The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 8

July 14, 2009

```
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 }
```

Figure 1: An incorrect attempt to implement a CLHLock.

Exercise 85. Fig. 1 shows 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.

Solution Suppose A acquires and releases the lock. At this point, the same node is referenced by tail and myNode. If A tries to reacquire the lock, it sets the node's locked field to *true* (Line 8), swaps that node with itself (Line 10), implicitly making the node its own predecessor. It then deadlocks waiting for that node to become unlocked.