EC440: Project 1 – Simple Shell

Project Goals:

To understand & correctly use important Unix/POSIX system calls.
To develop a simple shell application.

Collaboration policy:

You are encouraged to discuss this project with your classmates/instructors but are required to develop and turn in exclusively your own solution.
You must be able to fully explain your solution during oral examination.

Deadline (23:59ET no extensions or late submissions)

This project has two deadlines (see below for required deliverables on each)
Deadline 1: Wed September 20 (argument parsing and simple command invocation)
Deadline 2: Tue October 3 (all functionality described below)

Project Description:

The goal of this project is to implement a basic shell which is able to execute commands, redirect the standard input/output (`stdin/stdout`) of commands to files, pipe the output of commands to other commands, and carry out commands in the background.

Your shell should implement a simple REPL (read – eval – print – loop) paradigm. Your shell should use “`my_shell$`” (without the quotes) as prompt. At each prompt, the user should be able to type commands (e.g., `ls`, `ps`, `cat`) which should be executed by the shell. You can access these binaries by searching directories determined by the PATH environment variable that is passed to your shell.

As in reality, commands can have arguments that are separated by whitespace (one or more space characters). For example, if the user types `cat x`, your shell will need to invoke the `cat` binary and pass x as its argument. When the shell has received a line of input, it typically waits until all commands have finished. Only then, a new prompt is displayed (however, this behavior can be altered – see below for details).

Your shell must also be able to interpret and execute the following meta-characters: '<', '>', '|', and '&':

(a) *`command`* '<' *`filename`* In this case, a command takes its input from the file (not stdin). Note that spacing is irrelevant in this case. For example, `cat<file` and `cat <file` are valid inputs. Also, only one input redirection is allowed for a single command. (`cat <<file` is invalid)

(b) *`command`* `'>'` *`filename`* An input following this template indicates that a command writes its output to the specified file (not stdout). Again, spacing is irrelevant (see case a) and only one input redirection is allowed for a single command.

(c) '`|` ' The pipe character allows several commands to be connected, forming a pipeline: the output of the command before "`|`" is piped to the input of the command following "`|`". Multiple pipe signs are allowed on the command line. Spacing is irrelevant (described above).
Example:
"`cat a| sort | wc`" (without quotes) indicates that the output of the `cat` command is channeled to the `sort` and `sort` sends its output to the input of the `wc` program.

(d) The ampersand character '&' should allow the user to execute a command (commands) in the background. In this case, the shell immediately displays a prompt for the next line regardless of whether the commands on the previous line have finished).

For simplification purposes, you should assume that only one '&' character is allowed and can only appear at the end of the line. Also, if the input line consists of multiple commands, only the first command on the input line can have its input redirected, and only the last can have its output redirected. In case of a single command, standard rules apply (e.g., `cat < x > y` is valid, while `cat f | cat < g` is not).

In case of errors (e.g., if the input does not follow the rules/assumptions described above, command is not found, etc.), your shell should display an error message (cannot exceed a single line), print the prompt, and wait for the next input. The error message should follow the template "`ERROR:`" (without quotes) + your_error_message. To facilitate automated grading, when you start your simple shell program with the argument '`-n`', then your shell must not output any command prompt (no "my_shell$ "). Just read and process commands as usual.

To exit the shell, the user must type Ctrl-D (pressing the D button while holding control). You may assume that the maximum length of individual tokens (commands and filenames) is 32 characters, and that the maximum length of an input line is 512 characters. Your shell is supposed to collect the exit codes of all processes that it spawns. That is, you are not allowed to leave zombie processes around of commands that you start. Your shell should use the `fork(2)` system call and the `execvp(2)` system call (or one of its variants) to execute commands. It should also use `waitpid(2)` or `wait(2)` to wait for a program to complete execution (unless the program is in the background). You might also find the documentation for signals (and in particular `SIGCHLD`) useful to be able to collect the status of processes that exit when running in the background.

Some hints:

1. A simple shell such as this needs at least a simple command-line parser to figure out what the user is trying to do. To read a line from the user, you may use `fgets(3)`.
2. If a valid command has been entered, the shell should `fork(2)` to create a new (child) process, and the child process should `exec` the command.

3. Before calling `exec` to begin execution, the child process may have to close `stdin` (file descriptor 0) or `stdout` (file descriptor 1), open the corresponding file or pipe (with `open(2)` for files, and `pipe(2)` for pipes), and use `dup2(2)` or `dup(2)` to make it the appropriate file descriptor. After calling `dup2(2)`, close the old file descriptor.
4. The main challenge of calling `execvp(2)` is to build the argument list correctly. If you use `execvp(2)`, remember that the first argument in the array is the name of the command itself, and the last argument must be a null pointer.
5. The easiest way to redirect input and output is to follow these steps in order:

   (a) open (or create) the input or output file (or pipe).

   (b) close the corresponding standard file descriptor (stdin or stdout).

   (c) use dup2 to make file descriptor 0 or 1 correspond to your newly opened file.

   (d) close the newly opened file (without closing the standard file descriptor).
6. When executing a command line that requires a pipe, the pipe must be created before forking the child processes. Also, if there are multiple pipes, the command(s) in the middle may have both input and output redirected to pipes. Finally, be sure the pipe is closed in the parent process, so that termination of the process writing to the pipe will automatically close the pipe and send an EOF (end of file) to the process reading the pipe.
7. Any pipe or file opened in the parent process may be closed as soon as the child is forked – this will not affect the open file descriptor in the child.
8. While the project assignment talks about system calls, feel free to use the libc wrapper functions, documented in their corresponding beautiful man pages instead.

Submission Guidelines:

Your shell must be written in C and run on Linux. It must compile without any warning/errors and run on ec440.bu.edu. (i.e., `gcc -Werror` will be in use)

In your home directory create a folder (e.g., project1) and place `README`, `makefile`, `myshell.c` files (potentially, `myshell.h`) there. Switch to the project1 directory and execute `submit1`.

A confirmation mail of your submission is sent to your account on ec440.bu.edu. You can read this mail by executing `mail`.

To automatically test your solution, we execute `make` (no arguments will be provided to `make`). After `make` terminates, we execute the binary `./myshell` for extensive testing. (Please make sure that executing `make` compiles your source code and generates a binary named `myshell`)

In the `README` file explain what you did. If you had problems, tell us why and what.

You are allowed to resubmit your files. The last submission before the deadline will be graded

Do not forget that you must support the '`-n`' argument to suppress the output of the shell prompt for automated testing.

Details on Mini-Deadline 1 (Sept. 20th):

For the first (mini-) deadline, your shell must demonstrate that it can accept user-input, parse that input, and invoke simple commands with arguments. That is, you will need to implement the parsing logic, and at least steps 1,2, and 4 from the “Some hints” section above. In order to test your shell-program, it must REPL and for each command-line output the following:

<first command>
<second command>
…
<nth command>
whatever execvp() of the first command produces

For illustration purposes consider the following example:

```
/bin/echo foo | /bin/grep -o -m 1 f | /usr/bin/wc -c > /tmp/x
```

Your program would parse this command line into 3 different commands (technically, the last argument is a file) and output (line numbers for illustration only).

```
1: /bin/echo foo
2: /bin/grep -o -m 1 f
3: /usr/bin/wc -c > /tmp/x
4: foo
```

Above, lines 1-3 are the three commands making up the command-line, line 4 (the string `foo`) is the result of the first command (`/bin/echo foo`).

Oral Examination:

Deadline: Wed, October 11.

You are required to meet with one member of the course staff during office hours to explain your solution.

Oral examination will take place between Wed., October 4, and Wednesday, October 11.