CS1760: Multiprocessor Synchronization Due 6 December 2018

Homework 8

Professor: Maurice Herlihy *TA: Jonathan Lister*

Problem 1. Figure 1 shows a proposed implementation of a RateLimiter class, which runs jobs but limits the "weight" of the jobs started per minute using a quota, which is increased to LIMIT every minute by a separate thread. We want to guarantee that jobs will run promptly if there is enough quota. You may assume a fast processor and fair scheduler, so that the RateLimiter reaches a quiescent state (all jobs are sleeping in await() or running), if possible, before each call to increaseQuota().

a) Describe the distributions of weight values ($0 \leq$ weight $\leq$ LIMIT) under which this implementation works or fails and explain why.

b) Fix this implementation so it allows jobs to have any weight value from 0 to LIMIT, and describe how it may impact performance.

```java
public class RateLimiter {
    static final int LIMIT = 100;    // example value
    public int quota = LIMIT;
    private Lock lock = new ReentrantLock();
    private Condition needQuota = lock.newCondition();
    public void increaseQuota() {    // called once per minute
        synchronized(lock) {         // grab the lock
            if (quota < LIMIT) {     // if some of the quote has been used up:
                quota = LIMIT;       // increase quota to LIMIT
                needQuota.signal();  // wake up a sleeper
            }
        }                            // unlock
    }
    private void throttle(int weight) {
        synchronized(lock) {         // grab the lock
            while (quota < weight) { // while not enough quota:
                needQuota.await();   // sleep until increased
            }
            quota -= weight;         // claim my job's part of the quota
            if (quota > 0) {         // if still quota left over:
                needQuota.signal();  // wake up another sleeper
            }
        }                            // unlock
    }
    public void run(Runnable job, int weight) {
        throttle(weight);            // sleep if under quota
        job.run();                   // run my job
    }
}
```

Figure 1: A proposed RateLimiter class implementation

Problem 2. Design a "nested" read-write lock in which a thread must first grab the read lock in order to grab the write lock, and releasing the write lock does not release the read lock. In order for a reader to become a writer with exclusive write access, every other reader must either unlock the read lock or also attempt to lock the write lock. Show that your implementation is correct and has a reasonable fairness guarantee between readers and writers. Assume every thread obeys the protocol we describe (which is to say, no thread tries write-locking without read-locking, or tries to unlock something it doesn't have.)

Problem 3. The combining tree barrier uses a single thread-local sense field for the entire barrier. Suppose instead we were to associate a thread-local sense with each node as in Figure 2.

```
private class Node {
   AtomicInteger count;
   Node parent;
   volatile boolean sense;
   int d;
   // construct root node
   public Node() {
     sense = false;
     parent = null;
     count = new AtomicInteger(radix);
     ThreadLocal<Boolean> threadSense;
     threadSense = new ThreadLocal<Boolean>() {
       protected Boolean initialValue () { return true; };
   };
   }
   public Node(Node myParent) {
     this ();
     parent = myParent;
   }
   public void await() {
     boolean mySense = threadSense.get();
     int position = count.getAndDecrement();
     if ( position == 1) {    // I'm last
       if (parent != null) { // root?
         parent.await ();
       }
       count.set( radix );       // reset counter
       sense = mySense;
     } else {
       while (sense != mySense) {};
     }
     threadSense.set(!mySense);
   }
 }
```

Figure 2: Thread-local tree barrier.

Either:

- Explain why this implementation is equivalent to the other one, except that it consumes more memory, or.

- Give a counterexample showing that this implementation is incorrect.

Problem 4. A *dissemination barrier* is a symmetric barrier implementation in which threads spin on statically-assigned locally-cached locations using only loads and stores. As illustrated in Figure 3, the algorithm runs in a series of rounds. At round r, thread i notifies thread $i + 2^r (\bmod n)$, (where n is the number of threads) and waits for notification from thread $i - 2^r (\bmod n)$.

For how many rounds must this protocol run to implement a barrier? What if n is not a power of 2? Justify your answers.

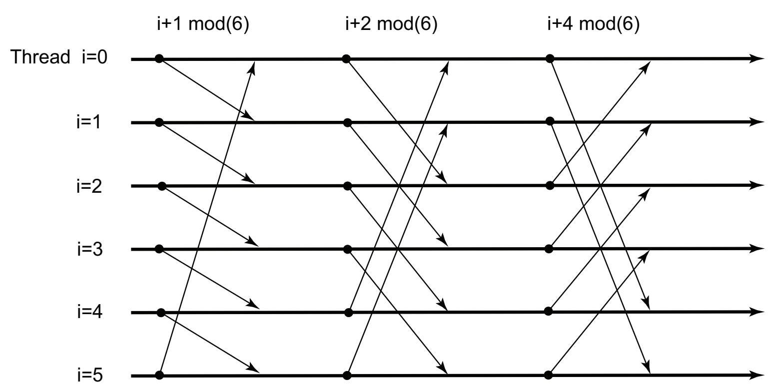

Figure 3: Communication in the dissemination barrier. In each round r a thread i communicates with thread $i + 2^r \pmod n$.