Department of Computer Science
CS240 Operating Systems, Communications and Concurrency - Dermot Kelly
Practical Number 9

Template Code Files for this Assignment are given on Moodle.
The **Readers/Writers problem**, covered in lectures, describes a synchronisation problem where some processes classed as readers and other processes classed as writers wish to share access to data items. Using the policy that **no reader should have to wait provided that a writer is not already using the shared item**, demonstrate a simulation of this synchronisation problem as a multithreaded Java program.

The methods needed for managing the data synchronisation using the policy described above (First Readers/Writers problem) are encapsulated in the class given below.

```
public class DataAccessPolicyManager
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore wrt;

   public DataAccessPolicyManager () {
      readerCount = 0;
      mutex = new Semaphore(1);
      wrt = new Semaphore(1);
   }

   public void acquireReadLock() {
      mutex.acquire();
      ++readerCount;
      if (readerCount == 1) // This is the first reader
         wrt.acquire();
      mutex.release();
   }

   public void releaseReadLock() {
      mutex.acquire();
      --readerCount;
      if (readerCount == 0) // Last reader
         wrt.release();
      mutex.release();
   }


   public void acquireWriteLock() {
      wrt.acquire();
   }


   public void releaseWriteLock() {
      wrt.release();
   }
}
```

Your implementation should **complete the definition of the class**

```
public class Reader extends Thread {

	DataAccessPolicyManager lockManager;

	public Reader (DataAccessPolicyManager lockManager) {
	}

	public void run() {
	}
}
```

and the definition of the class

```
public class Writer extends Thread {

	DataAccessPolicyManager lockManager;

	public Writer (DataAccessPolicyManager lockManager) {
	}

	public void run() {
	}
}
```

Each Thread subclass will have a `run()` method consisting of an infinite loop. Reading or writing activity in the loop can be simulated by a random sleep period (similar to what was done in Dining Philosophers and Producer/Consumer), but before doing reading or writing, the thread should call the appropriate `lockManager` entry method. Then it should print out either "Reader acquired read lock" or "Writer acquired write lock", then do a random sleep to simulate the activity and then print "Reader done, releasing read lock" or "Writer done, releasing write lock" and then call the appropriate lockManager exit method. Before looping around to read or write again, the run method should do a second random sleep period. Refer to the slides of lecture 16 and the comments within the given code for guidance on the code sequence and use of the lockManager within the run() methods.

Each `Thread` subclass above must have a constructor with the same name as the class and a `run()` method and should be stored in a file where the classname matches the file name. A shared reference to a common `DataAccessPolicyManager` object (instantiated in the `main()` method of your simulation program) will be passed to the constructor for `Reader` and `Writer` threads (also instantiated in `main()`) for using locally in each thread to synchronise its behaviour. Finish the constructors and `run()` methods above.

When the `Reader` and `Writer` classes are complete you should then complete the `ReadersWritersSimulation` class containing a `main()` method which should create a `DataAccessPolicyManager` object and a number of `Reader` and `Writer` threads with access to that `DataAccessPolicyManager` Object.

```
public class ReadersWritersSimulation {
	public static void main (String args[]) {
	}
}
```

Again, refer to the lecture slides for guidance completing this main() method.

You will also need the `Semaphore` class overleaf to be in your current directory, this was used in the `DiningPhilosophers` simulation last week:-

```
/* The Semaphore class contains methods declared as
synchronized. Java's locking mechanism will ensure
that access to Semaphore methods is mutually exclusive
among threads that invoke these methods.
*/
class Semaphore {
	private int value;

	public Semaphore(int value) {
		this.value = value;
	}

	public synchronized void acquire() {
		while (value == 0) {
			try {
				// Calling thread waits until semaphore is free
				wait();
			} catch(InterruptedException e) {}
		}
		value = value - 1;
	}

	public synchronized void release() {
		value = value + 1;
		notify();
	}
}
```

Implementation of solution to 2nd Readers/Writers Problem

When your simulation of the Readers/Writers problem using the given prioritised readers policy is working , complete the `DataAccessPolicyManager2` class so that it prioritises writers instead of readers.

This implementation merely requires some extensions to the code of the `DataAccessPolicyManager` class, as was described in class. Refer to lecture 16 slides.

Remember to replace `DataAccessPolicyManager` with `DataAccessPolicyManager2` in other parts of your code as needed.

Then run your simulation again to check everything is working but this time using the `DataAccessPolicyManager2` class which prioritises writers.

SUBMIT ON MOODLE BY TUESDAY 18th May 1pm
One text file containing:-

1) The completed Reader thread class
2) The completed Writer thread class
3) Your final completed ReadersWritersSimulation Class
4) The completed DataAccessPolicyManager2 Class