Project 1: A Simplified Unix Shell Program

Due: 10/18/2018

Project Statement: Develop a simplified version of Unix Shell program

Project Objectives: Practicing Unix system calls, understanding Unix process management, synchronization, and inter-process communication (IPC)

Project Description

In this project, you will develop a simplified version of Unix Shell or a command line interpreter such as sh or bash. When your shell starts, it should print a dollar sign ($) and a space to the standard output terminal, and then wait for input from the user. You can assume each input line from the user will have no more than 80 characters. At the minimum, your shell should meet the following requirements. Name your shell as myshell.

- Your shell should support four built-in commands "exit", "cd", and "pwd", and "set".
  - exit: terminate the shell
  - cd: change the current working directory. Syntax: cd dir. dir is the directory where the user wants to change to. If dir starts with a slash (/"), dir is an absolute path. Otherwise, dir is a path related to the current directory.
  - pwd: print the absolute path of the current working directory
  - set: set the value of the environment variables. You only need to support one environment value MYPATH, which is the search path for the external command you implement (see below). Syntax: set MYPATH=path1:path2...
- Your shell should also terminate if end of file character (CTRL-D) is encountered.
- In your project, you should also implement one external command (program): "myls"
  - myls: list files under a directory. The output format should be similar to "ls -l".
  - For this command, you cannot simply call exec and pass in ls to implement it. Instead, you must implement this command via file and directory access functions. In order for your shell to locate this command, it will search the MYPATH variable. That is, in order to run the command myls, one needs to put myls under certain directory and set up MYPATH properly.
  - Note also that, as an external command, you need to write a separate C/C++ program for this command (i.e., this command is not part of the shell you are implementing). If you are unfamiliar with the concept of internal (built-in) commands and external commands of shells, you may want to study a bit more on this topic.
- Your shell should support all existing external commands in the system (i.e., executable programs that can be found in one of the search path contained in environment variable PATH). For these commands, you can call "exec" functions in your shell implementation.
- Your shell should support pipes, e.g, command1 | command2 | command3 | command4. If it is needed, you can assume a maximum number of pipes in any command line. However, it should be greater than 1. Note that commands may have parameters.
- Your shell should support I/O redirections, e.g. command1 < file1, or command1 > file1, or command1 < file1 > file2.
- Your shell should support a simple form of background processes, that is, commands followed by an ampersand (&). We do not require a full support of background processes, but you are more than welcome to do so (see below).
- You can assume that pipe and I/O redirection will not appear together in any command line. You can

also assume that pipe and background process will not appear together in any command line.
- Your shell does not need to support any other special characters (in particular, special character expansion such as *).
- You cannot use the "system()" function (system call).

Notes on Project Requirements

- For "myls", the output should be as close to the "ls -l" as possible. There are a number of items that are harder to obtain than others. You are not required to output them as same as the Unix command.

- For the background process, you can decide to what degree you want to support background process. You can support background process as same as the Bourne shell (which does not support job control), or as same as bash (which supports job control). However, as a minimum requirement, your implementation of background process must let your shell proceed to print the next prompt so that users can input the next command, without waiting for the background process(es) to finish. A particular problem you need to pay attention to is that, whatever user types after the next shell prompt is printed is the input to the shell, not to the background process (if it also reads from standard input). Depending on if your shell really support job control, the way to handle this situation is different. You need to do a bit research on this.

- You cannot create any (long-term) zombie processes. For example, you cannot keep a terminated child process in the process table until your shell terminates. Points will be deducted if you have any (long-term) zombie processes running. That is, your shell must handle properly when a child process terminates.

Grading Policy

A program with compiling errors will get 0 point. A program containing "system()" function will get 0 point. Total points: 130.

1. proper README file (5)
2. proper makefile file (5)
3. built-in commands "exit", "cd", "pwd", "set". Terminating shell when end-of-file is encountered (15)
4. running single command (with and without parameters) on a command line (20)
5. running multiple piped commands ("|") (20)
6. allowing background execution (command line with &) (20)
7. allowing I/O redirection in a command ("<" and ">") (20)
8. your own external command (program) "myls" (20)
9. code readability (5)

Project Submission

Project must be submitted online on Canvas. Tar your readme file, makefile, all source code files into a single tar file. (You should have at least two C/C++ source code files, for example, myshell.c and myls.c.) Make sure that you tar files successfully (you can check the content of the tar file by running "tar -tvf tar_file"). You are responsible for empty tar file or wrong tar file submitted, and late penalty will apply if you need to re-submit after the deadline.

The readme file should at least contain the following information: functions that your shell supports, limitations that your shell may have. Any particular way to run your program or commands in your shell.

Hints

Make sure your shell can terminate properly when end-of-file is encountered. We may test your shell by passing (i.e., redirect) to it a file containing a sequence of command lines.

Read the manual page of your favorite shell such as bash, sh, or csh to have a better understanding on how a shell works, in particular, built-in commands and external commands. Also, observe how they support pipe and background process by running some commands.