CSCI 3323 (Principles of Operating Systems), Fall 2015

Homework 1

Credit: 35 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 one of my mailboxes (outside my office or in the ASO).

1. (10 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) Change whatever registers are used to determine which part of memory the current process has access to.
   (d) Set the time-of-day clock.
   (e) Switch from user mode to supervisor mode.

2. (10 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 write` produces information about a `write` command rather than a system call — but `man -a` (e.g., `man -a write`) will show you all `man` pages for functions and commands with a given name, one of which should be the one you need.

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", "-l", "/users/cs4320"], [/* 78 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 (78 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 3323 homework 1"). You can develop your programs on any system that provides the needed functionality (which for this assignment means something UNIX-like), 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. (15 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 repeatedly prompt the user for a command and command-line arguments and then run the given command with the given arguments. (Do *not* start writing code until you read the whole assignment. I am providing starter code that takes care of the parts that are tedious to program in C.) You can require that the user give the full path for the command (this is easier to implement and reasonable in context), and you do not have to do sophisticated parsing of the command-line arguments (such as wildcard expansion, recognition of environment variables, etc., etc.). The program, however, should do something sensible (such as displaying an error message) if it cannot run the command, and it should stop on reaching EOF on standard input so that it can accept commands from either a file or the keyboard (where pressing control-D signals EOF). Here is a sample execution:

```
$ ./shellsketch
next command?
/bin/ls
Makefile     shellsketch-starter.c  test-input.txt
shellsketch  shellsketch.c          typescript
```

```
next command?
/bin/echo ab cd ef gh
ab cd ef gh

next command?
junk
cannot execute command: No such file or directory

next command?
/bin/ls junk
/bin/ls: cannot access junk: No such file or directory

next command?
```

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. You may recall that anything dealing with text strings is apt to be tedious and messy in C, so here is code that takes care of most of that for you, including some debug prints to track what it is doing.

simple-shell.c[1].

Assuming you compile with `gcc`, you will need the `-std=c99` flag.

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 starter code produces (an array of pointers to strings) into suitable input to `execve`. (You should not need to do much.) Recall (or note) that man pages for functions tell you what if any `#include` directives you need to include in your code.

For extra credit (up to 5 points), 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, add something to the comments in the code describing your added functionality. If you insist, you can even rewrite any or all of the starter code in C++. Whatever changes you make, however, be sure your program will still work with input that is valid for the starter code.

C tip: Get in the habit of compiling with the `-Wall` flag and paying attention to warning messages. Sometimes warning messages really are just warnings you can ignore, but often they are signs of problems you should fix. Code that produces warnings with compiled with `-std=c99 -Wall -pedantic` will lose points.

[1]`http://www.cs.trinity.edu/~bmassing/Classes/CS3323_2015fall/Homeworks/HW01/Problems/simple-shell.c`