Concurrent Programming (Part II)
Lecture 9: New Concurrency Classes & Readers/Writers Example

Dr Kevin Bryson
K.Bryson@cs.ucl.ac.uk
Room 8.04

Course Web Site on Moodle
http://moodle.ucl.ac.uk/course/view.php?id=753
Enrolment Key: ATOMIC

Overview of Lecture

- In this lecture we will examine some of the new concurrency classes available in the `java.util.concurrent` and `java.util.concurrent.locks` packages.
- We will examine the behaviour of these classes using simple text-based examples (see listings).
- ***Predicting the expected behaviour will play a key role.***
- We will then examine the last of the classes (ReentrantReadWriteLock) in more detail and see how it would be implemented using traditional Java concurrency primitives.

Lock Interface and the Reentrant Lock Class

- This class can be used to lock critical sections within the Java code.
- The key methods are lock() and unlock(), although other methods allow timed waiting for the lock and ‘polling’ the lock (tryLock, etc.)
- One of the constructors can take the boolean value true which turns the lock into a ‘fair’ lock
  - How does this change its behaviour ?

- We will look at the example code and predict its behaviour.
- What are these locks used for ?

Semaphore Class

- A Semaphore object is constructed with a given number of ‘permits’.
- The key methods are acquire() and release(), although other methods allow you to obtain a number of permits atomically and also ‘polling’ to see if permits are available (tryAcquire, etc.)
- One of the constructors can take the boolean value true which turns the semaphore into a ‘fair’ semaphore
  - How does this change its behaviour ?

- We will look at the example code and predict its behaviour.
- What are these semaphores used for ?

CyclicBarrier and CountDownLatch Classes

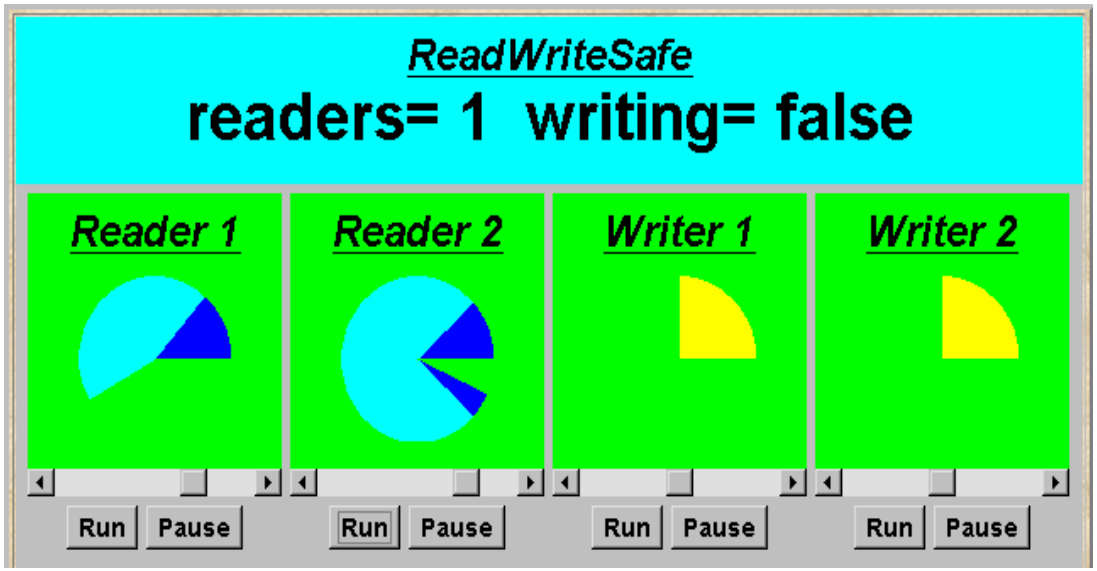
- These act as barriers to threads – it causes them to wait until a condition is true.
- The key method is await() – causing the Thread to wait until the required condition. The CountDownLatch also has a countDown() method.
- How are these two classes different ?

- We will look at the example code and predict its behaviour.
- What are these classes used for ?

ReadWriteLock Class

- Sometimes we have the situation where a number of Threads want to read the values of an object, and only infrequently will a Thread want to write to the object.
- In this case it is more efficient to allow multiple Threads to simultaneously read the state of the object, while only allowing one Thread mutual exclusive access to write to the object.
- The key methods are readLock() and writeLock() which obtain, respectively, read and write locks (which implement the Lock interface). These can then be used to lock the object for reading and writing.
- We will look at the example code and predict its behaviour.

Readers and Writers (Magee & Kramer, Ch. 7)



Light blue indicates database access.

A shared database is accessed by two kinds of processes. **Readers** execute transactions that examine the database while **Writers** both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

Readers/Writers– the Monitor Interface

We concentrate on the monitor implementation:

```
interface ReadWrite {
      public void acquireRead()
          throws InterruptedException;
      public void releaseRead();
      public void acquireWrite()
          throws InterruptedException;
      public void releaseWrite();
}
```

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface.

Readers/Writers - a safe implementation

Unblock a single writer *when no more readers.*

(conditional notification)

Unblock **all** readers

```
class ReadWriteSafe implements ReadWrite {
  private int readers =0;
  private boolean writing = false;

  public synchronized void acquireRead()
                throws InterruptedException {
    while (writing) wait();
    ++readers;
  }
  public synchronized void releaseRead() {
    --readers;
    if(readers==0) notify();
  }
  public synchronized void acquireWrite()
                 throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }
  public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
  }
}
```

Readers/Writers - a safe implementation

BUT POSSIBLE WRITER STARVATION
if the number of readers never drops to 0.

How to solve?

```
class ReadWriteSafe implements ReadWrite {
  private int readers =0;
  private boolean writing = false;

  public synchronized void acquireRead()
                throws InterruptedException {
    while (writing) wait();
    ++readers;
  }
  public synchronized void releaseRead() {
    --readers;
    if(readers==0) notify();
  }
  public synchronized void acquireWrite()
                 throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }
  public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
  }
}
```

Readers/Writers - a safe implementation without writer starvation

Count the number of waiting threads that want to write.

Note the similarity to the single lane bridge solution ...

```
class ReadWritePriority implements ReadWrite{
  private int readers =0;
  private boolean writing = false;
  private int waitingW = 0; // no of waiting Writers.

  public synchronized void acquireRead()
              throws InterruptedException {
    while (writing || waitingW>0) wait();
     ++readers;
  }
  public synchronized void releaseRead() {
    --readers;
    if (readers==0) notify();
  }
  synchronized public void acquireWrite() {
    ++waitingW;
    while (readers>0 || writing) try{ wait();}
          catch(InterruptedException e){}
    --waitingW;
    writing = true;
  }
  synchronized public void releaseWrite() {
    writing = false;
    notifyAll();
  }
}
```

Summary

- We have reviewed a selection of the Java 5 concurrency classes and seen examples of how they work.
  - ReentrantLock (standard & fair)
  - Semaphore (standard & fair)
  - CyclicBarrier
  - CountDownLatch
  - ReadWriteLock
- We examined the Read/Write lock in more detail – and how this is implemented using intrinsic Java sychronization.