COLBY COLLEGE
Computer Science Department

CS357 **Allen B. Downey**
Spring 1998

Assignment 6: Synchronization Primitives

Due: Tuesday 11 April

The purpose of this assignment is to experiment with three implementations of locks, and to measure the frequency of errors caused by incorrect synchronization primitives.

Reading

Read Silberschatz and Galvin, Chapter 6, Process Synchronization.

Programming with POSIX Threads, Chapter 3, Synchronization. (I'll hand this out on Tuesday; you can do the first two parts without it.)

Break the lock

1. Pick up the code from

   `http://rocky.colby.edu/cs357/code/hw6/`

   The file `lock.c` contains an incorrect implementation of a lock written in C. The file `lock.s` contains a correct (I think) implementation of a lock written in Intel x86 assembler code. I will explain the latter in class. The Makefile shows how to make two programs called `goodlock` and `badlock` based on the two versions of a lock. Compile and run both programs.

2. Write a program that uses at least two threads and that accesses a shared variable concurrently. Use your lock to enforce exclusive access to the shared variable. Now test whether your program is in fact achieving mutual exclusion.

   There are many ways to do this. Here is just one suggestion: make the shared variable a counter that hands out unique identifiers in sequence. Create a big array and count the number of times each identifier gets handed out. If the lock is working correctly, every identifier should get handed out exactly once, so each array element should be 1.

3. What is the frequency of synchronization errors using the defective lock implementation?

4. What is the frequency of synchronization errors using the "correct" lock implementation? How can we prove that the implementation is correct?

Estimate the quantum size

What if each thread were to put its pid into the array? Then the array would contain long streaks of one pid followed by long streaks of another pid. Does that give you any ideas?

Check out pthread mutexes

1. Read the handout from *Programming with POSIX threads* that describes pthread mutexes. Print the documentation of the relevant system calls.
2. Make a copy of lock.c and call it mutex.c. Change the implementation of `make_lock`, `acquire` and `release` so that they use pthread mutexes. This should be a trivial implementation, since mutexes are the same thing as locks. All you are doing is creating a veneer that changes the interface.
3. Again, confirm that there are no synchronization errors for a large number of iterations.
4. Time your program to compare the efficiency of my lock implementation with pthread mutexes. Which is faster? Where is the extra time spent, in user code, system code, or operating system overhead?

Have fun! Learn something!