CSCI 4320 (Principles of Operating Systems), Fall 2009

Homework 1

Credit: 20 points.

1 Reading

Be sure you have read Chapter 1.

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) For each of the following instructions, say whether it should be executed only in kernel (i.e., supervisor) mode and briefly explain why.

   (*Hint*: In general, user programs should not be allowed to execute instructions that might interfere with the operating system's control of the machine. The most reasonable way to keep them from doing so is to allow such instructions only in supervisor mode. Notice that this question refers to machine-level *instructions*, not necessarily functionality. An operating system could make the functionality of some of these instructions available to user programs by wrapping them in system calls, and possibly requiring user programs to supply a password to (successfully) execute these calls.)

   (a) Disable all interrupts.

   (b) Read the time-of-day clock.

   (c) Set the time-of-day clock.

   (d) Change whatever registers are used to determine which part of memory the current process has access to.

   (e) Switch from user mode to supervisor mode.

2. (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 (part 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`.

As an example of what I have in mind, here is a line from a trace of the command `ls /users/cs4320` with commentary. (You should choose system calls other than `execve`.)

```
execve("/bin/ls", ["ls", "/users/cs4320"], [/* 71 vars */]) = 0
```

The call to `execve` creates a new process to run the command. Parameters are the command to execute, the arguments to pass to it, and an array of environment variables (71 of them, apparently!). The return value of 0 probably doesn't mean anything, since the `man` page for `execve` says that the function doesn't return if the call is successful.

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 1"). 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) Figure 1-19 in chapter 1 of the textbook (p. 54) 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@xena02]$ ./simple-shell-c
? /bin/ls
simple-shell-c simple-shell-c.c  simple-shell-cpp  simple-shell-cpp.cpp
? /bin/ls -l
total 36
-rwx------ 1 bmassing bmassing  6920 Sep 14 07:51 simple-shell-c
-rw------- 1 bmassing bmassing  5175 Sep 14 07:51 simple-shell-c.c
-rwx------ 1 bmassing bmassing 15440 Sep 14 07:52 simple-shell-cpp
-rw------- 1 bmassing bmassing  3984 Sep 14 07:51 simple-shell-cpp.cpp
```

```
? /bin/ls junk
/bin/ls: cannot access junk: No such file or directory
? /usr/local/bin/p simple
bmassing  2859  0.0  0.0   1856   412 pts/3    S+   07:55   0:00 ./simple-shell-c
?
```

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 up to 5 extra-credit 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` — carefully — 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, make the program fail as gracefully as possible 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.