Sample Solution for Homework 4

Problem 1 AMP, p. 174: Exercise 85: BadCLHLock (5 Points)

One way in which the lock may fail is as follows. Suppose there is only one thread T, which calls the `lock` and `unlock` methods in succession. After `unlock` terminates, we have that `tail` points to `myNode`. Hence, when T calls `lock` again, after executing line 10, we have that `qnode` and `pred` are aliased. Moreover, `pred.locked` is true, even though there is no thread in the critical section. Hence, T will spin forever waiting for `pred.locked` to become false.

Problem 2 AMP, p. 174: Exercise 86: Barriers (7 Points)

The two barrier implementations perform similarly to their lock counterparts. Specifically, the first barrier is more space efficient than the second barrier because it only requires constant space as opposed to linear space in the number of threads. On the other hand, the second barrier will likely perform better under high load because each thread is spinning on its own memory location, avoiding bus traffic due to cache misses when the lock state changes. However, similar to the ALock, the second barrier may suffer from false sharing if multiple array entries occupy the same cache line.

Problem 3 AMP, p. 176: Exercise 91: Testing lockedness (6 Points)

```
// TSA
public boolean isLocked() {
  return state.get();
}

// CLH
public boolean isLocked() {
  return tail.get().locked;
}

// MCS
public boolean isLocked() {
  return tail.get() != null;
}
```

Problem 4 Another bad CLH Lock (7 Points)

The problem with the implementation of the CLH Lock given in Figures 7.9 and 7.10 is that the `tail` reference is not initialized. Thus, in the first call to `lock` after executing line 23, `pred` will be `null` and a null pointer exception will be raised in line 25. A simple fix to this problem is to replace line 2 by:

```
AtomicReference<QNode> tail = new QNode();
```