CprE 308 Project 1: UNIX Shell

Department of Electrical and Computer Engineering
Iowa State University

This is a two-week programming project assignment. Attendance is only counted the first week. You are encouraged to attend the second week as well if you have not finished the project, or if you have any questions.

1 Submission

Submit the following items to Canvas:

- 15 pts — A .doc, .docx, or .pdf file containing a summary of what you learned in this project. It should be no more than two paragraphs.
- 85 pts — A .zip or .tar file containing all of the source code files and a makefile for your program. It will be graded by the following criteria:
  - 10 pts — program compiles by executing the make command
  - 10 pts — source code is neatly formatted and contains comments that describe the important parts of the program (i.e., *briefly* walk us through your code so that we know you know what you did—bearing in mind that your code should be written cleanly enough that we don't *need* comments to understand it!)
  - 65 pts — required functionality works

2 Project Description

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

2.1 Requirements

1. 5 pts — Your shell should accept a `-p <prompt>` option on the command line when it is initiated. The `<prompt>` option should become the user prompt string. If this option is not specified, the default prompt of "`308sh>` " should be used. Read section 3.1 on command line options for more information.
2. 5 pts — Your shell should run as an infinite loop accepting input from the user and running commands until the user requests to exit.
3. Your shell should support two types of user commands: (1) built-in commands, and (2) program commands.

(a) 15 pts — Built-in commands are commands to interact with your shell program, such as `cd`, `cd <dir>`, etc. See the list of required built-in commands in section 2.2.

(b) 15 pts — Program commands require your shell to spawn a child process to execute the user input (exactly as typed) using the execvp() call. This means a user command will be entered either using an absolute path, a path relative to the current working directory, or a path relative to a directory listed in the PATH environment variable. (Read *man execvp*, especially the "Special semantics for execlp() and execvp()" section. Note that execvp() will search $PATH for you.)

4. 5 pts — The shell should notify the user if the requested command is not found or fails for any reason.

5. When executing a program command (not a built-in command), print out the creation and exit status information for child processes (see section 4 for examples):

   (a) 5 pts — When a child process is spawned, print the process ID (PID) before executing the specified command. It should only be printed once and it must be printed before any output from the command. You can include the program name if you find it useful.

   (b) 5 pts — When a child process is finished, print out its PID and exit status. Revisit Lab 2 for a refresher on how to handle exit status.

6. 10 pts — Your shell should support background program commands using suffix "&." By default, the shell should block (wait for the child to exit) for each command. Thus the prompt will not be available for new user input until the command has completed. However, if the command ends with suffix "&," the child process should run in the background, meaning the shell will immediately prompt the user for further input without waiting for the child process to finish. You still need to print out the creation and exit status of these background processes.

2.2 Built-in Commands

The following commands are special commands (also called built-in commands) that are not to be treated as programs to be launched. No child process should be spawned when these are entered, meaning that they should be accomplished with library/system calls.

- `exit` – the shell should terminate and accept no further input from the user
- `pid` – the shell should print its process ID
- `ppid` – the shell should print the process ID of its parent
- `cd <dir>` – change the working directory of the shell process. With no arguments, change to the user's home directory (which is stored in the environment variable HOME)
- `pwd` – print the current working directory

2.3 Extra Credit (5 pts)

If you implement any of these features, make sure you clearly describe your implementation in your project report. Your summary may be longer than two paragraphs in this case.

- 3 pts — add a built-in command `jobs` that outputs the "name" and PID of each child process of the shell that is currently alive

- 3 pts — add basic output redirection to your shell. In bash (and most shells) if you do `cmd arg1 arg2 argN > filenam` then the output from cmd will go the file `filename`. If the file doesn't exist, it is created. If it does exist, it is truncated (removes all its data, allowing you to effectively overwrite it). Add this feature to your shell.
- 4 pts — add pipes, or something that works like pipes. In bash (and most shells) if you do e.g. `cmd1 arg1 | cmd2 | cmd3 arg3` then the output from cmd1 will be used as the input to cmd2, and the output of cmd2 will be used as the input to cmd3, etc. The vertical bars (and the connection they create between processes) are called pipes. Add this feature to your shell. It may require concepts not yet covered in class.

3 Useful Information

3.1 Command line options

Command line options are options that are 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`). In this example, `/home/me` is also a command line option which specifies the desired directory for which the 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().

```
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, as mentioned above, is the name of the executed program. The program below simply prints each command line option passed in 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 with how command line options work, enter the above program and try it with different values. (e.g. `./myprog I guess these are options`)

3.2 Useful system and library calls

The following system calls will be useful in this project. Read the corresponding man pages (or Google) for those you are unfamiliar with.

- fork – create a child process
- 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 process to exit, with the given return value
- chdir – change working directory

- getcwd – get current working directory
- getenv/setenv – retrieve and set environment variables.
- perror – display error messages based on the value of errno
- strcmp, strncmp, strcpy, strcat, strtok – string manipulation

4 Example and Test Cases

This is an example test case and output for your progrem. Your output does not have to look exactly like this; it is simply meant to give you some idea of how the shell should work and how output looks. A `$` prompt indicates a bash prompt, used to execute your shell, and `308sh>` indicates your shell prompt. Also, 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.

You should test all of the builtin commands in your shell, in addition to several commands that are not builtin. You should test any error handling you use. You do not have to turn in any test code that you use, but don't expect to find errors without testing!

Note that you do not have to output the process name on the "exit" line, but if you're keeping track of it for extra credit, you might as well.

Shell command line:

```
$ ./shell -p "hello> "
hello> exit
$
```

Simple execution:

```
308sh> /bin/ls
[977801] ls
shell.c  shell.o  shell
[977801] ls Exit 0
308sh> pwd
/home/danield/308/
308sh> cd
308sh> pwd
/home/danield/
308sh> cd 308
308sh> pwd
/home/danield/308/
308sh> ps
[977819] ps
   PID TTY          TIME CMD
  3590 pts/2    00:00:01 bash
977797 pts/2    00:00:00 shell
977819 pts/2    00:00:00 ps
[977819] ps Exit 0
308sh>
308sh> nonexistant
[977850] nonexistant
Cannot exec nonexistant: No such file or directory
```

```
[977850] nonexistant Exit 255
308sh> test 2 = 3
[978229] test
[978229] test Exit 1
308sh>
```

Background processes:

```
308sh> sleep 1 &
[977999] sleep
308sh>
[977999] sleep Exit 0
308sh>

308sh> sleep 1 &
[977864] sleep
308sh> sleep 2
[977865] sleep
[977865] sleep Exit 0
[977864] sleep Exit 0
308sh>

308sh> sleep 10 &
[977943] sleep
308sh> kill 977943
[977945] kill
[977945] kill Exit 0
[977943] sleep Killed (15)
308sh>
```