Concurrent Programming, August–November 2016

Assignment 2, 14 November, 2016
Due: 26 November, 2016

Note: Only electronic submissions accepted, via Moodle.

All exercises are from "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit and the exercise numbers are from the book. Refer to the book if there is any ambiguity.

Chapter 7

Exercise 85 Here is an alternative implementation of `CLHLock` in which a thread reuses its own node instead of its predecessor node. Explain how this implementation 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    public void lock() {
7      Qnode qnode = myNode.get();
8      qnode.locked = true;      // I'm not done
9      // Make me the new tail, and find my predecessor
10     Qnode pred = tail.getAndSet(qnode);
11     // spin while predecessor holds lock
12     while (pred.locked) {}
13   }
14   public void unlock() {
15     // reuse my node next time
16     myNode.get().locked = false;
17   }
18   static class Qnode { // Queue node inner class
19     public boolean locked = false;
20   }
21 }
```

Chapter 9

Variant of Exercise 105 Can the `contains()` method for the fine-grained algorithm [Section 9.5] be implemented without hand-over-hand locking. Explain why, or provide a counterexample.

Chapter 10

Exercise 125 Consider the unbounded queue implementation shown below. This queue is blocking, meaning that the `deq()` method does not return until it has found an item to dequeue.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    ...
5    public void enq(T x) {
6      int i = tail.getAndIncrement();
7      items[i].set(x);
8    }
9    public T deq() {
10     while (true) {
11      int range = tail.get();
12      for (int i = 0; i < range; i++) {
13        T value = items[i].getAndSet(null);
14        if (value != null) {
15          return value;
16        }
17       }
18     }
19   }
20 }
```

The queue has two fields: `items` is a very large array [assume unbounded], and `tail` is the index of the next unused element in the array.

1. Are the `enq()` and `deq()` methods wait-free? If not, are they lock-free? Explain.
2. Identify the linearization points for `enq()` and `deq()`. (Careful! They may be execution-dependent.)