Lecture: Concurrency Theory and Practise
Exercise 2
http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2010ws/

1 Theory

1.1 Atomic Integers

The AtomicInteger class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is **boolean** compareAndSet(**int** expect, **int** update). This method compares the object's current value to expect. If the values are equal, then it atomically replaces the object's value with update and returns true. Otherwise, it leaves the object's value unchanged, and returns false. This class also provides **int** get() which returns the object's actual value.

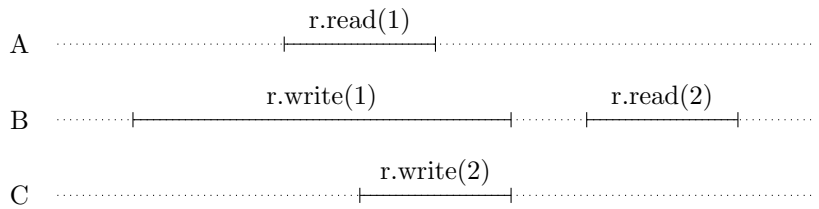
Consider the FIFO queue implementation:

```
1  class IQueue<T> {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicInteger tail = new AtomicInteger(0);
4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
5    public void enq(T x) {
6      int slot ;
7      do {
8        slot = tail.get();
9      } while (! tail.compareAndSet(slot, slot+1));
10     items[slot] = x;
11   }
12   public T deq() throws EmptyException {
13     T value;
14     int slot;
15     do {
16       slot = head.get();
17       value = items[slot];
18       if (value == null)
19         throw new EmptyException();
20     } while (! head.compareAndSet(slot, slot+1));
21     return value;
22   }
23 }
```

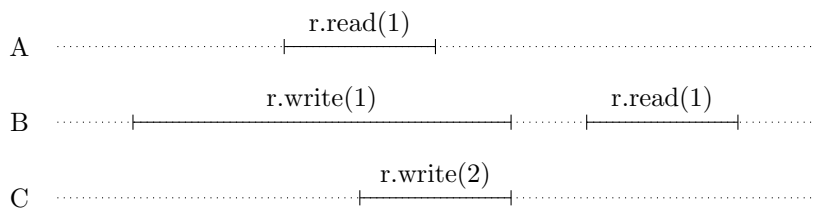
It stores its items in an array items, which, for simplicity, we will assume has unbounded size. It has two AtomicInteger fields: tail is the index of the next slot from which to remove an item, and head is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

1.2 Classifying histories

For each of the histories shown, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer. History 1:



History 2:



1.3 Strange methods....

Consider the following rather unusual implementation of a method m. In every history, the ith time a thread calls m, the call returns after 2^i steps. Is this method wait-free, bounded wait-free, or neither?

1.4 A Bad CLHLock

Explain how the following implementation of CLHLock can go wrong:

```
1  public class BadCLHLock implements Lock {
2    // most recent lock holder
3    AtomicReference<Qnode> tail;
4    //thread-local variable
5    ThreadLocal<Qnode> myNode;
6
7    public void lock() {
8      Qnode qnode = myNode.get();
9      qnode.locked = true; // I am not done
10     // Make me the new tail, and find my predecessor
11     Qnode pred = tail.getAndSet(qnode);
12     // spin while predecessor holds lock
13     while (pred.locked) {}
14   }
15
16   public void unlock() {
17     // reuse my node next time
18     myNode.get().locked = false;
19   }
20   static class Qnode { // Queue node inner class
21     public boolean locked = false;
22   }
23 }
```

2 Practice

2.1 Simple Reader-Writer lock

Re-implement the SimpleReadWriteLock class using Java **synchronized**, wait(), notify(), and notifyAll() constructs in place of explict locks and conditions.

2.2 Sharing bathrooms

In the shared bathroom problem, there are two classes of threads, called MALE and FEMALE.There is a single Bathroom resource that must be used in the following way:

1. persons of opposite sex may not occupy the bathroom simultaneously, and
2. everyone who needs to use the bathroom eventually enters.

The protocol is implemented via the following four procedures: enterMale() delays the caller until it is ok for a male to enter the bathroom, leaveMale() is called when a male leaves the bathroom, while enterFemale() and leaveFemale() do the same for females.

1. Implement this class using locks and condition variables.
2. Implement this class using **synchronized**, wait(), notify(), and notifyAll() constructs.

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

Submission

- Deadline: 23.11.2010
- Please submit solutions in teams of 2 or 3 people. (Submission of single-person teams will not be corrected!)