OS Week 4 - Lab

Mutual Exclusion

Introduction

Mutual exclusion is a fundamental technique used in concurrent systems. In concurrent programming, mutual exclusion is a principal way of avoiding *race conditions*, which was illustrated in the last week's lab. Make sure you have a good grasp of what was happening in last week's examples with a shared counter before carrying on to the exercises below.

There are several different ways to *implement* mutual exclusion. In last week's lecture, we discussed "memory-based" algorithms like Peterson's Algorithm. These implement mutual exclusion by adding extra shared variables to "guard" critical regions. We also discussed system and language interfaces that provide mutual exclusion as a service. These included *semaphores* and *monitors*.

This lab will start by considering two attempted solutions to an exercise given last week because that disposes of the first of the implementation types - "memory-based" algorithms.

An Attempt at Mutual Exclusion

While reading this section and the next, you may want to refer to week 3 lecture slides - especially slides 11 to 15.

Last week we gave an example we called the "Slow Race". This had a *critical section* like this:

```
int x = count;
System.out.println("Thread " + name + " read " + x) ; delay();
count = x + 1;
System.out.println("Thread " + name + " wrote " + (x + 1));
```

Remember that a critical section is a section of code that accesses a shared data structure. In our case the shared data structure is the "counter" variable `count`.

An exercise we set was to find a scheme for adding extra shared variables to "guard" the critical section - in other words, to implement mutual exclusion.

One plausible attempt is to use a single "lock" variable. A thread sets the lock "true" immediately before entering the critical section and unsets it on leaving the critical section. Before setting the lock variable, the thread waits until it is false. The idea is that in this way the thread never enters a critical section while another thread is inside one.

Experiment 1

Try running this version of the "slow race":

```
public class SlowRace {
	public static void main(String args []) throws Exception {
	MyThread.count = 0;

	MyThread thread1 = new MyThread();
	thread1.name = "A";

	MyThread thread2 = new MyThread();
	thread2.name = "B";
	thread1.start();
	thread2.start();
	thread2.join();
	thread1.join();
	System.out.println("MyThread.count = " + MyThread.count);
	}
}

class MyThread extends Thread {
	volatile static int count;
	volatile static boolean lock = false;
	String name;

	public void run() {
		for(int i = 0 ; i < 10 ; i++) {
		delay();

		while(lock) {}    // wait until lock is false
		lock = true;      // claim access to critical region

		// start of critical section
		int x = count;
		System.out.println("Thread " + name + " read " + x);
		delay();
		count = x + 1;
```



```
            System.out.println("Thread " + name + " wrote " + (x +
      1));
            // end of critical section
            lock = false;     // release access to critical region
            }
      }

      void delay() {
            int delay = (int) (1000 * Math.random());
            try {
            Thread.sleep(delay);
            }
            catch(Exception e) {
            }
      }
}
```

Look back at the original "Slow Race", and identify the few lines that have changed. If the `lock` is `true`, the statement: while(lock) {} // wait until the lock is false goes into a loop that can only end when the other thread changes `lock` to `false`.

Run this new version of `SlowRace`. You will *may well* see that the final value for the count is 20, which is what we expect if there is no race condition.

So - have we implemented mutual exclusion, and solved the race condition? Sadly, no.

Experiment 2

In the code above, modify these lines:

```
            while(lock) {}    // wait until lock is false
            lock = true;      // claim access to critical region
```

by adding a delay in between the two, like this:

```
            while(lock) {} // wait until lock is false

            delay() ;
            lock = true ;   // claim access to critical region
```

Run the new version. You will now *probably* find that the race condition has reappeared (run the program a few times if you don't see it immediately).

What has happened here is that the other thread has come in again and tested the value of the `lock` variable, *between* the completion of the wait loop and the assignment of `true` to the `lock` variable. Both threads have then been able to get inside their critical sections at the same time.

(A way of thinking about this is that our attempt at implementing mutual exclusion *itself* contained a race condition.)

You might say "well just leave out the delay, and everything will be fine". But this won't do. Correctness of a program - concurrent or otherwise - *must not* depend on the exact timing of execution. By adding a delay we have only changed the timing. The fact is that if we ran our original program many times, threads would eventually - by chance - be scheduled in such a way that mutual exclusion failed. One failure of mutual exclusion in a long-running concurrent program (like an operating system or an embedded system) could be enough to crash the system.

Peterson's Algorithm in Java

Experiment 3

This program is a little more complicated, but you should try running it:

```
public class Peterson {

    public static void main(String args []) throws Exception {
    MyThread.count = 0;
    MyThread thread1 = new MyThread() ; thread1.name = "A";
    thread1.id = 0;

    MyThread thread2 = new MyThread() ; thread2.name = "B";
    thread2.id = 1;

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

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

    System.out.println("MyThread.count = " + MyThread.count);
    }
}

class MyThread extends Thread {
```



```
    volatile static int count;
    String name;
    int id;

    public void run() {

        for(int i = 0 ; i < 10 ; i++) {
            delay();
            beginCriticalSection();
            int x = count;
            System.out.println("Thread " + name + " read " + x);
            delay();
            count = x + 1;
            System.out.println("Thread " + name + " wrote " + (x + 1));

            endCriticalSection();
        }
    }

    // Peterson's algorithm

    volatile static int turn;
    volatile static boolean [] interested = new boolean [2];

    void beginCriticalSection() {
        interested [id] = true;
        int jd = 1 - id;
        turn = jd;
        while(interested [jd] & turn == jd) {}
    }

    void endCriticalSection() {
        interested [id] = false;
    }

    void delay() {
        int delay = (int) (1000 * Math.random());
        try {
            Thread.sleep(delay) ;
        }
        catch(Exception e) {
        }
    }
}
```

To make this work we gave the threads integer ids (0 and 1). If you run this program you should *not* see the race condition.

Remembering experiences you may have had with the simple lock variable, you should be suspicious. But actually, Peterson's algorithm is correct, and this program

really should *not* have any race condition. Mutual exclusion has been successfully enforced.

One of the exercises at the end of this lab asks you to provide some experimental evidence for this.

Semaphores in Java

While reading this section, you may want to refer to week 3 lecture slides - especially slides 34 to 36.

Semaphores are implemented in Java by the class `Semaphore` from the package `java.util.concurrent`. We can declare a semaphore called `mySemaphore` with an initial value of `N` like this:

```
Semaphore mySemaphore = new Semaphore(N);
```

The *P* operation that decreases the semaphore is implemented in Java by the method `acquire`. The *V* operation that increases the semaphore is implemented by the method `release`.

Experiment 4

Implementing mutual exclusion using semaphores is quite intuitive. Start from the original code for the "Slow Race", and change the definition of `MyThread` to:

```
class MyThread extends Thread {
    volatile static int count;
    static Semaphore mySemaphore = new Semaphore(1);
    String name ;
    int id ;
    public void run() {
        try {
            for(int i = 0 ; i < 10 ; i++) {
                delay() ;
                mySemaphore.acquire(); // P - decrease semaphore
                int x = count ;
                System.out.println("Thread " + name + " read " + x);
                delay() ;
                count = x + 1 ;
                System.out.println("Thread " + name + " wrote " + (x + 1));

                mySemaphore.release(); // V - increase semaphore
            }

        }
        catch(Exception e) {}
```



```
    }

    void delay() {
        int delay = (int) (1000 * Math.random());
        try {
        Thread.sleep(delay);
        }
        catch(Exception e) {
        }
    }
}
```

You will also need to add the declaration:

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

near the start of the program - immediately after the package declaration if you have one.

Here `mySemaphore` is a static variable so access to it is shared by the threads. The only annoyance here is that `acquire` is declared to throw a checked exception, which we have to catch in a `try/catch` block. This is a feature of Java programming that is unrelated to concurrency. If you aren't already familiar with it, just take my word that it has to be this way.

Run the program. You will see no evidence of a race condition, and in fact, the semaphore has provided mutual exclusion.

Note the `acquire()` method in the Java implementation is equivalent to the operation we have usually called *P* and the `release()` method is equivalent to what we have usually called *V*.

Synchronized Methods in Java

Here again, you may want to refer to week 3 lecture slides - especially slides 36 to 39.

We have gone a long way round to get here, but in the end, Java has a mechanism for mutual exclusion *built into the language*. Methods on a class can be declared to `synchronized`. No two calls to synchronized methods on the same object can "get inside" the object at the same time. For static synchronized methods, no two *static* synchronized calls can be operating "inside the class" at the same time.

Experiment 5

Try this variation on our very first race condition example from last week:

```
public class Race {

    public static void main(String args []) throws Exception {
        MyThread.count = 0;
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();
        thread1.start() ; thread2.start();

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

        System.out.println("MyThread.count = " + MyThread.count);
    }
}

class MyThread extends Thread {
    static volatile int count;
    String name;


    public void run() {

        for(int i = 0 ; i < 100000000 ; i++) {
            increment();
        }
    }
```



```
    synchronized static void increment() {
        int x = count;
        count = x + 1 ;
    }
}
```

We have reverted to the original "fast race", because here we would have had to refactor some of our supporting functions for running delays and printing thread names, and that would just be distracting at this stage.

Experiment 6

Now run exactly the same program without the `synchronized` modifier on the definition of `increment`.

Reflective Questions

*What **two** differences do you notice between the **behaviour** of the versions of the `Race` program with and without `synchronized` methods (experiments 5 and 6)? Why do you think I reduced the loop count of the "fast race" from one billion to a hundred million...?*

Exercises

1. Modify the implementation of Peterson's algorithm above, placing delays anywhere within the code where you think they might make a difference - for example you might scatter them liberally inside `beginCriticalSection` and `endCriticalSection`.

   Increase the number of counter updates in each thread to a hundred or more, if you like. You can reduce the average delay period if necessary.

   Can you break the mutual exclusion in this way? (I haven't managed to myself; but my implementation is slightly cavalier with the definition of the volatile variables - maybe you will find a flaw.)

2. An exercise at the end of the Week 3 lab introduced a program with a race condition involving a queue data structure.

   Use one or more of the techniques you have learned about since last week's lab to fix the race condition in this code.

3. (Optional.) Write a program with two threads sharing access to the queue data structure above. One thread should write messages to the queue, and the other should remove the messages from the queue and print them out.

   This is slightly tricky because it involves two threads with different behaviour (different `run` methods) sharing access to the same shared data structure. One solution is to have the second thread implemented by a class that extends the `MyThread` class of the first thread (so it shares static variables). Just override the `run` method in the second class with a different behaviour.

   There is a certain shortcoming in this method of passing messages between two threads. Can you identify what it is?

