is by using Windows Task Manager. On the "Processes" tab, find the java.exe process and end it.

Run the program several times. Threads will sometimes manage to acquire and release the resources they want at least once before the system eventually deadlocks. By tweaking the delays we could make deadlock more or less likely. As with all non-deterministic behaviour, rare occurrences are more insidious.

Reflective Questions

We have now encountered two fundamental kinds of error that are unique to concurrent systems - race conditions and deadlocks. Do you see any similarities, differences, or other relationships between these two kinds of problems?

Exercises

1. Choose one of the longer runs of `Deadlock2` you produced (i.e. one that doesn't deadlock "immediately"). You may want to save the output of the program in a file somewhere - e.g. by copying and pasting.

Now create an *animation* of the *Resource Allocation Graph (RAG)* for your run. Here is a rather dull example that I created in PowerPoint. For your own animation, you should *choose a more interesting run*, in which some threads successfully acquire and release the resources they need, *at least once*.

You are welcome to use your imagination to make your animation more interesting. But *do* stick to the conventions on how a RAG is represented, as described in the lecture.

Finally, you certainly don't have to use PowerPoint for this animation. But if you do there are some hints below.

Hints: using squares, circles, and arrowed lines from the "Shapes" menu, draw and label your threads and resources, together with *all possible* arrows connecting them. Then go to the "Animations" tab. In the order that events occur in your run, select the relevant arrow, then click "Add Animation". The effects you may want to add include "Entrance -> Appear" and "Exit -> Disappear". But be creative!).

Attach your animation to your Wiki.

2. Extend `Deadlock2` to `Deadlock3`, which involves threads *A*, *B* and *C*, competing for the resources *R*, *S* and *T*, which they try to acquire *two at a time* (*A* tries to acquire *R* and *S*, etc), as illustrated in the week 4 lecture (slides 16-17).

