CSCI 4320 (Principles of Operating Systems), Fall 2005

Homework 2

Assigned: October 11, 2005.

Due: October 18, 2005, at noon. *Not accepted late.*

Credit: 30 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) Consider a computer that does not have a test-and-set-lock (TSL) instruction, but does have an instruction to swap the contents of a register and a memory word in a single indivisible action. Use such an instruction (call it SWAP) to write a routine *enter_region* like the one found in Figure 2-22 in the textbook, or explain why this is impossible.

2. (5 points) Consider the procedure *put_forks* in Figure 2-33 in the textbook. Suppose that the variable *state[i]* was set to *THINKING* *after* the two calls to *test* rather than before. How would this change affect the solution? (I.e., would it work as well as before? better? not as well?)

3. (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 whether you want to do optional programming problem 2.)

   - 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.

4. (5 points) Recall that some proposed solutions to the mutual-exclusion problem (e.g., Peterson's algorithm) involve busy waiting. Do such solutions work if priority scheduling is being used and one of the processes involved has higher priority than the other(s)? Why or why not? How about if round-robin scheduling is being used? Why or why not?

5. (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?

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 Core 4 Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (5 points) The starting point for this problem is a simple C++/threads implementation threads-cr.cpp[1] of the mutual-exclusion problem. 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. There is probably more than one way to do this, but the easiest is to use the "mutex" library functions, which provide simple locking/unlocking. `man pthread_mutex_init` will tell you about these functions.

   Start by compiling the program and observing its behavior with different numbers of threads. To compile with `g++`, you will need the extra flag `-pthread`, e.g.

   ```
   g++ -o threads-cr -pthread threads-cr.cpp
   ```

   (The behavior of this program may depend on the release of the operating system and/or the number of processors. You might find it interesting to try it on a one-processor machine (one of the lab machines) and also on one of the multiprocessor machines (Dione01, Dione02, and SnowWhite). You may need to recompile recompile when switching to a machine running a different release of the operating system. Compiling the above code on SnowWhite generates some warnings; a version of the program that compiles without warnings there is old-threads-cr.cpp[2].)

   Then make your changes and confirm that the program now behaves 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_2005fall/Homeworks/HW02/Problems/threads-cr.cpp`

[2] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2005fall/Homeworks/HW02/Problems/old-threads-cr.cpp`

2. (Optional — up to 10 extra-credit points) The starting point for this problem is a program scheduler.cpp[3] that simulates execution of a scheduler, i.e., generates solutions to problem 3. 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 3. (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 standard input — i.e., no GUIs, Web-based programs, etc. The latter requirement is to make it easier for me to test your code, at least partially automatically. 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.)

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2005fall/Homeworks/HW02/Problems/scheduler.cpp`