CS1760: Multiprocessor Synchronization Due 29 October 2022

Homework 4

Professor: Maurice Herlihy *TA: James Scherick*

Problem 1. Propose a way to fix the universal construction described in class and in the text to work for objects with nondeterministic sequential specifications.

Problem 2. Figure 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, and how the MCS lock avoids the problem even though it reuses thread-local nodes.

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

Figure 1: An incorrect attempt to implement a CLHLock.

```
1  public class ConsensusProposal {
2          boolean proposed = new boolean[2];
3          int speed = new Integer[2];
4          int position = new Integer[2];
5          public ConsensusProposal(){
6                  position [0] = 0;
7                  position [1] = 0;
8                  speed[0] = 3;
9                  speed[1] = 1;
10         }
11         public decide(Boolean value) {
12                 int i = myIndex.get();
13                 int j = 1 − i;
14                 proposed[i]=value;
15                 while (true){
16                         position [i]=position [i]+speed[i];
17                         if (position [i]>position [j]+speed[j]) // I am far ahead of you
18                                 return proposed[i];
19                         else if (position [i]<position [j]) // I am behind you
20                                 return proposed[j];
21                 }
22         }
23 }
```

Figure 2: Proposed consensus code for thread $i \in \{0, 1\}$.

Problem 3. Consider the algorithm in Figure 2 for wait-free 2-thread binary consensus.

- Show that the algorithm is consistent and valid (that is, an output value must be an input of one of the threads, and the output values cannot differ).
- Since the algorithm is consistent and valid and only uses read-write registers, it cannot be wait-free. Give an execution history that is a counterexample to wait-freedom.

Problem 4. Imagine n threads, each of which executes method foo() followed by method bar(). Suppose we want to make sure that no thread starts bar() until all threads have finished foo(). For this kind of synchronization, we place a *barrier* between foo() and bar().

First barrier implementation: We have a counter protected by a test-and-test-and-set lock. When a thread reaches the barrier, that thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n.

Second barrier implementation: there is an n-element Boolean array b$[0..n-1]$, all initially *false*. When thread 0 reaches the barrier, it sets b$[0]$ to *true*. Every thread i, for $0 < i < n$, when it reaches the barrier, spins until b$[i-1]$ is *true*, sets b$[i]$ to *true*, and then waits until b$[n-1]$ is *true*, after which it proceeds to leave the barrier.

Explain (briefly) how you would expect each of these barrier protocols to perform on a bus-based cache-coherent architecture. For each protocol, explain whether false sharing is or is not an issue, and if so, how to fix it.