Problem 1. 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 $P_{i+1 \mod n}$. 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 $P_{i+1 \mod n}$.
- 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 $P_{i+1 \mod n}$.
- 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?
public class RegBooleanMRSWRegister implements Register<Boolean> {
    ThreadLocal<Boolean> last;
    boolean s_value;  // safe MRSW register
    RegBooleanMRSWRegister(int capacity) {
        last = new ThreadLocal<Boolean>() {
            protected Boolean initialValue () { return false; }
        };
    }
    public void write(Boolean x) {
        if (x != last.get()) {
            last.set(x);
            s_value = x;
        }
    }
    public Boolean read() {
        return s_value;
    }
}

Figure 1: A regular Boolean MRSW register constructed from a safe Boolean MRSW register.

Problem 2. Consider the regular Boolean MRSW construction shown in Fig. 1 discussed in class. True or false: if we replace the safe Boolean MRSW register with a safe $M$-valued MRSW register, then the construction yields a regular $M$-valued MRSW register. Justify your answer.
Problem 3. Consider the atomic MRSW construction shown in Fig. 2 which was discussed in class. True or false: if we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer.
**Problem 4.** For this problem you will be utilizing the Peterson Locking algorithm to create a lock that will provide \( n \)-thread mutual exclusion. Recall that the Peterson lock only works correctly when at most two threads are calling lock(). Should more than two threads attempt to use the Peterson lock in this way, the lock will break and mutual exclusion will be violated. Thus, in order to support a program with \( n \) threads, you will be implementing a TreeLock composed of multiple Peterson locks to guard a critical section where threads will increment a counter. Threads who would like to acquire the TreeLock and enter the critical section will have to successfully acquire a Peterson lock at each level of the tree from leaf to root. Once the root is acquired, the thread has then acquired the TreeLock and can proceed to the critical section.

The stencil code for this assignment is located at the course’s pub directory. To copy the stencil code to your current directory simply type the following command in the terminal:

```
cp /course/cs1760/pub/treeLock/* .
```

Once everything’s been copied in, you’ll be ready to go!

To make your lives a bit easier when doing this problem, below are some tips:

1. The number of PetersonNodes that compose your TreeLock should be a given power of 2. If \( n \) is not a power of 2, you should choose the next highest power of 2 for your nodes. Doing this will ensure that your tree is balanced and help make your life easier when handling edge cases that arise with unbalanced trees.

2. One of the files in the pub directory that we’ve provided is ThreadID.java. You should use this to give your threads their unique Thread IDs starting from 0. The reason why it is advised to use this is explained in the comments of the file.

3. When implementing your tree, you may modify any file of the stencil code as you see fit. Just make sure that your TreeLock satisfies mutual exclusion by the time you finish.

   The files you may modify include PetersonNode.java as well! So feel free to add whatever fields, constructors, methods, etc you think would work better for your tree. However, please note that submissions which modify the form of the Peterson algorithm such that it no longer essentially a Peterson lock will not be accepted and receive no credit.

4. When debugging your program, you may find Java’s built in \texttt{assert} statements to be of great use to you when figuring out what’s going on. If you are running your code from the terminal, note that that you will need to run the \texttt{java} command with the \texttt{-ea} flag to enable assertions. This command is provided in the Makefile of the stencil code.

   It is advised that you sprinkle these statements wherever you’d like to make sure certain conditions are met. When an assertion fails, the thread that triggered it will receive an assertion exception and cease to run. Be aware, however, that the other threads in the system will still keep going as if nothing ever happened.

   To have your assert statements print out useful information such as variable information, Thread ID values, and the like, use the following form in your code:

   ```java
   assert <condition> : "Variable 1:" + var1 + " Variable 2:" + var2
   ```

   When the assertion exception is triggered, the variable information will be printed to standard out during runtime.