CSCI 3323 (Principles of Operating Systems), Fall 2012
Homework 2

Credit: 20 points.

1 Reading

Be sure you have read Chapter 2, sections 2.1 through 2.3.

2 Problems

Answer the following questions. You may write out your answers by hand or using a word processor or other program, but please submit hard copy, either in class or in my mailbox in the department office.

1. (5 points) In class we discussed a proposed solution to the mutual-exclusion problem based on disabling interrupts, and rejected it because it doesn't work for systems with more than one CPU. For a system with a single CPU, however, this could be an acceptable solution, especially if the critical region is short. Write pseudocode for an implementation of semaphores for a single-CPU system that might not have a TSL instruction but does have library functions `enable_int()` and `disable_int()` to enable and disable interrupts respectively. (I.e., say what variables you would need for each semaphore, and give pseudocode for `up()` and `down()`.)

2. (5 points) The programming assignment for Homework 1 asked you to write a simple shell program using `fork()` to create a new process for each command executed by the shell. `fork()` essentially creates this new process by duplicating the process that calls it, including the state of any data structures related to open files. What advantages does this have? What are some possible disadvantages? Consider both situations in which the parent process waits for the child to finish (as in the shell program) and situations in which both processes continue concurrently. (*Hint:* Think about the standard input/output/error streams and also about other kinds of open files. Also try to apply what you know about buffering of input/output.)

3 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to `bmassing@cs.trinity.edu`, with each file as an attachment. Please use a subject line that mentions the course number and the assignment (e.g., "csci 3323 homework 2"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (10 points) The starting point for this problem is a simple implementation of the mutual exclusion problem in C with POSIX threads m-e-problem.c[1]. Each thread executes a loop similar to the one presented in class for this problem, except that:

[1] `http://www.cs.trinity.edu/~bmassing/Classes/CS3323_2012fall/Homeworks/HW02/Problems/m-e-problem.c`

- Rather than looping forever, each thread makes a finite number of trips through the loop.
- The critical region is represented by code to print some messages and sleep for a random interval.
- The non-critical region is represented by code to sleep for a random interval.

Currently no attempt is made to ensure that only one thread at a time is in its critical region, and if you run it you will see that in fact it frequently happens that all the threads are in their critical region at the same time. Your mission is to correct this.

Start by compiling the program, running it, and observing its behavior. To compile with `gcc`, you will need the extra flag `-pthread` and also `-std=c99`, e.g.,

```
gcc -Wall -std=c99 -pthread m-e-problem.c
```

(Or download this Makefile[2] and type `make m-e-problem`.) The program requires several command-line arguments, described in comments at the top of the code. (If you have trouble remembering the order, notice that the program prints a meant-to-be-helpful usage message if run with no arguments.)

You are to produce two corrected versions of this program:

- The first version should use shared variables only and one of the following algorithms:
  - Strict alternation, extended to work for an arbitrary number of threads. (No, this isn't a perfect solution, but it does enforce the "one at a time" condition.)
  - Peterson's algorithm, for two threads only. For extra credit, research and implement a variation that works for more than two threads. Cite a source for your solution if appropriate — e.g., "I found pseudocode for this solution at the following Web site." Or look up and implement Leslie Lamport's bakery algorithm.
- The second version should use one of the following sets of library functions:
  - The POSIX threads mutex functions. `man pthread_mutex_init` is a good starting point for finding out about these functions.
  - The POSIX threads semaphore functions. `man sem_init` is a good starting point for finding out about these functions.

Places in the program that should change are marked with "TODO" comments. You should not need to add much code. Confirm that your two improved versions behave as expected, i.e., when one thread starts its critical region no other thread can start *its* critical region until the first one finishes.

NOTE about shared variables: Optimizing compilers play a lot of tricks to reduce actual accesses to memory, as do most processors. What this means for multithreaded programs is that it is very difficult to guarantee that changes made to a shared variable in one thread are visible to other threads. Declaring shared variables `volatile` avoids at least some compile-time optimizations but does not provide any guarantees about what will happen at runtime, especially if there are multiple processors. For the latter, what is needed is a "memory fence", i.e., a way of specifying that at a particular point in the program all memory reads and writes have completed. As far as I know there is no portable way to achieve this in C99; one

[2] `http://www.cs.trinity.edu/~bmassing/Classes/CS3323_2012fall/Homeworks/HW02/Problems/Makefile`

must fall back on compiler- or processor-specific code. The starter code includes a function `memory_fence` that invokes a gcc-specific function providing a memory fence and recommends its use in the functions to begin and end the critical region. (*Disclaimer:* Apparently the version of this function present on our classroom/lab machines does nothing! This may be a bug in `gcc`. My sample solutions seem to work correctly anyway.) Note that some library functions for synchronization (e.g., the ones included with POSIX threads) incorporate this functionality as well.