CS1760: Multiprocessor Synchronization Due 5 October 2023

Homework 2

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

Problem 1. A *regular* register has the property that if a read() call overlaps a write() call, then the value returned can be either the new value or the old. A read() that does not overlap a write() returns the last value written. (Note that a regular register is not linearizable.)

An *safe* register has the property that if a read() call overlaps a write() call, then the value returned can be arbitrary. A read() that does not overlap a write() returns the last value written.

An *wraparound* register has the property that there is a value v such that adding 1 to v yields 0, not $v + 1$.

If we replace the Bakery algorithm's shared variables with either regular, safe, or wrap-around registers, then does it still satisfy (1) mutual exclusion, (2) first-come-first-served ordering?

You should provide six answers (some may imply others). Justify each claim.

Problem 2. Programmers at the Flaky Computer Corporation designed the protocol shown in Figure 1 to achieve n-thread mutual exclusion.

For each question, either sketch a proof, or display an execution where it fails.

- Does this protocol satisfy mutual exclusion?
- Is this protocol starvation-free?
- Is this protocol deadlock-free?
- Is this protocol livelock-free?

```
1  class Flaky implements Lock {
2    private int turn;
3    private boolean busy = false;
4    public void lock() {
5      int me = ThreadID.get();
6      do {
7        do {
8          turn = me;
9        } while (busy);
10       busy = true;
11     } while (turn != me);
12   }
13   public void unlock() {
14     busy = false;
15   }
16 }
```

Figure 1: The Flaky lock used in Exercise 2.

```
1  class Bouncer {
2    public static final int DOWN = 0;
3    public static final int RIGHT = 1;
4    public static final int STOP = 2;
5    private boolean goRight = false;
6    private ThreadLocal<Integer> myIndex; // initialize myIndex
7    private int last = -1;
8    int visit () {
9      int i = myIndex.get();
10     last = i;
11     if (goRight)
12       return RIGHT;
13     goRight = true;
14     if (last == i)
15       return STOP;
16     else
17       return DOWN;
18   }
19 }
```

Figure 2: The Bouncer class implementation.

Problem 3. Suppose n threads call the Visit() method of the Bouncer class shown in Figure 2. Prove the following:

- At most one thread gets the value STOP.
- At most $n-1$ threads get the value DOWN.
- At most $n-1$ threads get the value RIGHT.

Note that the last two proofs are *not* symmetric.

```
1  class Flakee implements Lock {
2    private int turn;
3    private boolean busy = false;
4    public void lock() {
5      int me = ThreadID.get();
6      do {
7        do {
8          turn = me;
9        } while (busy);
10       busy = true;
11     } while (turn != me);
12   }
```

Figure 3: The Flakee lock.

Problem 4. Consider the following implementation of a Register<> in a distributed, message-passing system. There are n processors $P_0, \ldots, P_{n-1}$ arranged in a ring, where P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link.

Each processor keeps a copy of the shared register.

- To read a register, the processor reads the copy in its local memory.
- A processor P_i starts a write() call of value v to register x, by sending the message "P_i: write v to x" to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_j: write v to x," for $i \neq j$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_i: write v to x," then it writes v to its local copy of x, and discards the message. The write() call is now complete.

Give a short justification or counterexample.
If write() calls never overlap,

- Is this register implementation regular?
- Is it atomic?

If multiple processors call write(),

- Is this register implementation atomic?