CS307 (Operating Systems)

Programming Assignment #2 Report

Hagverdi Ibrahimli (30014)

In this report, I shall explain the design decisions that I made in the implementation of the programming assignment, the CLI simulator. I decided to utilize the C++ programming language for this assignment.

In the beginning of the program, I start by importing the necessary libraries for the implementation (such as the iostream, sstream, etc.). Then, I declare *p_thread_mutex_t* type mutex for the shell process. Also, I initialize two global vectors for bookkeeping purposes: for threads and processes, that I will explain in the upcoming paragraphs. In the main function of the program, we open the corresponding input and output files: "commands.txt" and "parse.txt", respectively. We are trying to parse commands from "commands.txt" using the *getline()* method, and for each line parsed we do the following:

- Initialize an array of size 4, all slots set to NULL. The idea is that this array will constitute the command for the *execvp* system call and the first three slots of this array will be the *command-name*, possible *input*, and a possible *option*. To note the case of handling special characters in the options such as the '-' character, we convert each part of this command to a *char** array with *.c_str()* function and take a dynamically allocated copy through *strdup()*. This ensures a successful input to the *execvp*. Also, we initialize some other variables to be informed whether the command includes a redirection and/or it is a background process that we should handle differently.

- We write all the information about the command constituents to the "parse.txt" file in a predefined format. Also, we use the *flush()* method to make sure that output is visible immediately to the console.

- Then, we arrive at command processing. If the current command to be executed is *wait*, then we run a function called *WaitForPT()* that does the following:
    - Wait for all the running processes, by calling *waitpid* on their process IDs stored in the process ID vector.
    - Clear the process ID vector.
    - Wait for all the running threads, by calling *pthread_join* on their thread IDs stored in the thread ID vector.
    - Clear the thread ID vector.

- Else, if the command includes a redirection sign '*>*', then output is to the redirection file. We fork from the shell process. In the shell process, we check whether this command is a background process or not. If it is a background process, we add its process ID to our process ID vector, else we wait for it to finish execution. In the forked process, the command process, we first open the redirection file and change the *STDOUT_FILENO* descriptor to point to the redirection file with *dup2* function call. Then we close the unused file descriptor and execvp the command in that process.

- Else, the output of the command is to the console. First, we initialize a pipe, that shall be unique for each command process to effectively communicate with their parent, the shell process. This decision was critical because, in this assignment, one of the main goals was to sustain synchronization among multiple running processes printing their output to console without any interruption. Thus, we also initialize a thread, one per each command process, that should be running in the shell process, performing the task of reading command processes' messages and outputting them to the console. This ensures that shell process can continue with its main task of parsing and executing commands, while also outputting the command processes' messages to the console, thus we have a multithread program. Then we fork from the shell process.

    - In the case of continuation of shell process after the fork, we first close the write end of the pipe. Then we initialize the thread to begin listening to the pipe, through the *listener* function which we will explain next. Then, if it is a background process, we add its process ID and corresponding thread ID to their corresponding vectors. Else, we wait

for this process to finish by waiting for both the process and the corresponding thread to finish their execution.

- In the forked process, aka command process, we first check whether there is a redirection sign '<'. If yes, it means that command must get its input from the corresponding redirection file. Thus, we open the file and via *dup2* command manipulate its *STDIN_FILENO* descriptor to point to the file and close the unused file descriptor. Then we do the following:
  - We redirect the *STDOUT_FILENO* descriptor of the process to the read end of the pipe, that will enable communication with the parent, the shell process. Then we close the unused file descriptors and call *execvp*.

In the abovementioned *listener* function, the assigned thread does the following:

- First, it opens the corresponding read end of the pipe (established for communication between a specific command process and the shell process) via its file descriptor. Then, we also initialize a boolean variable called *still_trying*, first set to true.
- While *still_trying* is true, the thread first tries to acquire the mutex of the shell process. Once it gets it, it tries to read from the pipe and if it reads before anything is written to the pipe, then it unlocks the mutex giving other threads a chance and tries to acquire it again, basically to try again until it performs a successful read operation.
- If it acquires the lock and successfully reads from the pipe, it first prints its own thread id and the first line read to the console then it keeps reading and printing until the *EOF* signal is sent by the command process to the pipe, so that the thread sets the *still_trying* to false, unlocks the mutex, and finishes execution by closing the pipe.

Once the shell process is done with parsing and executing the command processes, as we described above, it closes the corresponding "parse.txt" and "commands.txt"files, and calls *WaitForPT()* function to wait for all background processes and the corresponding threads executing output job for them.