Solution notes

Concurrent Systems and Applications 2005 – Paper 5 Question 4 (JKF)

(*a*) (*i*) Synchronized can be used as a modifier on methods. Also to indicate that a block of code is synchronized: "synchronized {...}". Of all the synchronized methods and synchronized blocks in an object (ie instance of a class) only one of them may be being executed by a thread at once. Other threads must wait until the active thread leaves the method/block. There is no guarantee which will be the next thread to acquire the lock, nor any language-level means to control it with absolute certainty. If the method/block is static then the scope of the exclusion is across all instances of that data type, just as static methods are shared between all instances of their class.

(*ii*) Re-entrant means that a thread can repeatedly acquire a mutex that it already holds, it doesn't get blocked if the thread holding the lock is itself. This allows us to call one synchronized method from another and avoids any need for messy wrappers to provide entry/exit protocols.

(*iii*) Accesses to long and double fields are not atomic in Java. We use synchronized accessor methods to read/write their values in thread-safe ways.

(*iv*) In Java 1.5 with generics, the scope of a synchronized method is across all type-specialisations of the generic class. This is much wider scope than people expect!

(*b*) (*i*) The four properties that hold when deadlock exists are: resource requests can be refused, there is no pre-emption of holding resources, resources are held while waiting for others, and a circular wait exists.

(*ii*) All except the circular wait are static properties of the Java language; the last is a property of the program being considered.

(*iii*) Enforcing an ordering for acquiring locks eliminates the possibility of deadlock occuring because circular waits are impossible. However, it means that some threads will be forced to acquire resources before they need them and concurrency might be reduced in those situations.

(*c*) Mutexes can be implemented with counting semaphores like this:

```
class Mutex {
  CountingSemaphore sem;

  Mutex() {
    sem = new CountingSemaphore(1);  // 1 indicates UNLOCKED
  }

  void acquire() {
```

```
    sem.P();       // the method usually referred to as P in literature
  }

  void release () {
    sem.V();       // the method usually referred to as V in literature
  }
}
```

or in terms of hardware CAS operations like this:

```
class Mutex {
  int lockField = 0;

  void lock() {
    while (CAS(&lockfield,0,1) != 0) {
      /* someone else has the lock */
    }
  }

  void unlock() {
    lockField = 0;
  }
}
```