**James Scherick** Due 20 October 2022

# Midterm 1

*Professor: Maurice Herlihy*

CS1760: Multiprocessor Synchronization

Please solve *exactly* 3 out of the 4 questions (if you solve more, please specify which 3 you would like us to check).

You may use the textbook and slides, but no search engines or humans!

Good luck!

**Problem 1.** Sketch a proof that sequential consistency is non blocking.

**Problem 2.** Your mission is to transform a sequential stack implementation into a wait-free, linearizable stack implementation, without regard for questions of efficiency or memory use.

You are given a "black-box" Sequence type with the following methods. You can atomically *append* an item to the end of the sequence. For example, if the sequence is $\langle 1, 2, 3\rangle$, and you append 4, the sequence becomes $\langle 1, 2, 3, 4\rangle$. This operation is wait-free and linearizable: if a concurrent thread tries to append 5, the sequence becomes either $\langle 1, 2, 3, 4, 5\rangle$. or $\langle 1, 2, 3, 5, 4\rangle$. Note that Sequence items *do not* have to be integers: they can be any kind of object you like.

You can also iterate through the elements of a sequence. Here, we iterate through a sequence printing each value until we see the string "stop".

```
1 foreach x in s {
2   if (x == "stop") break;
3   System.out.println(x)
4 }
```

(Notice that if you are iterating through a sequence at the same time another thread is appending new values, you might keep going forever.)

Implement a wait-free linearizable stack using an atomic sequence object, and as much atomic read-write memory and sequential Stack implementations as you like.

Your stack should support both push() and pop() operations with the usual meanings. Again, do not worry about efficiency or memory use.

Explain briefly why your construction is wait-free and linearizable (in particular, identify the linearization points).

Hint: where have you seen a similar transformation from sequential to wait-free linearizable data structures? One where memory use and efficiency were ignored?

```
1  class WriteOnceRegister implements Register{
2    // N is the length of the register
3    private SafeMRSWRegister[] r = new SafeMRSWRegister[3];
4
5    public void write(int x) {
6      s[0].write(x);
7      s[1].write(x);
8      s[2].write(x);
9    }
10   public int read() {
11     v2 = s[2].read()
12     v1 = s[1].read()
13     v0 = s[0].read()
14     if (v0 == v1) return v0;
15     else if (v1 == v2) return v1;
16     else return v0;
17   }
18 }
```

Figure 1: Write-Once Register

**Problem 3.** Figure 1 shows an implementation of a multi-valued *write-once*, single-writer, multi-reader register from an array of multi-valued safe, single-writer, multi-reader registers. Remember, there is one writer, who can overwrite the register's initial value with a new value, but once only. You do not know what the register's initial value is.

Is this implementation regular? Atomic?

**Problem 4.** Another way to generalize the two-thread Peterson lock is to arrange a number of 2-thread Peterson locks in a binary tree. Suppose $n$ is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.) For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated.

1. mutual exclusion.

2. freedom from deadlock.

3. freedom from starvation.

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?