Homework 5: Synchronization Primitives

cs341
Spring 2002

Allen B. Downey
Computer Science Department

Due Thursday 28 March

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.

In the cow book, please read Chapter 12 on structures and pages 304–316 on modules and Makefiles. Also, in the Anderson handout, read pages 15–18 on pointers to structures. As usual, you will want to investigate other C information as it comes up.

Break the lock

Pick up the code from

`http://rocky.wellesley.edu/cs341/code/hw5/`

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 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.

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

Starting with a copy of `example`, write a program that uses at least two threads and that accesses a shared variable concurrently. 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 there are no synchronization errors, every identifier should get handed out exactly once, so each array element should be 1. Run the program and see how frequently synchronization errors occur.

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?

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

We can use the mechanism we built to make an estimate of the quantum size.

Again, run two threads with shared access to a counter and an array. Each thread should iterate, incrementing the counter and storing values in the array. The threads should store different values so that after they run we can tell which thread wrote which values. We expect the array to contain long streaks of one value followed by long streaks of another value.

Can you estimate the number of instructions each thread runs during a quantum? You can count the instructions in `lock.x86.s`. To count instructions in `main.c`, you might want to compile it into assembly code:

```
gcc -S example.c
```

The result will be in a file named `example.s`.

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.

Experiment 3

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

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?