Spring, 2015 ECEN 4003, CSCI 4830-012

Concurrent Programming

Homework 4

Assigned: 04/01/2015; Due: 04/15/2015, before class.
Please submit your solutions through Moodle.

Problem 1

A savings account object holds a non-negative 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. `getbalance()` gives the current balance.

1. Implement this savings account using locks and conditions (use `java.util.concurrent.locks.ReentrantLock`). Test by using the 3 functions.
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.

Problem 2

Consider the following conditions: An enqueuer waiting on a full-queue or a dequeuer waiting on an empty queue sleep indefinitely, unless woken up by another thread. A thread must send a signal ONLY when it adds an element to an empty queue or removes an element from a full-queue.

1. Implement a bounded partial queue using a signaling mechanism that signals to all waiting dequeuers.
2. Implement the bounded partial queue by using a signaling mechanism (your own scheme) that signals to only one waiting dequeuer or enqueuer, and ensure that the lost-wake-up problem does not happen.

Problem 3

We have n threads, each of which executes method `foo()` followed by `bar()`. We want to add synchronization to ensure that no thread starts executing `bar()` until all threads have finished executing `foo()`. To achieve this, we will insert some barrier code between the two methods. Implement the following two schemes for barrier code:

- Use a shared counter protected by test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and repeatedly reads the counter until it reaches n.
- Use an n-element Boolean array `A`. Initially all entries are 0. When thread 0 executes its barrier, it sets `b[0]` to 1, and repeatedly reads `b[n - 1]` until it becomes 1. Every other thread i, repeatedly reads `b[i - 1]` until it becomes 1, then it sets `b[i]` to 1, and repeatedly reads `b[n - 1]` until it becomes 1.

Implement both these schemes in Java. Each of the methods `foo()` and `bar()` just sleeps for 20 milliseconds. Test the two schemes for $n = 16$. Run each scheme at least ten times, measure the total runtime in each test run, discard the highest and lowest values, take the average, and use it to compare the two schemes. Which performs better? Can you explain the reason? You can run the experiments on ecen5033.colorado.edu. Submit the code as well as experimental results.