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 and instructors. You **must** develop your own solution. While you **must not** share your parser code with other students, you **are encouraged** to share testcase code that you developed to test your solution for this assignment.

Deadline:

Project 1 is due February 16 at 4:30 PM EDT.

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. In the *optional previous* assignment, you developed a shell parser that implements the same input requirements as this assignment. If you started or completed challenge 0, be sure to use that as your starting point for this assignment!

Your shell must implement a simple REPL (read – eval – print – loop) paradigm. Your shell must use “my_shell$” (without the quotes) as prompt. At each prompt, the user must be able to type commands (e.g., ls, ps, cat) which shall 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 (**HINT**: read the man pages for the various execv wrappers. One of them performs the search for you).

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 an 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 around the < operator. 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) *command1 | command2*: 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 "|". A series of multiple piped commands is allowed on the command line. Spacing is irrelevant (described above). Example: `cat a| sort | wc` 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) *command &* The ampersand character '&' should allow the user to execute a command (or pipeline of 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 or are still in progress.

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 with no pipes, consider the command as both the first and last of a pipeline. E.g., `cat < x > y` is valid, while `cat f | cat < g` is not.

During regular execution (i.e., with no errors), your shell must *only* print the prompt ("my_shell$", or an empty prompt when run as `./myshell -n`). Extra output (e.g., from debugging printf calls) will result in test failures.

In case of errors (e.g., if the input does not follow the rules/assumptions described above, a command fails to execute, etc.), your shell should display an error message (cannot exceed a single line), then print the prompt, and wait for the next input. The error message should follow the template `ERROR:` + 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.

When the user types Ctrl-D (pressing the D button while holding control), the shell must cleanly exit. 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 of commands that you start. Your shell should use the `fork(2)` system call and the `execve(2)` system call (**HINT**: you can use its `execvp(3)` library wrapper) 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.

Example Shell Sessions

```
root@ea3aca4ec300:/ec440# ./myshell
my_shell$whoami
root
my_shell$pwd
/ec440
my_shell$/bin/pwd
/ec440
my_shell$try to run this!
ERROR: try: No such file or directory
my_shell$ls -al
total 36
drwx------. 6 1000 1000   150 Jan 27 06:16 .
drwxr-xr-x. 1 root root    30 Jan 27 06:16 ..
-rw-------. 1 1000 1000  1779 Jan 22 01:55 .bash_history
-rw-r--r--. 1 1000 1000    18 Aug  5  2019 .bash_logout
-rw-r--r--. 1 1000 1000   141 Aug  5  2019 .bash_profile
-rw-r--r--. 1 1000 1000   376 Aug  5  2019 .bashrc
drwx------. 3 1000 1000    20 Jan  8 04:17 .config
drwxrwxr-x. 3 1000 1000    17 Jan  8 04:52 .local
drwx------. 2 1000 1000    29 Jan 13 21:50 .ssh
drwxrwxr-x. 2 1000 1000   114 Jan 27 06:16 hw1
-rwxr-xr-x. 1 root root 19584 Jan 27 06:16 myshell
my_shell$ls -al | wc -l
12
my_shell$ls -al |wc -l>lines_in_ls
my_shell$cat lines_in_ls
12
my_shell$root@ea3aca4ec300:/ec440#
         ^-- Note: I pressed CTRL-D here

root@ea3aca4ec300:/ec440# ./myshell -n
whoami
root
pwd
/ec440
root@ea3aca4ec300:/ec440#
```

^-- Note: I pressed CTRL-D here

You may find it easier to automate your own tests with the following pattern:

```
echo 'ls' | ./myshell
```

Given Code:

No new code is provided for this assignment. If you chose to implement challenge 0 (where we did provide some starting code), we recommend that you begin with your solution to that challenge.

Some Hints:

1) **First**: If you haven't looked at challenge 0 (or if you don't remember it), review that assignment as well as its hints. That assignment was designed to give you a head start on this assignment, so all of those hints still apply! You are permitted to ignore challenge 0 if you prefer to solve this challenge in a different way.
2) Your solution to challenge 0 includes a `pipeline_build()` function that accepts a single line of text as its input argument, and outputs a structure containing all of the data you need to implement a solution for this assignment. You can use that function in challenge 1. To obtain a single line of text from a user, you can use the `fgets(3)` function.
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(3)` is to build the argument list correctly, as defined in its man page. Your solution for challenge 0 should already have your arguments in that format.
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 mostly about system calls, feel free to use the libc wrapper functions, documented in their corresponding man (section 3) pages.
9) Develop your own test cases. Your test cases from challenge 0 are still valid, so don't delete those. By keeping your old tests, you ensure that new features you add won't break old ones you already figured out. While the autograder has some descriptive explanations of what it is trying to do, it will not show you how or why your submitted code is failing a test. A good test suite for this assignment has many test cases. A good

test case has some kind of expectations about how a program will react to different inputs and system state changes, and will fail when those expectations are not met.

10) Let your software development tools help you:
    a) Don't ignore compiler warnings. If the compiler tells you that something looks suspicious, it is because a compiler engineer determined that the suspicious code pattern is likely to lead to a bug or some other undesirable behavior.
    b) The autograder won't accept extra text output beyond what is specified in this document (output of commands, and your messages when an error is encountered). Instead of repeatedly recompiling with new printf() statements to figure out what is happening in your program, use a debugger such as *gdb* to inspect and modify your program while it is running.
    c) Use memory-aware tools, such as *valgrind*, to detect memory leaks or bad memory accesses.
    d) Use sanitizers when you are building your tests (e.g., ASAN, UBSAN). This is easy if you start with the makefile provided in challenge 0. For example, to use the address sanitizer, build and run your tests by running `make check CFLAGS=-fsanitize=address LDFLAGS=-fsanitize=address` (you may need to run make clean first, if you already built without sanitizers).
    e) If you aren't sure whether you are sending the correct arguments to a syscall, run your shell inside the `strace` tool to observe when it makes syscalls, and which arguments it sends to those syscalls. The -e option may be particularly useful, since you can use it to *only show* the syscalls you are interested in.

Submission Guidelines:

Your shell must be written in C and run on the Linux Gradescope environment (very similar to the container environment we provided along with challenge 0). Submit your ***source code and makefile*** to Gradescope. The shell must execute from a file named `myshell`, which is built by running `make` in your submission. You are permitted (and recommended) to use the starter code and starter makefile we provided in challenge 0. You will receive your submission's grade shortly after submitting the solution.

Your final submission must also include a `README` file (README.md is also okay), so we recommend that you start working on the `README` as part of this assignment. If you relied on any external resources to help complete this assignment, note the resource and the challenge it helped resolve. Use this file to provide any additional notes you would like to share with instructors about your submission. Aside from that, it doesn't need much detail (though it may benefit your future self if you give a short description of the purpose of the project).

Oral Exams

When the submission deadline is reached, we will start oral exam sessions over Zoom. In these sessions, we will ask you to explain how some parts of your program work. Details about how to schedule an oral exam time will be posted in the week leading up to the exams.