COLBY COLLEGE
Department of Mathematics
and Computer Science

CS357 **Allen B. Downey**
Spring 1998

Assignment 6: Synchronization Primitives

Due: Thursday 16 April

The purpose of this assignment is to implement locks and semaphores, and to measure the frequency of errors caused by incorrect synchronization primitives.

Reading

Read Silberschatz and Galvin, Chapter 6, Process Synchronization.

Break the lock

1. In class I presented a simple implementation of locks that does not really guarantee mutual exclusion. You can pick up this code from the class web page. It is in a tar file called lock.tar. lock.c is the incorrect implementation; lock.s is (I think) correct.
2. Write a program that uses at least two threads (created with sproc) 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 is handed out. If the lock is working correctly, each array element should be 1.
3. What is the frequency of synchronization errors using the defective lock implementation?
4. Replace the defective lock implementation with the correct one and confirm that the error rate goes to zero.

NOTE: 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?

Implement semaphores

1. Print the man page for `blockproc`. This is a system call processes can use to block themselves (i.e. indicate to the operating system that they would like to yield the CPU) and to wake up other processes.
2. Use the correct implementation of locks and `blockproc` to implement sempahores as discussed in class.

3. Modify your program from the previous section so that it uses semaphores to enforce mutual exclusion rather than using locks. Check that your implementation enforces mutual exclusion. Of course, it is possible that you will see no synchronization errors even if your implementation is incorrect—there is only so much testing can do for you.

4. Time the two versions of the program and see which is faster. I suspect that for the type of program we are talking about, the simple locks will be faster than the semaphores. Write a few sentences explaining under what circumstances you would expect semaphores to provide superior performance.

5. Modify your program so it does different stuff, like CPU-bound loops, or I/O, or whatever you think it will take for the semaphore version to out-perform the lock version.

What to turn in

(You may notice that the following text is the same on every assignment.)

As usual, I would like a single report from each unit, including an explanation of each experiment you ran, and what you discovered. As in the previous reports, try to integrate your tables and code samples with the text rather than making them appendices. That way you have more incentive to be concise and to explain the relevance of everything you present. Since this assignment explicitly involves an implementation, you should take special care to present the code in a way that makes it comprehensible to humans. At the very least, include lots of comments.

Also, if you observe something you can't explain, don't be shy about describing it as it happened. Resist the temptation to sweep unexpected behavior under the rug. Also, don't feel like you have to offer a definitive explanation for everything—it is reasonable to list two or three possible explanations.

As usual: Label your axes. Mark the data points. Explain every number. Along with every number try to give some indication of its accuracy. Don't report more significant digits than you really have. Use more words than numbers. Use more pages for words than for graphs or code.

Have fun! Learn something!