CSCI 4320 (Principles of Operating Systems), Fall 2004

Homework 2

Assigned: October 12, 2004.

Due: October 19, 2004, at 5pm. *Not accepted late.*

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

1 Reading

Be sure you have read chapters 1, 2 and 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) Most Unix systems include some command that allows you to trace all system calls made by a process or command. Under Linux, this command is `strace`. For example, to trace all the system calls made during execution of the command `ls -l` and record the output in `OUT`, you would type

   ```
   strace -o OUT ls -l
   ```

   Your mission for this problem is to run `strace` for a command of your choice, capture the output, and then describe what some of it means. Specifically, I want you to pick at least four lines of the output using different system calls and briefly explain each of these lines, describing in general terms what the system call is supposed to do and what the parameters and return value mean. (So, you will turn in a printout of the output of `strace` with your homework. You might want to mark it up with numbers and then refer to these numbers in your explanation.)

   The `man` page for `strace` explains the general format of the output. To find out what the individual system calls do, you will need to read their `man` pages. Some of these are easy to find — e.g., the first call is usually to `execve`, and `man execve` will tell you about it. Some are a little harder to track down — e.g., `man open` produces information about an `open` command rather than a system call. `man -k open` produces a list of all `man` pages whose one-line descriptions include "open", and from this list one can perhaps guess that to look at the desired `man` page you need the command `man 2 open`. If the system call reported by `strace` ends in 64 (e.g., `fstat64`), the right `man` page can be found by removing "64" from the name (e.g., `man fstat`).

2. (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.

3. (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?)

4. (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 3.)

   - 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.

5. (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?

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?

7. (5 points) Suppose you are designing an electronic funds transfer system, in which there will be many identical processes that work as follows: Each process accepts as input an amount of money to transfer, the account to be credited, and the account to be debited. It then locks both accounts (one at a time), transfers the money, and releases the locks when done. Many of these processes could be running at the same time. A friend proposes a simple scheme for locking the accounts: First lock the account to be credited; then lock the account to be debited. Can this scheme lead to deadlock?

If you think it cannot, briefly explain why not. If you think it can, first give an example of a possible deadlock situation, and then design a scheme that avoids deadlocks, but in such a way that once an account is locked, it is not released until the funds transfer is complete (i.e., a design that relies on repeatedly locking one account, trying the other, and releasing the first is not allowed).

3 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Turn in your code by sending mail to `csci4320-homework@cs.trinity.edu`, with each of your code files as an attachment. If there's any question of which file(s) correspond to which problems, explain in the body of the mail message. Please use a subject line such as "homework 2" or "hw2". 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 2 Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (Optional — up to 10 extra-credit points) Figure 1-19 in chapter 1 of the textbook presents pseudocode for a simple command shell. Your mission for this problem is to turn this into a C or C++ program that runs on a Linux system. Your program should prompt the user for a command and command-line arguments (the prompt can be something simple, such as "?") and then run the given command with the given arguments. You can require that the user give the full path for the command, and you do not have to do sophisticated parsing of the command-line arguments (such as wildcard expansion, recognition of environment variables, etc., etc.). Here is a sample execution, terminated by control-C.

```
[bmassing@Athena]$ ./simple-shell
? ls
Unable to find command
? /bin/ls
Makefile  another  simple-shell  simple-shell.cpp  somefile
? /bin/ls -l
total 28
-rw-------    1 bmassing bmassing      119 Oct  3 08:02 Makefile
-rw-------    1 bmassing bmassing        5 Oct  3 08:00 another
-rwx------    1 bmassing bmassing    22035 Oct  3 08:15 simple-shell
-rw-------    1 bmassing bmassing     1407 Oct  3 08:15 simple-shell.cpp
-rw-------    1 bmassing bmassing        5 Oct  3 08:00 somefile
? /bin/ls junk
/bin/ls: junk: No such file or directory
?
[bmassing@Athena]$
```

   You can add more functionality (searching a path for the command, doing more sophisticated parsing of inputs, exiting when the user types "exit", etc.). If you do, describe the added functionality in comments at the top of your code. I will give 5 extra-credit points for the basic functionality described above, and up to 5 additional points for added functionality.

   Turning the pseudocode into code mostly involves defining appropriate data structures for the variables in the pseudocode and replacing the `type_prompt` and `read_command` functions with

appropriate real code. Your first step should probably be to read the `man` page for `execve` to see what arguments it expects, and then figure out what you need to do to turn what the user types in into suitable input to `execve`.

You will probably find that most of the code you write for this problem will be code to parse the input (accept a line of text and break it into a command and arguments). You can do this using C functions such as `scanf`, with the C++ `string` class, or whatever you prefer. If you use the C functions and fixed-size character arrays, try to make the program fail gracefully if the user supplies more input than your code has room to accept.

You may be tempted to just use the C library function `system`. Don't. You won't learn what this problem is meant to teach you, and you won't get credit for such a solution.

2. (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
   ```

   (Interestingly enough, the exact behavior of this program seems to depend both on the number of processors and on the release of the operating systems — try it on one of the lab machines and then on one of the Dwarf machines to see what I mean. You may need to recompile recompile when switching to a machine running a different release of the operating system. Compiling the above code on a Dwarf generates some warnings; a version of the program that compiles 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.

3. (Optional — up to 5 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 4. (This is an updated version that corrects a bug. The original version is available as scheduler-v0.cpp[4].) 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 4. (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

[1] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2004fall/Homeworks/HW02/Problems/threads-cr.cpp`

[2] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2004fall/Homeworks/HW02/Problems/old-threads-cr.cpp`

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2004fall/Homeworks/HW02/Problems/scheduler.cpp`

[4] `http://www.cs.trinity.edu/~bmassing/Classes/CS4320_2004fall/Homeworks/HW02/Problems/scheduler-v0.cpp`

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, change the comments so they describe the changed requirements.)