CS1760: Multiprocessor Synchronization Due 2 November 2023

Homework 4

Professor: Maurice Herlihy *TA: cs1760tas@lists.brown.edu*

Problem 1. (This is a question about the universal construction.) 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 3-thread PeekableStack(2) object from an arbitrary number of atomic Stack objects and atomic read-write registers? Prove your claim!

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.

Problem 3. Design a linearizable isLocked() method that tests whether some thread is holding that 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.

Explain briefly why each one works.

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(). (This kind of barrier is not the same as the memory barriers needed for hardware memory)

1. First barrier implementation: There is a counter protected by a test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n.
2. There is an n-element Boolean array b$[0..n-1]$, all initially *false*. Thread 0 sets b$[0]$ to *true*. Every thread i, for $0 < i < n$, 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.
3. There is a $64 * n$-element Boolean array b$[0..64*n-1]$, all initially *false*. Thread 0 sets b$[0]$ to *true*. Every thread i, for $0 < i < n$, spins until b$[64*(i-1)]$ is *true*, sets b$[i*64]$ to *true*, and then waits until b$[64*n-1]$ is *true*, after which it proceeds to leave the barrier.

   Describe (as succinctly as possible) how you would expect these three barriers to scale (with n) on a on a bus-based cache-coherent architecture with cache line size 64.