CS1760: Multiprocessor Synchronization Due 25 October 2018

Homework 5

Professor: Maurice Herlihy *TA: Jonathan Lister*

Problem 1. The universal construction for consensus works well for objects with a deterministic specification, but fails in the case of a nondeterministic specification. Why is this? Additionally, propose a way to use the universal construction with objects with nondeterministic sequential specifications.

Hint: See Figure 1 for an example of a nondeterministic specification. You may modify the object, the universal construction protocol, or both.

```
public class TwinStack {
        Stack stacks [] = new Stack[2];
        public void push(int v) { // Specification : Pushes to either stack 1 or 2.
                int idx = randInt() % 2;
                return stacks[idx].push(v);
        }
        public void pop() { // Specification : Pops from either stack 1 or 2.
                int idx = randInt() % 2;
                return stacks[idx].pop();
        }
}
```

Figure 1: A concurrent object with a nondeterministic specification.

Problem 2. Consider a concurrent atomic PeekableStack(k) object: An atomic Stack with an added look() operation. It allows each of n threads to execute push() and pop() operations atomically with the usual LIFO semantics. In addition, it offers a look() operation, the first k calls of which return the value at the bottom of the stack (the least recently pushed value that is currently in the stack) without popping it. All subsequent calls to look() after the first k return *null*. Also, look() returns *null* when the Stack is empty.

- Is it possible to construct a wait-free Queue (accessed by at most two threads) from an arbitrary number of PeekableStack(1) (i.e., with $k = 1$) objects and atomic read-write registers? Prove your claim!
- Is it possible to construct a wait-free n-thread PeekableStack(2) object from an arbitrary number of atomic Stack objects and atomic read-write registers? Prove your claim!

Problem 3. Figure 2 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.

```
public class BadCLHLock implements Lock {
  AtomicReference<Qnode> tail = new AtomicReference<QNode>(new QNode());
  ThreadLocal<Qnode> myNode = new ThreadLocal<QNode> {
    protected QNode initialValue() {
      return new QNode();
    }
  };
  public void lock() {
    Qnode qnode = myNode.get();
    qnode.locked = true;        // I'm not done
    // Make me the new tail, and find my predecessor
    Qnode pred = tail.getAndSet(qnode);
    while (pred.locked) {}
  }
  public void unlock() {
    // reuse my node next time
    myNode.get().locked = false;
  }
  static class Qnode { // Queue node inner class
    volatile boolean locked = false;
  }
}
```

Figure 2: An incorrect attempt to implement a CLHLock.

Problem 4. Design a linearizable isLocked() method that tests whether any thread is holding a lock (but does not acquire the lock). Give implementations for

- a test-and-set spin lock,
- the CLH queue lock, and
- the MCS queue lock.