Programming Paradigms for Concurrency

Homework 1

November 4, 2010; Due November 18, 2010

Homework submissions: by email to the instructor.

Problem 1

Filter lock algorithm is one way to generalize the two-thread Peterson lock to \( n \) threads. Another way, suggested in class, is to arrange a number of two-thread Peterson locks in a binary tree. Suppose \( n \) is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. In the tree-lock’s `lock()` method, a thread acquires every two-thread Peterson lock from that thread’s leaf to the root. The `unlock()` method of the tree-lock unlocks each of the two-thread Peterson locks that the thread has acquired, from the root back to the thread’s leaf.

1. Write an implementation of tree-lock that satisfies mutual exclusion and deadlock-freedom.

2. Benchmark your implementation with \( n = 8 \) and \( n = 64 \) threads and compare its performance to that of the filter lock. For each benchmark, use a test where the lock protects a counter. In each test, the counter should be incremented (and thus the lock acquired and released) a large number of times. Run each test 3 times and report the average time it takes, for the two different algorithm (filter-lock and tree-lock), to complete the counter increment tests.

The course page has a sample Java code for Peterson and filter locks, as well as simple implementations that use these locks. You should run the experiments on a machine with at least eight cores. (You can use the machine `guest.ist.ac.at`.) Submit the code as well as a table with the experimental results.

Problem 2

Consider the lazy list’s validation method. Can it be simplified by dropping the check that `pred.next` is equal to `curr`? After all, the code always sets `pred` to the old value of `curr`, and before `pred.next` can be changed, the new value of `curr` must be marked, causing the validation to fail. Explain the error in this reasoning. Find a counterexample, that is, an execution that would not be linearizable. The code for the lazy list (and the other list algorithms we saw in class) is available from the course website.