CS307 (Operating Systems)

Programming Assignment #2 Report

Hagverdi Ibrahimli (30014)

The CLI simulator serves as the focal point for this report. Its primary goal is to execute commands parsed from a file ("commands.txt"). The simulator employs robust design principles to handle background processes, input/output redirection, and synchronous interactions between processes and threads.

Beginning with the main (shell) process, the initial step involves parsing each line from "commands.txt," extracting essential details like the command, inputs, options, and potential redirection. After this extraction, we initialize an argument array of size 4, last slot reserved for the *NULL* terminator. The first 3 slots of this array are reserved for the command-name, input (if any), and option (if any). This argument array will constitute the command line for the *execvp* function in the following parts of the program. Also, it is important to note that, by providing the arguments to the *execvp* through this argument array, we eliminate the problem of special characters in the command execution.

Subsequently, each command information is logged in "parse.txt" for comprehensive record-keeping. A crucial aspect of the simulator is its ability to differentiate between foreground and background processes. The presence of the '&' symbol at the end of a command signifies a background process, allowing the child process to operate independently while the parent process (shell process for each of the command processes) proceeds with its tasks without waiting for completion.

For the possible ‘wait’ command, the program does not fork any child process and is assigned the task of waiting for all background processes and threads to finish their job with the help of *WaitForPT* function, so that it can continue. For this purpose, to effectively implement bookkeeping, we initialize two global vectors: one for process IDs, other for the thread IDs. How and why these processes and threads are created will be explained in the subsequent parts of the report.

The code adeptly manages input and output redirection through the identification of '<' and '>' symbols. In the case of ‘>’ as the redirection sign, the command essentially outputs to the corresponding redirection file and thus, there is no output on the terminal. This is an easy case to handle such that we first fork a child process from its parent (the main process). In the forked child process, we open the redirection file and change the *STDOUT_FILENO* descriptor of the process to be directed to the file (with the help of *dup2*), so that whatever it outputs is directed to the given file, not to the terminal. After forking, if the parent process continues to execute, we simply check if this process is a background process or not. If it is a background process, we add its process ID to our vector of process IDs for bookkeeping purposes. Else, if it is a foreground process, the main process waits for it to finish before fetching the new command (if any).

For the other cases, either the redirection sign is '<' and/or the output of the command to be executed is to be directed to the terminal, so we follow a different approach. First, we create a unique pipe that will be used as a means of communication between the main process and the child process to be forked that should execute the command. Also, we create a thread (uniquely for each command process that belongs to this section in the report) that shall be assigned the task of printing to the console, which we will explain in detail. Then we fork the new child process. In the child process, we check if the command contains the '<' input redirection sign, and if it does, we open the corresponding input redirection file and redirect the *STDIN_FILENO* descriptor of the process to the file with *dup2*, so that it reads the input from that file.

After implementing this step specially for the commands including '<' redirection sign, we change the *STDOUT_FILENO* descriptor of the command process to be redirected to the pipe, so that the shell process can listen to the pipe and act accordingly. Then we *execvp* the command in the child (command) process. If the shell process executes after the *fork()* statement, we first close the write end of the pipe, since the shell will not perform write operations to the pipe. Then, in the shell process, we assign the unique thread the task of listening to this unique pipe's read end and logging the output of the command to the console. Also, depending on whether this new command process is a background process or not, we do the following: if it is a background process, we add its process ID and the corresponding listener thread ID to their corresponding global vectors in the program. If it is a foreground process, on the other hand, we wait both for the process and its corresponding listener thread to finish. The reason for such an implementation is because:

- We needed to sustain synchronization among the console outputs of different processes, such that output of one command process to the console does not get interrupted by the output of the other. And since a mutex can belong to only one process, and we have many command processes executing different commands, we needed a way to sustain this synchronization in the following manner: All command processes have a unique pipe that they utilize to communicate with their parent (shell process) and shell process assigns a unique thread to listen to the read end of the corresponding pipe to output the results of the command processes to the console. The purpose of using this thread idea is simply because we want the main process to continue fetching commands independent of outputting the command process' outputs to the console. This is vital for the concurrency and the multi-threaded nature of the program.
- Synchronization, a critical consideration in concurrent programming, is achieved through the implementation of a global mutex (`pthread_mutex_t mutex`). This mutex acts as a gatekeeper, preventing concurrent access to shared resources and mitigating potential race conditions in the listener function that each thread created for each command process (described above) is assigned to. In the listener function, the thread of the shell process for the corresponding command process first tries to acquire the lock of the mutex. Once it acquires it, it listens to the read end of the pipe for any possible signals from the command process. As long as there is something written to the pipe, the thread extracts that information and writes that to the console, encapsulated by its thread id for clarification purposes. Once the command process

sends *EOF* signal to the pipe, the thread finishes execution by unlocking the mutex. This guarantees a streamlined and orderly process.

- To effectively manage background processes, we employ two vectors: `threads` and `pids`. These vectors keep track of threads and process IDs (PIDs) associated with background processes. The `WaitForPT` function, executed at the program's conclusion and for the implementation of the wait command itself, waits for all background processes and threads to complete, utilizing `waitpid` and `pthread_join`. The vectors are subsequently cleared, concluding the execution cycle.