COMP 530 INTRODUCTION TO OPERATING SYSTEMS

Fall 2017
Kevin Jeffay

Homework 2, September 6

Due: 11:15 AM, September 20

Building a Simple Shell

The purpose of this assignment is to do something substantive with Linux processes. For this assignment you will develop a simple Linux shell (*i.e.*, a command interpreter). Although this is only a toy shell, its basic structure and operation will be not that far removed from that of a real shell.

Your simple shell program will:

- Output the string "% " as your shell's prompt to *stdout*.[1]
- Read a line of input from *stdin* (a line here is a series of characters terminated by a newline).
- Fork a new process.
- The parent process will simply wait for the termination of the child process.
- The child process will parse the line of input (the "command line") into a set of words ("arguments"). Words are defined as sequences of printable characters separated by whitespace (spaces, tabs, *etc*.).
- The child process will interpret the first word in the set (the first argument) as a filename and will *exec* the file, passing the filename and the remaining arguments (if any) to *exec* as the "*argv*" parameter (the "arguments" parameter) for the program being *exec*'ed.
- After the child terminates, the parent (your shell program) continues to read and process input lines as described above from *stdin* until end-of-file is reached. Once end-of-file is reached the program terminates.
- It is OK for your program to have some non-trivializing limitation, such as a limit on the number of characters that can appear on the command line. However, in all cases it should not be possible for a user of your program to ever make your program crash or hang. Thus, if your program has some limitation, you must detect when the limitation is reached and take an appropriate action (*e.g.*, output a meaningful error message to *stderror* [standard error]).
- Along the lines of the previous point, any error conditions generated by *fork* or *exec* (or any other system calls you make) should be processed by your program and should result in the generation and output of an appropriate error message.

Details

Write a C program on Linux that performs the above steps in a loop. To *exec* the file whose name was read from the command line, use the system call *execvp*. You should carefully read the on-line manual ("man") pages on

[1] Note that the prompt is the two-character sequence of the percent sign followed by a space. If you don't use *exactly* this character sequence as your prompt your program's output won't match our test output and you will likely fail all our tests and you'll end up with a score of "0" for this assignment. Thus, it's *really important* that you output *exactly* the right prompt!

Linux for *execvp* before getting started. Note the differences in handling filenames that do and do not contain a "/" character.

Dealing With Zombies!

Whenever you fork a process (*i.e.*, create a new process), the forked (child) process runs in parallel with the forking (parent) process. If these processes are not synchronized properly, or do not terminate properly, you run the risk of creating a "zombie" process. A zombie process is a process that does not execute (is "dead") but does not terminate and go away ("die"). As a result zombie processes continue to consume resources within the operating system such as process descriptors. If zombie processes accumulate, it is possible to slow down, hang, or even crash the operating system.

Since this assignment will be the first time many of you have created processes, there is an excellent chance you will create one or more zombie processes because of bugs in your program. If this happens, the server *classroom.cs.unc.edu* can become sluggish and/or hang.

That you will have bugs related to the use of fork is to be expected. However, to mitigate the effect of these bugs on the performance of the server, you need to take steps to limit the number of processes you can create. Every time you log into the servers *classroom* or *snapper* please execute the following commands from the command line:

```
limit
limit maxproc 10
limit
```

The first command will show the limits of various resources you can consume. The second command limits the number of processes you can create to 10. The third command will again show your limits and allow you to confirm that you've correctly limited the number of processes you can create.

It is essential that you execute these commands every time you use the system. For this reason, the best thing to do is to edit the file ".cshrc" (with a leading period) in your home directory, and add the line "limit maxproc 10" to the file as the last line in the file. This will always set the process limit and then you don't manually have to do it every time you log in.

If you limit the maximum number of processes, then, if you have a bug in your program and are creating zombie processes, you'll eventually get an error message when you try to run your program (the message indicating the maximum number of processes has been exceeded). This error message will be the only indication you get that you have a bug in your program. Should this happen, you won't be able to continue testing your program until you kill off your zombie processes. To kill zombie processes, first, use the "ps" command to see the identities of the processes you've created:

```
ps -ef | egrep -e PID -e YOUR-LOGIN-NAME
```

where you replace YOUR-LOGIN-NAME with your Linux login name. For example, if this were my login session I'd type:

```
ps -ef | egrep -e PID -e jeffay
```

You can then use the "kill" command to kill any found zombie processes by using the process number (the PID) which is shown under the second column of output by the ps command. To kill a process use the command:

```
kill PID
```

where PID is the umber you get from executing the ps command.

Generally, until you are certain your program is working, execute the ps command prior to logging out so you can see if you are leaving behind any zombie processes and kill them before you log out.

Extra Credit[2]

For extra credit, have your child process test to see whether or not the file name to passed to *exec* is a valid filename before calling *exec*. (Read the man 2 page for the system call "*stat*.")

A more challenging extension is to use the *execv* system call instead of *execvp*. The primary difference between these two versions of *exec* is that with *execv* you will have to read the shell environment variable PATH and search the directories in the PATH. When using *execv*, use *stat* to determine the appropriate directory where the command is stored, and construct a complete file pathname to use with execv. (Read the man page for "*getenv*.")

For an even more challenging extension, you can investigate the use of *signals* in Linux and use signals to allow the child process to be either "interrupted" or "stopped" without adversely affecting the parent. That is, the user should be able to type cntrl-C (generate a "*SIGINT*") or cntrl-Z (generate a "*SIGTSTP*") while the child process is executing and have control gracefully returned to the parent process. (Read the man 2 and 7 pages for "*signal*.")

Grading

Place copies of your C program(s) in your *~/comp530/submissions/HW2* directory. Name your program 530shell.c. Send mail to *jacobdf@cs.unc.edu* when your program is ready for grading.

Functional correctness will be based on the program's ability to parse and execute arbitrary commands, deal with errors (*e.g.*, "file not found"), and deal with arbitrary behavior of the child process (premature termination, faulting, *etc*.).

Important reminders:

- When you send your submission email, be sure to include your Linux login name in the email.
- Remember that Linux file names are case sensitive! Make sure you name your file exactly as specified herein.
- As always, after you submit your homework, *do not modify your files*. If you want to keep working on the assignment, do so on a copy.

[2] As a general matter, you should NOT attempt any extra credit portion of an assignment before you've *fully* completed the "regular" assignment.