CprE 308 Project 1: UNIX Shell

Department of Electrical and Computer Engineering
Iowa State University

Fall 2012

Due Monday, September 24th at 11:59pm

Submission

Include the following:

- A cohesive summary of what you learned in the project. This should be no more than two paragraphs.
- A complete listing of your source code, commented and formatted neatly. Use good programming practices (i.e. create functions to abstract large bodies of code and code common to multiple methods, only use global variables when you must, and check syscall or system dependent libcall return values for errors - don't assume the OS will always succeed in fulfilling your requests).
- A Makefile to compile your code. The grader should be able to compile your project by issuing the `make` command.
- The assignment should be submitted through Blackboard before the deadline. Your submission should be a single `tar` archive containing your source code, your report, and your Makefile. Your archive should be named *shell*, combined with your name and lab section (e.g. `shell-lastname-A.tar.gz`). To create a `tar` archive, simply run the command "`tar -czvf shell-lastname-A.tar.gz folder/`" where `folder/` is the folder you wish to archive.

Late Policy

You will lose 10% per day for five days; after that you will no longer be able to receive credit. For example, submissions received between 12am to 11:59pm on the 25th will be deducted 10%.

Introduction

In this project you will create your own version of a UNIX shell. It will perform a similar function to *bash*, the shell you have been using in the lab, but will not be as sophisticated. A UNIX shell is an interactive interface between the OS and the user. It repeatedly accepts input from the user and initiates processes on the OS based on that input.

Requirements

- Your shell should accept a `"-p <prompt>"` option on the command line when it is started. The `<prompt>` option should become the user prompt. If the option is not specified, the default prompt of '308sh>' should be used. Read Appendix A for more information on command-line options.
- Your shell must run in a loop, accepting input from the user and running commands. The loop should exit only when the user requests to exit.
- Each user input should be treated as a command to be executed. However, there will be a few special cases that the shell will treat differently (see below).
- For each command, the shell should spawn a child process to run the command.

- The shell should try to run the command exactly as typed. This will require the user to know the full path (absolute, i.e. `/some/dir/some/exec`, or relative, i.e. `../../some/exec`) to the executable, which can be obtained using the command-line program `which`. HINT: `execvp` can search for commands in the directories listed in the PATH environment variable. You can see this variable by executing "`echo $PATH`" from the command-line.

- The shell should notify the user if the requested command is not found and cannot be run. HINT: read about the libc call, `perror`.

- For each spawned child, print the process ID (PID) before executing the command. It should only be printed once, and should be printed *before* the output from the command.

- By default, the shell should block (i.e. wait for the child to exit) for each command, except when the last character in the user input is an ampersand (see below). Thus, by default, the prompt will not be available for additional user input until the command has completed. Your shell should only wake up when the most recently executed child process completes (see `waitpid`).

- If the last character in the user input was an ampersand (&), the child should be run in the background (i.e. the shell will not wait for the child to exit before prompting the user for further input). Make sure you remove the & before passing the parameters to exec. When your background process does exit, you must print its status like you do for foreground processes.

  Example: if the user enters "`/usr/bin/emacs`" then the shell would wait for the process to complete before returning to the prompt. However, if the user enters "`/usr/bin/emacs &`" then the prompt would return as soon as the process was initiated. HINT: to evaluate and print the exit status of a background child process, call `waitpid`

with -1 for the pid and `WNOHANG` set in the options. This checks all the children and doesn't block - see the man page. To do this periodically, a check can be done every time the user enters a command.

- The shell should evaluate the exit status of the child process and print the conditions under which it exited (identify which process exited by its PID). Read the man page for wait to see a list of macros that provide this information (man 2 wait). Two examples of exit status are normal exit and killed (signaled).
- The following commands are special commands (also called built-in commands). See Appendix B for some useful functions to help implement these features.

  cd
  the shell should change the current working directory. For example, "`cd /usr/bin`" changes the directory to `/usr/bin`

  cwd
  the shell should print the current working directory

  exit
  the shell should terminate and accept no further input from user

  history -x
  the shell should print out a list of the last x commands entered by the user. If there are less commands in the history than requested by the user, simply print all commands (see Appendix C for an example)

  pid
  the shell should print its own process ID

  ppid
  the shell should print the process ID of its parent

Other Comments

- Note that all command-line parameters (e.g. the command `ls` has two command-line parameters in "`ls -al` *`home/user`*") *need to be passed on to the executed command via exec. These parameters become /argv* to the new process. See Appendix A for more information on command line parameters.
- You may want to prefix all your output messages with some sort of identifier, such as ">>>", to more easily determine which lines are yours, and which came from the executed program.

Grading Criteria

Summary

15% Well-written and specific.

Submitted Source Code

- Compiles - 10%
- Can be read and understood easily - 10%
- Performs the required functionality - 65%

Appendix A - Command-line Options

Command-line options are options passed to a program when it is initiated. For example, to get `ls` to display the size of files as well as their names the `-l` option is passed to it. (e.g. "`ls -l /home/me`", where `/home/me` is also a command-line option which specifies the desired directory for which contents should be listed).

From within a C program, command-line options are handled using the parameters *argc* and *argv*, which are passed into main like this: "`int main(int argc, char **argv)`"

The parameter *argc* is a value that contains the number of command-line arguments. This value is always at least 1, as the name of the executable is always the first parameter.

The parameter *argv* is an array of string values that contain the command-line options. The first option, mentioned above, is the name of the executed program. The program below simply prints each command-line option passed on its own line.

```
int main(int argc, char **argv)
{
    int i;
    for(i = 0; i < argc; i++)
        printf("Option %d is :%s\n", i, argv[i]);
    return 0;
}
```

If you are unfamiliar how command-line options work, enter the above program and try it with different values. (e.g. "`./myprog I guess these are options`")

Appendix B - Useful System and Library Calls

All of the following C functions can be found in sections 2 and 3 of the Linux man page. Example: type "`man 2 fork`" to look up `fork` in the man pages.

`fork`
create a child process

exec calls, such as `execvp`
replace the current process with that of the specified program

`waitpid`
wait for a child to exit (or get exit status)

`exit`
force the current program to exit

chdir
change directories

getcwd
get the current working directory

getenv, setenv
retrieve and set environment variables

perror
display error messages based on the value of *errno*

strcmp, strcpy, strcat
manipulate strings

Appendix C - Example input and output

Below is an example of how your program should respond to a series of inputs.

```
308sh> cwd
/home/amungons

308sh> cd /etc/java

308sh> ls -la
pid: 2043
total 68
drwxr-xr-x   3 root root  4096 Oct  5  2009 .
drwxr-xr-x 116 root root 12288 Sep  7 17:14 ..
-rw-r--r--   1 root root 13537 Dec 15  2004 font.properties
-rw-r--r--   1 root root   493 Jan 11  2007 java.conf
-rw-r--r--   1 root root    51 Jan 11  2007 jpackage-release
drwxr-xr-x   3 root root  4096 Oct  5  2009 security

308sh> history -10
cwd
cd /usr/bin
ls -la

308sh>
```