CprE 308 Project 1: UNIX Shell

Department of Electrical and Computer Engineering
Iowa State University

This is a two-week programming project assignment, starting from the week of 9/16 and ends in the week of 9/23. Attendance (10 pts) will only be taken during the first lab session. You can skip the second lab session if you already completed the assignment and has no more questions for the TA.

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.
- 75 pts - A zip file containing all 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
  - 55 pts - program performs required functionality

Due: one week after the second lab session (in the week of 9/30 - 10/4)

2 Project Description

2.1 Purpose and Requirements

In previous lab sessions, you have been using the command line tool *bash* - a UNIX shell program that lets the user interact with the OS. In this project, you will create your own shell program. Your shell does not need to be as sophisticated as *bash*, but it should meet the following requirements:

1. Your shell should accept a `-p <prompt>` option on the command line when it is initiated. The `<prompt>` string should become the user prompt string. If this option is not specified, the default prompt of "`308sh> `" should be used. See section 4.1 for example of the prompt option.
2. Your shell should run an infinite loop, accepting user input and executing commands accordingly, until the user requests to exit.
3. Your shell should support two types of user commands: (1) build-in commands, and (2) program commands.
   (a) Build-in commands are commands to interact with your shell program, such as `cd`, `cd <dir>`, etc. See the list of required build-in commands in section 2.2.

   (b) Program commands require your shell to spawn a child process to execute the user input using the execvp() call. (read *man execvp*, especially the "Special semantics for execlp() and execvp()" section)

4. Your shell should notify the user if the requested command is not found or cannot be executed.

5. When executing a program command, print out the creation and exit status information for child processes (See Section 4 for example):

   (a) 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) When a child process is finished, print out its exit status. Revisit Lab 2 - Experiment 3.8 for how to handle child exit status.

   NOTE: This requirement is only for program commands since built-in commands do not need a child process.

6. 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.

- `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. With no arguments, change to the user's home directory (which is stored in the environment variable HOME). HINT: this can be done with the chdir() function.
- `pwd` – print the current working directory. HINT: this can be done with the getcwd() function.

2.3 Extra Credit (10 pts)

- A "jobs" command to output the name and PID of all your alive child processes. You would need to keep track of your child processes and maintain a list of currently executing background tasks.
- Sometimes it's nice to have your program dump its output to a file rather than the console. In bash (and most shells) if you do `cmd arg1 arg2 argN > filename` 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.
- Make sure you clearly describe how your added feature works in your lab report.

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 the number of command line arguments. This value is always at least 1, as the program name 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.

```
#include <stdio.h>
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;
}
```

Compile the above program and run it with different options such as:

- ./test I guess these are options
- ./test how about "options in quotes"

3.2 Useful system and library calls

The following system calls might be useful in this project. Read the corresponding man pages for those you are unfamiliar with.
Process related system calls:

- 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

String manipulation (Useful link: http://www.cplusplus.com/reference/cstring/):

- strcmp – compare two strings
- strncmp – compare the first n characters of two strings
- strtok – split string with given delimiters string
- strcpy – copy characters from one string to another
- strcat – concatenate one string to another

Other useful library calls:

- 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 error
- open, dup2 – for the extra credit

3.3 A shell program example in pseudo-code

```
int main(int argc, char** argv) {

  while(1) {
    show_prompt();

    read_command();
    if (command is built-in) {
           ...

    } else if (command is non-builtin) {
           ...

    }
  }

  return 0;
}
```

4 Examples and Test Cases

Below are some example test cases and outputs for your program. Your output does not have to look exactly like them; it is simply meant to give you some idea of how the shell should work.

You should test all of the built-in commands (from Section 2.2) in your shell, in addition to several commands that are not built-in. 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!

4.1 Starting your shell program

In this example, a `$` prompt indicates a bash prompt used to execute your shell program (called *MyShell*), and `308sh>` or `HELLO>` indicates your shell prompt. There should be two ways of starting your shell program:

1. **With the -p parameter:**

```
$ ./MyShell -p "HELLO> "
HELLO>
```

2. **Without the -p parameter (use default prompt string "308sh> "):**

```
$ ./MyShell
308sh>
```

4.2 Simple test cases

You may want to prefix all your output messages with some sort of identifier, such as “>>>”, to distinguish the child process information from command output. Note that you do not have to output the process name on the “exit” line unless you are doing the extra credit; just the PID and status is sufficient.

```
308sh> ls                    // user wants to execute program "ls"
>>>[977801] ls               // announce creation of child process: PID and program name
shell.c  shell.o  shell      // program output
>>>[977801] ls Exit 0        // announce the exit of child process

308sh> pwd
/home/timmy/308              // built-in commands do not need a child process

308sh> cd ..

308sh> pwd
/home/timmy

308sh> cd 308

308sh> pwd
/home/timmy/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> nonexistent
>>>[977850] nonexistent
Cannot exec nonexistent: No such file or directory
>>>[977850] nonexistent Exit 255

308sh> echo 123
[978229] echo
123
[978229] echo Exit 1
308sh>
```

4.3 Background process examples (with suffix &)

```
308sh> sleep 3 &
>>>[977999] sleep              // announce the creation of new child process
308sh>                         // return to prompt
>>>[977999] sleep Exit 0       // child exits after 3 seconds
308sh>
```

```
308sh> sleep 2 &
>>>[977864] sleep                  // new background process for the first sleep
308sh> sleep 1
>>>[977865] sleep                  // new foreground process for the second sleep
>>>[977865] sleep Exit 0           // second child exits
>>>[977864] sleep Exit 0           // first child exits
308sh>

308sh> sleep 10 &
>>>[977943] sleep
308sh> kill 977943
>>>[977945] kill
>>>[977945] kill Exit 0
>>>[977943] sleep Killed (15)      // "sleep 10" gets killed before reaching 10 seconds
308sh>
```