CS1760: Multiprocessor Synchronization Due XX November 2022

Homework 6

Professor: Maurice Herlihy *TA: James Scherick*

Problem 1. Would the contains() method of the lazy and lock-free algorithms still be correct if logically removed entries were not guaranteed to be sorted?

Problem 2. A *savings account* object holds a nonnegative balance, and provides deposit(k) and withdraw(k) methods, where deposit(k) adds k to the balance, and withdraw(k) subtracts k, if the balance is at least k, and otherwise blocks until the balance becomes k or greater.

1. Implement this savings account using locks and conditions.
2. Now suppose there are two kinds of withdrawals: *ordinary* and *preferred*. Devise an implementation that ensures that no ordinary withdrawal occurs if there is a preferred withdrawal waiting to occur.
3. Now let us add a transfer () method that transfers a sum from one account to another:

```
void transfer(int k, Account reserve) {
  lock.lock();
  try {
    reserve.withdraw(k);
    deposit(k);
  } finally {lock.unlock();}
}
```

   We are given a set of 10 accounts, whose balances are unknown. At 1:00, each of n threads tries to transfer \$100 from another account into its own account. At 2:00, a Boss thread deposits \$1000 to each account. Is every transfer method called at 1:00 certain to return?

Problem 3. Consider an application with distinct sets of *active* and *passive* threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A CountDownLatch encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls countDown(), which decrements the counter. Each passive thread calls await(), which blocks the thread until the counter reaches zero. (See Figure 1.)

Provide a CountDownLatch implementation.

```
1  class Driver {
2    void main() {
3      CountDownLatch startSignal = new CountDownLatch(1);
4      CountDownLatch doneSignal = new CountDownLatch(n);
5      for (int i = 0; i < n; ++i)  // start threads
6        new Thread(new Worker(startSignal, doneSignal)).start ();
7      doSomethingElse();              // get ready for threads
8       startSignal .countDown();      // unleash threads
9      doSomethingElse();              // biding my time ...
10     doneSignal.await ();            // wait for threads to finish
11   }
12   class Worker implements Runnable {
13     private final CountDownLatch startSignal, doneSignal;
14     Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
15       startSignal = myStartSignal;
16       doneSignal = myDoneSignal;
17     }
18     public void run() {
19       startSignal .await ();        // wait for driver 's OK to start
20       doWork();
21       doneSignal.countDown(); // notify driver we're done
22     }
23      ...
24   }
25 }
```

Figure 1: The CountDownLatch class: an example usage.