CS 470 - Operating Systems
Spring 2019 - Shell Project
40 points

Out: January 25, 2019
Due: February 11, 2019 (Monday)

The purpose of this project is provide experience with process manipulation in UNIX.

Problem Statement

Complete the UNIX shell and history feature project described on pages 157-160 in the textbook. (Note the last paragraph of the project is on page 160.) As explained in the textbook, this project is organized into two parts:

1. Create a child process that executes the given command that the parent may or may not wait for using the `fork()`, `exec()`, and `wait()` system calls.

2. Implement a history mechanism

Additional Design Notes

- A user command of "`exit`" should cause the shell program to return (terminate). I.e., don't try to exec it.
- In current versions of Linux, the `cd` command is built into the bash shell and does not have a separate program executable. As a result, it cannot be executed using the exec system call. Therefore, you should not use it to test your shell, and you do not need to provide that functionality. This project is just a simple shell that can be viewed as an application launcher.
- The third step in Figure 3.36 should be "`(3) if command` **`does not`** `include &, parent will invoke wait()`". I.e., the parent waits when the command is not run in the background.
- Note that the history is listed with the most recent command first. (Opposite of the what the Linux history command does.) Also, the history data structure should only store 10 commands. (I. e., you cannot store all commands and only print out the last 10 for the history command.)
- Note that the `args` array passed to `execvp` is an array of pointers to char array (C string). Don't forget the actual char arrays (C strings) that the elements of `args` point to must be allocated (either statically or dynamically). Also be sure to deallocate explicitly any space that has been allocated dynamically explicitly.
- An `exec` call only returns if there was an error trying to start the new program, e.g. if the command does not exist. However, if an `exec` call fails, its process must be terminated, otherwise it will continue to run the parent (shell) code. In particular, it will print (extra) prompts to the screen.
- Background processes still need to be waited upon in order for them to terminate properly. Otherwise, when they try to terminate, they will become zombies for the duration of the new shell execution. The best way to handle this is to take advantage the fact that when a process is orphaned (i.e., its parent has terminated), it is adopted by the init process that automatically issues a wait for it. By having the child process do a second fork, and then the grandchild process do the `exec` call while the child process terminates immediately, the grandchild process is adopted by the init process so when it terminates, there is a wait call for it. This is called the *double-fork technique* and often is used for long-lived server-type processes (e.g., web server) that fork a separate process to handle an individual client request. **Using this technique also means that the parent process always waits for the child, even for background processes.**

Assignment

(20 points) Implementation of this project. This project is to be done individually. While there are no explicit design guidelines for this course, programs should exhibit a modular or object-oriented design. Projects exhibiting particularly poor design will not earn full credit. **This project may be written in any language as long as the language supports process creation and execution, and runs on csserver.** Please note that the instructor will only be able to provide assistance for projects written in C/C++. Provide a makefile that will make your project if it needs to be compiled.

(10 points) Provide a high-level discussion of the program design describing the functionality of the major components of the program and how they interact with each other, and a more detailed discussion for the data structures and algorithms used in the history feature portion of the program. If the program does not meet all of the project requirements, describe which parts are missing and what you think should be done to implement them.

(10 points) Provide a test plan for your project. I.e., give a list of commands that will demonstrate the features of your shell. Annotate this list with notes regarding what result is expected. The grade for this portion of the project will depend on how thorough the test plan is. Note that the test plan should cover all of the project requirements whether or not the program actually implements them.

In addition, answer the following questions:

1. What aspect of process manipulation did you find most difficult to understand?
2. What aspect of process manipulation did you find least difficult to understand?
3. What, if anything, would you change in your current design?
4. What, if anything, did you find interesting or surprising about process manipulation that you did not know before doing this project?

What to Submit

Create a **tarfile** or **zipfile** archive containing the following items:

- The well-documented source code for your shell.
- A makefile to make your project, if needed
- A **single** document, **in PDF format**, with the discussion of the functional design of your project, the test plan for your project, and the answers to the questions above. **If the project is not written in C/C++, put instructions on how to build and/or run the program at the beginning of this file.**

Submit your archive using the submission system (http://submission.evansville.edu) no later than 11:59pm on the due date. Reminder: your username is your ACENET username with "-cs470" appended to it, and your password is your student ID number including the leading 0 (i.e. 7 digits) unless you have changed it. The grading script only will accept submissions. It will not run anything.