CSCI 4320 (Principles of Operating Systems), Fall 2009

Homework 2

Credit: 50 points.

1 Reading

Be sure you have read Chapter 2.

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) If you were designing data structures for a process table and a thread table, say whether you would include the following in the process table, the threads table, or both, and briefly explain why.

   - A place to save CPU registers.
   - A place to save information about what memory is owned by the process or thread.

2. (5 points) When a computer is being designed, it is common to first simulate it using a program that runs one (simulated) instruction at a time. Even computers with more than one processor are simulated strictly sequentially like this. Is it possible for a race condition to occur when, as in this situation, there are no truly simultaneous events? Why or why not?

3. (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()`.)

4. (5 points) Restrooms are usually designated as men-only or women-only, but this requires having two restrooms if everyone is to be accommodated. A less expensive approach consistent with cultural norms in the U.S. would be to have one restroom with a sign on the door that indicates its current state — empty, in use by at least one woman, or in use by at least one man. If it is empty, either a man or a women may enter; if it is occupied, a person of the same sex may enter, but a person of the opposite sex must wait until it is empty. Write pseudocode for four functions to implement this approach: `woman_enter`, `man_enter`, `woman_leave`, and `man_leave`, to be used by the following pseudocode:

```
/* woman process */
while (TRUE) {
    woman_enter();
```

```
        use_restroom();
        woman_leave();
        do_other_stuff();
    }
    /* man process */
    while (TRUE) {
        man_enter();
        use_restroom();
        man_leave();
        do_other_stuff();
    }
```

You can use any of the synchronization mechanisms we have talked about (shared variables, semaphores, monitors, or even message passing). (If you'd rather write real code, do optional programming problem 2 instead.)

5. (5 points) Five batch jobs (call them A through E) arrive at a computer center at almost the same time. Their estimated running times (in minutes) and priorities are as follows, with 5 indicating the highest priority:

job	*running time*	*priority*
A	10	3
B	6	5
C	2	2
D	4	1
E	8	4

For each of the following scheduling algorithms, determine the turnaround time for each job and the average turnaround time. Assume that all jobs are completely CPU-bound (i.e., they do not block). (Before doing this by hand, decide how much of programming problem 3 you want to do.

- First-come, first-served (run them in alphabetic order by job name).
- Shortest job first.
- Round robin, using a time quantum of 1 minute.
- Round robin, using a time quantum of 2 minutes.
- Priority scheduling.

6. (5 points) Suppose that a scheduling algorithm favors processes that have used the least amount of processor time in the recent past. Why will this algorithm favor I/O-bound processes yet not permanently starve CPU-bound processes, even if there is always an I/O-bound process ready to run?

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 4320 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 Fedora 11 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:

   - 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`, e.g.

   ```
   gcc -o m-e-problem -pthread m-e-problem.c
   ```

   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 (declare them `volatile` so the compiler knows that it should access them in memory every time rather than keeping them in registers) 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.

[1] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2009fall/Homeworks/HW02/Problems/m-e-problem.c`

2. (Optional — up to 10 extra-credit points) Write a program to test your solution to problem 4. If you want to do this using C and POSIX threads, you could start with the code for programming problem 1. Or you could rewrite in Java and use either its monitor-based synchronization (synchronized methods/blocks plus `wait`, `notify`, and `notifyAll`) or features of the `java.util.concurrent` library package (which has, among many other things, a `Semaphore` library class). You can find some simple examples of multithreaded Java programs on the "Sample programs" page for my parallel programming class: http://www.cs.trinity.edu/ bmassing/CS3366/Sa The bounded buffer example may be useful if you want to use monitor-based synchronization.

3. (10 points) The starting point for this problem is a Java program that simulates execution of a scheduler, i.e., generates solutions to problem 5. Currently the program simulates only the FCFS algorithm. Your mission is to make it simulate one or more of the other algorithms mentioned in problem 5. You will get full credit for simulating one algorithm, extra points for simulating additional algorithms. The program consists of several classes, collected in a package called `scheduler`:

   - API documentation[3].
   - Code[4]. (Class `SchedulerTest` contains the `main` method of interest.)
   - Sample input[5].

   Feel free to rewrite anything about this program, including starting over in a language of your choice. Just remember that the program has to run on one of the department Linux machines, and it needs to accept input from command-line arguments and files — i.e., no GUIs, Web-based programs, etc. The latter requirement is to make it easier for me to automate testing your code. If you make changes to the format of the input — and I prefer that you don't — change the comments so they describe the changed requirements.

[2] `http://www.cs.trinity.edu/~bmassing/CS3366/SamplePrograms/`

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2009fall/Homeworks/HW02/Problems/scheduler/docs/`

[4] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2009fall/Homeworks/HW02/Problems/scheduler/source/scheduler`

[5] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2009fall/Homeworks/HW02/Problems/scheduler/sample.in`