Project 1 Simple Shell Program

Xian Ma

October 1, 2024

1 Summary of Project 1

1.1 Design Ideas

Task 1

This task was relatively straightforward since it only required basic understanding of how GitHub works. I forked the repository that contained the starter code and cloned it onto my local machine. I had also launched a codespace on GitHub, which served the same purpose but it allowed me to more easily debug my code when an error occurred.

Task 2

This task required that I prepare my newly forked repository for development. I copied the starter code into my repoository (src/lab.h). This file outlined the necessary methods that would need to implement for the shell. Since I was following along with the powerpoint and the vitepress site, I did not add in the starter code for the tests (tests/test-lab.c) since it would cause my make to show errors.

Task 3

This task required me to implement the a print version function for my shell. This function was rather simple and only required me to modify the code in src/lab.c. At first I struggled since I didn't understand if I was supposed to implement the function in lab.c or if I should put it in main.c. At first I implemented it in main.c, but later on I realized the error of my ways. I realized that I was suppose to implement it in the parse_args function.

Task 4

This task required me to implement user input for the shell. Initially I thought that the user input would be in lab.c since that made the most sense to me. I implemented it in get_prompt function and called in in main.c. I later found out that it belonged in main.c since main.c is the entrance to our program. I implemented the task without much trouble by leveraging the libraries readline/readline.h and readline/history.h.

Task 5

This task required me to implement a custom prompt for my shell. Initially, I was confused about what this entailed. After some deliberation, I realized it meant allowing the user to define a custom symbol for their prompt. I implemented this by retrieving the prompt using getenv(env). If the result was 'NULL', it was replaced with the default "shell>". Otherwise, I used the custom prompt provided by the user.

Task 6

This task required me to implement a slew of built-in commands that were handled by the shell itself. I was required to implement the following commands: exit, cd, and history. Before starting on the implementation of the commands, I implemented the sh_init and sh_destroy functions. These functions were used to initialize the shell and destroy the shell, respectively.

After I was done with the functions, I started on the implementation of the built-in commands. The exit command was rather simple since it only required me to check if the argument was "exit" and if it was, I would use exit(0) to terminate the program.

When implementing the cd command, I noticed that there was a method called change_dir in the lab.h file. I simply implemented this function signature. I checked if the second argument after cd was empty, and if it was, I would use getenv to obtain the HOME directory. I then created a check for the HOME directory in the case that it was unable to be found. I would use getpwuid(getuid()) to find the home directory of the user.

The history command was stumped me at first since I didn't know how to obtain the history from the history library. After some research, I found that I could use history_get and a loop to obtain the history. I was able to store the history in a HIST_ENTRY pointer.

Task 7

This task required me to implement the shell's process creation feature. I first created and initialized the necessary variables in cmd_parse. I used sysconf and _SC_ARG_MAX to obtain the maximum length of the arguments that can be executed in one line. I then allocated the char **argv array using the maximum limit obtained from sysconf(_SC_ARG_MAX). I would then trim the whitespace from the arguments and use a delimiter to parse out each argument. This method was used in do_builtin to get the arguments to execute with execvp.

Task 8

This task required me to implement the shell's signal handling feature. I needed to ensure that it ignored specific signals (SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, SIGTTOU) during its operation. This was achieved using the signal function to set the aforementioned signals to SIG_IGN. In the child process, I would reset the signals to their default behavior of SIG_DFL to allow for proper termination. I used the system calls tcgetpgrp and tcsetpgrp to manage the foreground process group. If no process is executed, the shell simply displays a new prompt and ignores any input on the current line.

This was the first time I used the signal function. Luckily with the provided example on the vitepress site, I was able to figure out how and where to use it in my program.

Task 9

In this task, I implemented the shell's capability to manage background processes initiated by the user. When a command ends with an ampersand (&), the shell executes the process in the background, allowing the user to continue entering additional commands without delay. Upon starting a background process, the shell displays the job number, process ID, and the command executed. Once the process completes, it notifies the user by displaying the job number, the message "Done," and the command that was executed.

To aid in implementing this functionality, I created the helper function check_background_processes, which monitors the status of background processes. This function utilizes the 'WNOHANG' option with the 'waitpid' system call to efficiently check for completed processes.

Task 10

This task required me to implement a new built in command called jobs. The function would print out all the background commands that are running or are done. The first job was assigned the job number 1 and each additional job number was incremented by 1. I initially struggled to implement this feature because I attempted to implement it without creating a new struct. After some research on how to store the information in C, I created a new struct called job. This struct stored the job number, the process ID, the command, and a status. I then created a method to print out the jobs in the required format.

To more easily manage the jobs, I created multiple helper functions. These helper functions help me add and remove jobs from the job list as well as update the status of the jobs. With these functions, implementing the jobs command was straightforward.

1.2 Functionality Achieved

- **Print Version**: Prints the version of the shell specified in lab.h with lab_MAJOR_VERSION and lab_MINOR_VERSION.
- **Custom Prompt**: Allows the user to define a custom prompt by specifying the "MY_PROMPT" environment variable.
- **User Input**: Allows the user to input commands into the shell.
- **Command Parsing**: Parses the commands inputted by the user and executes them.
- **Background Process Management**: Allows the user to run commands in the background.
- **Job Management**: Allows the user to view all the background commands that are running or are done.
- **Signal Handling**: Allows the shell to ignore specific signals and reset the signals in the child process.
- **Built-in Command Execution**: Allows the shell to execute built-in commands such as exit, cd, history, and jobs.
- **Other Commands Execution**: Allows the shell to execute other commands using execvp.
- **Prevent Signal from Interrupting**: Allows the shell to prevent signals from terminating the program.