O.S. Lab 1: A Simple Linux Shell

We will now consider some concepts related to processes running under Linux. We start with the *shell*, that part of the operating system with which a computer user actually interacts. It provides an interface for the user to run programs and to manage files. Early shells were nearly all command line-based; at the prompt, the user types in commands or names of programs to be executed. In this project, you will write a simple command-line shell that will run on a Linux computer. Your shell should allow the user to start programs and perform several simple file manipulation tasks.

You must write your shell in the standard C language. To complete this project you will need to use both standard C functions and POSIX system calls. You should not make use of any existing shell; in other words, you should not do any form of scripting. This project requires that you have access to a Linux account. While it is not required to complete this particular project on the computers in CAS 241, these computers will be used to grade your shell. Whether you work in the lab or remotely on `knuth`, to keep from blowing the thing up, you should limit the number of threads you start by issuing the `ulimit -u` command before you execute your shell. (For details check the `man` page for `bash`.) Create a make-file that does this before starting your shell.

Shell Components

Your basic shell should prompt the user to enter a command, read in a text string from the user, parse the text string to determine a command, execute the command, and repeat these steps forever. Your shell should be able to execute a variety of commands. Your shell should also be able to recognize incorrect commands and give the appropriate error message.

Your shell should also accomplish the following:

- It should accept one command per line. It can execute any valid command typed in with arguments using the **execvp()** (and not **system**) system call. The separation of multiple commands on a single line by semicolons will not be implemented.
- If the user presses the [Enter] key without a command, the shell displays another prompt.
- The default prompt for this assignment will be of the form

  linux (*your uanet id***)|>**

A good initial version of this program may be found online. Note that this starter file is not yet a working program. In addition to refining this file, you now must add multiple things described in this handout.

Shell Commands

Your shell must support the following internal commands:

- C *file1 file2* — Copy; create *file2*, copy all bytes of *file1* to *file2* without deleting *file1*.
- D *file* — Delete the named file.
- E *comment* — Echo; display *comment* on screen followed by a new line (multiple spaces/tabs may be reduced to a single space); if no argument simply issue a new prompt.
- H — Help; display the user manual, described below.
- L — List the contents of the current directory; see below.
- M *file* — Make; create the named text file by launching a text editor.
- P *file* — Print; display the contents of the named file on screen.
- Q — Quit the shell.
- S — Surf the web by launching a browser as a background process.
- W — Wipe; clear the screen.
- X *program* — Execute the named program.

All commands are case sensitive. Any command not part of this list should just be passed to **execvp()** and normal execution attempted.

Program Execution

The **execvp** system call is used to load a program into memory and execute it. It has the syntax **execvp(***filename***,** *args***)** where

- *filename* is a character array (string) consisting of the name of the program to be executed, and
- *args* is an array of strings consisting of the command line arguments to be passed to the program.

If you are not passing any command line arguments, you should define an empty array of strings and pass that as *args*. This may be done easily:

```
char args[1][1];
args[0][0] = '\0';
```

One challenge you will face is that if you use **execvp** to run the program, when the program finishes it will not go back to your shell; instead your shell will terminate. To get around this, you should create a new process, use that process to execute the program, and keep your shell in the background to take over when the program finishes. This may be done with the **fork** system call, which has the syntax

```
int pid = fork()
```

Once you call **fork**, two processes will execute the subsequent code. One process will have the number 0 in pid, the other will have a different number in `pid`. The process with the number 0 in `pid` should call **execvp**, the other should return to the prompt.

Once you get towards the end, you will notice that the above method of executing a program has a problem: your shell will prompt again before the program you called finishes running. You should improve your shell so that the process that does not call **execvp** waits until the program has finished before it prompts again (with one exception; see below). This can be done with the **wait** system call, which suspends a process until another process terminates. The syntax of the call looks like this:

```
int status;
wait(&status);
```

Shell Implementation

Most of the needed commands have a Linux analog. Simply substitute the correct Linux command for your shell instruction and **execvp()**:

- C *file1 file2* — **cp** *file1 file2*
- D *file* — **rm** *file*
- M *file* — **nano** *file*; terminate with [Ctrl][X]
- P *file* — **more** *file*
- S — **firefox**
- W — **clear**
- X *program* — Get rid of "X" and simply use the name of the program

For "L" first skip a line, give the current directory (**pwd**), skip another line, then list the directory contents in long form (**ls –l**). "E" and "Q" should be straight-forward. If the user presses the [Enter] key without a command, the shell should display another prompt.

For "S" run **firefox** in the *background*. This means that the shell does not wait for the child thread to finish executing before issuing another prompt and accepting another command. In a standard Linux shell this is done via the "**&**" operator, i.e. **firefox &**. For your homemade shell, simply edit the code so the shell doesn't wait fot the "S" command to finish.

Notes on Help

When the user types "H" a simple manual describing how to use the o.s. and shell should be displayed on screen. The manual should contain enough detail for a Linux beginner to use it. For an example of the sort of depth and type of description required, execute **man csh** or **man tcsh**. These shells have much more functionality than yours, so your manuals don't have to be quite so large. Keep in mind that this is an operator's manual and not a developer's manual.

Notes on Plagiarism

This shell project is a fairly common operating system project, and I am aware of C code available on the Internet that duplicates much of what I am asking you to do. You will almost certainly need to make use of books and websites in order to get the POSIX calls working. However, *please do not copy any online code (other than the provided seed file and text book excerpts) into your project*. If you make any extensive use of a book or website, please cite it in your submission report.

Submission Requirements

When finished, submit a single .zip or .tar file (no .rar please) with your name as the filename containing the following deliverables to me:

- Your source code, well commented;
- Your executable file; and
- A README file consisting of the following:
    - a step-by-step description of how to compile your source code and run your shell,
    - an explanation of how to use your shell,
    - a list of what functions your shell contains, and
    - citations of any resources you used in completing your project.

Last updated 9.10.2020 by T. O'Neil. Some parts based on original material by M. Black. Previous revisions 8.5.2019, 3.26.2015, 9.17.2014.