0107451 Project 1: Unix Shell

Department of Computer and Communications Engineering
Tafila Technical University

1 Submission

Include the following:

- A cohesive summary of what you learned in this project. This should be no more than two paragraphs.
- A complete listing of your source code; commented and formatted neatly. Use good programming practices (create methods 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.)
- The project should be submitted on Teams by the deadline. Your submission should be two files; a single file of source code and single file of your report.

Grading Criteria:

- Summary — 10% (Well written and specific).

Submitted Source Code

- Compiles — 10%
- Can be read and understood easily — 10%
- Performs the required functionality — 20%
- Answer any question about the code — 50%

2 Project

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 Linux, 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 on the OS based on that input.

2.1 Requirements

1. Your shell should accept a `-p <prompt>` option on the command line when it is initiated. The `<prompt>` option should become the user prompt. If this option is not specified, the default prompt of "`451sh>` " should be used. Read section 3.1 on command line options for more information.
2. Your shell must run an infinite loop accepting input from the user and running commands until the user requests to exit.
3. Each line of user input should be treated as either a builtin command, or a command to be executed. See section 2.2 for a list of the builtin commands your shell should support.

4. For each executed 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 `/some/dir/some/exec` or relative `../../some/exec`) to the executable, unless the executable they wish to run is in one of the directories listed in the PATH environment variable. HINT: execvp() will search PATH for you.

5. The shell should notify the user if the requested command is not found and cannot be run.

6. For each spawned child, 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 other information (name of program, command number, etc) if you find it useful.

7. By default, the shell should block (wait for the child to exit) for each command. Thus 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.

8. 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).

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

   An example is, if the user entered "/usr/bin/leafpad" then the shell would wait for the process to complete before returning to the prompt. However, if the user entered "/usr/bin/leafpad &" 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 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.

2.2 Builtin 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)
- `pwd` – print the current working directory
- `set <var> <value>` – sets an environment variable (which is visible in all future child processes). If there is only one argument, clears the variable.
- `get <var>` – prints the current value of an environment variable

2.3 Extra Credit

- Keep track of the names of your child processes and output them with their PIDs (i.e. "process pid (procnam) "). For this, you need a data structure which can maintain a list of currently executing background tasks – we suggest using a list. Also, now that you are maintaining a list of background tasks, add another built-in command "jobs" that outputs the name and PID of each task running in the background.
- 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.

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 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, strcpy, strcat – string manipulation.
- open, dup2 – for the extra credit

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 `451sh>` 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 unless you are doing the extra credit; just the PID and status is sufficient.

Shell command line:

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

Simple execution:

```
451sh> /bin/ls
[977801] ls
shell.c  shell.o  shell
[977801] ls Exit 0
451sh> pwd
/home/danield/308/
451sh> cd
451sh> pwd
/home/danield/
451sh> cd 308
451sh> pwd
/home/danield/308/
451sh> 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
451sh>
451sh> nonexistant
[977850] nonexistant
Cannot exec nonexistant: No such file or directory
[977850] nonexistant Exit 255
451sh> test 2 = 3
[978229] test
[978229] test Exit 1
451sh>
```

Background processes:

```
451sh> sleep 1 &
[977999] sleep
451sh>
[977999] sleep Exit 0
451sh>

451sh> sleep 1 &
[977864] sleep
451sh> sleep 2
[977865] sleep
[977865] sleep Exit 0
[977864] sleep Exit 0
451sh>

451sh> sleep 10 &
[977943] sleep
451sh> kill 977943
[977945] kill
[977945] kill Exit 0
[977943] sleep Killed (15)
451sh>
```