Homework 5

Software Systems
Fall 2006

Allen B. Downey

Due: Monday 6 November

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

Break the lock

Pick up the code by running

```
wget http://wb/ss/code/lock.tgz
tar -xzf lock.tgz
cd lock
```

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

The file `example.c` contains the skeleton of a multi-threaded program with shared state. The shared state is encapsulated in an object called `Environment`. Compile and run `example`.

Experiment 1

1. Starting with a copy of `example.c`, write a program that uses at least two threads and that accesses a shared variable concurrently. Your goal is to get both threads to access the shared variable over and over in order to check whether any synchronization errors occur, and if so, to characterize how often they occur. We will discuss this in class.

2. Run the program and see how frequently synchronization errors occur when there is no synchronization control.

3. Now use the broken lock implementation to enforce exclusive access to the shared variable. Test whether your program is in fact achieving mutual exclusion. What is the frequency of synchronization errors now?

4. Finally, replace the broken lock implementation with the "correct" one. What is the frequency of synchronization errors now? Can we prove that the "correct" implementation is correct?

Experiment 2

1. Read the handout from *Programming with POSIX threads* (butenhof97mutex.pdf) that describes pthread mutexes. Print the documentation of the relevant library functions.

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

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?

Experiment 3

1. Read the handout from *Programming with POSIX threads* (butenhof97condition.pdf) that describes pthread condition variables. Print the documentation of the relevant library functions.

2. Write an implementation of a Semaphore using pthread mutexes and condition variables. Your implementation should provide functions named `make_semaphore`, semaphore_wait, and semaphore_signal, in a file named `semaphore.c`. You should also provide a header file named `semaphore.h` that includes the type definition for `Semaphore` and prototypes of the semaphore "methods".

3. Create a modified version of your program that uses your Semaphore implementation as a mutex.

4. Add a line to the Makefile to build a program named `semlock` using your Semaphore implementation.

5. Run your program and confirm that there are no synchronization errors.

6. Time your program to compare the efficiency of using your Semaphore as a mutex as opposed to using a pthread mutex. Which is faster? Where is the extra time spent, in user code, system code, or operating system overhead?