PA1 Report

CLI Simulation in C

CS307 - Operating Systems

Mert Ekici 26772
ekicimert@sabanciuniv.edu

I. Command Line Interface

Command Line Interface (CLI) or Shell is a user program that allows users to interact with the OS by entering commands. In a loop, shell first shows the prompt and then waits for the command (and following arguments if needed). After getting the command, shell creates a child process for executing the command user entered. During the execution by the child process, the parent process waits for the command to end. When the termination occurs from the child process, the shell process starts a new iteration by prompting for the next command. In this programming assignment, we are asked to implement a shell with some basic functionalities such as redirectioning and background job handling. In the following sections, I will explain how I designed the shell, parsing method I used and how I implemented the given functionalities.

II. Command Struct

First, I decided to create a struct that stores command information which makes command access and modifications easier. Struct ***command*** contains 8 different variables.

- name: char pointer
- input: char pointer
- input2: char pointer
- option: char pointer
- rd: char
- rdFile: char pointer
- bg: char
- size: integer

Explanation of each variable goes follows; *name* is the name of the command, ***input*** and ***input2*** variables are for command inputs (and option inputs if exist), ***option*** variable holds the option string such as “-i” or “-l”, ***rd*** is a character that shows the symbol of redirection, ***rdFile*** stores the filename if ***rd*** equals to “>” (output redirectioning), ***bg*** holds the ‘&’ (ampersand) character if the command will be executed in the background (NULL otherwise), and ***size*** variable represents the size of the command array that is going to be the first parameter of ***execvp()*** function.

III. Parsing

Unlike normal shells, our command line interface gets its input from the given txt files. To execute those input commands, we need to parse each command line of the txt file and categorize each unit of the command to prompt the whole command with its subsections. As a common strategy, I used ***fopen()*** function to open files and read each line to ***line*** variable by using ***getline()*** function. Each iteration of the ***while*** loop with ***getline()*** function handles one line of the file which is equivalent to a command line. For each command line, first I created a default ***command*** (struct) and then defined three variables: ***rd*** (0 if no redirectioning, 1 if '>' output redirectioning and 2 if '<' input redirectioning), ***bg*** (1 if the command is a background job, 0 otherwise) and ***option_input*** (it is not necessary for this assignment but it helps to handle option inputs such as "-i string" or "-cf filename", I implemented it because I like improving my coding skills with these little challenges). The variables ***rd*** and ***bg*** will be used in Redirectioning and Background Job functionalities. Before starting "parsing the line", we escape special characters which will be explained in the next chapter. Next, I divided ***line*** into tokens by using ***strtok()*** function with ***whitespace*** as delimiter. Then, each token is investigated using ***strchr()*** function to decide its type among ***input, option, redirection, background***. Finally, ***prompter()*** function prints the command parts. Additionally, ***fflush()*** function is called after every ***printf()*** function without ***newline*** character to print immediately.

IV. Special Characters

In ASCII code, some characters are special and they must be escaped to prevent the command from disrupting. In this section, I will explain the special characters that I identified and how I escaped them by using other special characters. Two functions are used for special character handling, ***escapeChar()*** and ***specialChar()***. Identified special characters that might be problematic for the command line are;

- Newline -> '\n'
- Carriage Return -> '\r'
- Double Quotation Mark -> '\"'
- Single Quotation Mark or Apostrophe -> '\''
- Vertical Tab -> '\v'
- Horizontal Tab -> '\h'

The strategy is replacing detected special characters (from the list above) with ***whitespace***, since whitespace will be used as delimiter in ***strtok()*** function. Thus, ***escapeChar()*** takes a string and a char as an argument and replaces all the occurrences of the char with ***whitespace*** in the given string. Next, ***specialChar()*** does this replacement for each identified special character from the list above. If ***'&', '>', '<'*** and ***'-'*** characters are used in the string input (when double quotes are removed) parser does necessary checks before categorizing each token.

V. Command Array

Parsing each line into command parts is not enough. We need to create a command array to pass it as an argument to ***execvp()*** function and it must be in the correct format. The function ***commandArray()*** takes three arguments, first one is ***char pointer array*** (the array we will use in execvp), the second one is ***command struct*** and the third one is option ***integer*** to arrange option inputs (additional functionality). Since the command format is known, it starts a ***for*** loop using the size of ***command (cmd.size)*** which is updated with the function ***sizeUpdate()*** that only counts the parts that are going to be used in the argument list for ***execvp()*** which are ***name***, ***input***, ***input2*** and ***option***. In this loop, each element of ***arglist[]*** array will be assigned (among previously counted command struct variables) considering the opt parameter (0,1 or 2). The function ***commandPrinter()*** is used for debugging.

VI. Redirectioning

After the parsing part, ***prompter()*** function prints the command parts and the size of the command is updated with ***sizeUpdate()*** function. Based on ***rd*** variable which was assigned in the parsing part, there are three main conditions for the command execution.

- If ***rd*** is ***0***, that means no redirection detected and the process goes like a regular command execution in C. I used ***fork()*** function, called ***execvp()*** in the child process and used ***argument_list[]*** filled by ***commandArray()*** function as the second parameter and for the first parameter, ***argument_list[0]*** is used (command name). Parent process waits for the child process to be completed using ***waitpid()*** with the specific ***pid*** of the child process (if ***bg*** is 0). I preferred ***waitpid()*** instead of ***wait()*** because using ***wait(NULL)*** might cause some problems such as waiting some of the processes that run in the background rather than waiting the related child process.
- If ***rd*** is ***1,*** there is an output redirectioning with '>' character. In this case, I first close the **stdout file descriptor** by ***close(STDOUT_FILENO)*** in the child process. Then, I open the redirection file by ***open()*** function with ***cmd.rdFile*** as its first parameter and this allows me to write the output of the command into this file. Basically, output stream is redirected to the file we selected. Then ***execvp()*** function is called and the rest is the same.
- If ***rd*** is ***2,*** input redirection happens ('<' character). Since the commands that accept inputs can also take them as arguments, I assign the filename that has been stored in ***cmd.rdFile*** to ***cmd.input*** or ***cmd.input2*** depending on the availability. Then, ***sizeUpdate()*** is called again (because there is an extra input) and new ***argument_list[]*** of ***commandArray()*** is passed to ***execvp().*** Simply, it converts the redirection input to command input and calls ***execvp()*** as usual.

VII. Background Jobs

As mentioned in Redirectioning part, there are three conditions depending on the value of ***rd*** variable. In each of these conditions, ***bg*** variable is checked and two different possibilities occur based on its value.

- If ***bg*** is 0, then there is no background job (or command is not going to be sent to the background). In this scenario, the command is executed with ***execvp()*** function call and the child process is waited by ***waitpid()*** function. Parent process waits for the child process and blocks the next process which indicates that command is getting executed in the foreground.
- If ***bg*** is 1, then the command will be executed in the background. For this case, we can send the command to the background with a small change. If we ***do not wait*** for the child process to terminate, then it completes the execution in the background without blocking the next process. Thus, if we remove the ***waitpid()*** function call from the regular command execution code of ***bg == 0 case***, we obtain background job functionality.

VIII. Wait

If user enters ***wait*** command or input command file includes a ***wait*** command without a given ID which represents our case, then it waits for all active child processes. Hence, providing this functionality is simply achieved by using ***wait(NULL)*** function call in a ***while*** loop which waits for all active child processes. At the end of ***commands.txt*** reading process, shell must also wait for all the continuing background jobs to finish as the PA1 guideline suggests. Due to this reason, I implemented the same functionality as the ***wait*** command, after the end of ***while getline()*** loop (when all commands are executed).

IX. Additional Functionalities

Besides these functionalities, I also implemented ***cd*** command functionality with ***chdir()*** function which also supports ***"cd .."*** with the ***prevdir*** variable that stores the previous directory using ***getcwd()*** function. Furthermore, multiple command inputs and option inputs are supported as well. In addition to the extra functionalities I implemented, ***prompter()*** function and its arguments should be mentioned. Since I complete the "special character escape" part before prompting the command parts, inputs and options will not be shown with double quotes or other special characters if they exist. Thus, I create two variables for each iteration of ***getline() loop***: ***pureLine*** and ***pureCmd***. They are used in the ***promptedCommand()*** function that updates ***pureCmd*** based on ***pureLine*** and eventually ***pureCmd*** is passed to ***prompter()***. As a result inputs like ***"rwx"*** will be prompted with their special characters.