CSE3033 - Operating Systems Assignment 2 Report
Fall 2020

Ali Reza Ibrahimzada	Huzeyfe Ayaz	Sena Dilara Yangoz
Student ID: 150119870	Student ID: 150119700	Student ID: 150119706

Note: For us to be able to do the bonus part, at each command prompt of myshell, a while loop checks if the entered character is ^. Therefore, you might be wondering why the first button pressed does not have any effect. The reason is getch() in while loop becomes false and then your subsequent typings will appear on the screen. For instance, if you want to write " ls -l ", then the best way to do it is to write " lls -l ", or any other character before your desired command. OUR MYSHELL WOULD WORK NORMALLY IF WE REMOVED THE BONUS PART, but we wanted to do the bonus. We spent a lot of time trying to solve this problem, and this is the best we could do. If you want to test our project without bonus, please comment lines 318 - 339 (inclusive). Thank You.

Execution

For the execution part, we use a variable to differentiate between program execution and built-in functions. If the variable is true, we fork a child for the current process and check if it is successfully created. Inside the child process, first we check if we have I/O redirection. The necessary explanations about I/O redirection have been written in later parts of this report. Since we are only allowed to use execv() rather than execvp(), in the first step we get all of the directories in the PATH environment variable. Then we loop over all directories and check if the program entered by the user exists in any of the directories. If so, we execute execv() with the absolute path we found in that iteration with the arguments array. We did not face any real difficulty finishing this section. In the parent process, there are 2 scenarios that are given below:

background

If the end user enters a command with &, we consider it as a background process. Inside the parent process, we create a linked list node and store the details of the current background process, and add it to the end of the linked list. After adding the node and executing the program, we immediately move on and ask the user for another input. This storing procedure is vital in the ps_all built-in functionality which will be discussed later.

foreground

Similarly, if the end user enters a command without &, we assume it as a foreground process. It is not necessary to store the contents of a foreground process, hence after execution the parent process will wait until its created child (foreground process) is terminated.

Built-in Commands/Functions

In the beginning, once the user enters their input and setup function parses it, we compare the zeroth index of the args array with possible built-in commands' names. In case of a match, we

set the variable for differentiating between built-in commands and normal commands to false, and perform the necessary actions based on the command. Changing the mentioned variable helps us to organize the code in a better way and avoid child creation for built-in commands.

ps_all

The ps_all built-in function is used to print the status of running and finished background processes. In order to make ps_all functional, we thought what we need to do is to keep background processes in a data structure. Therefore, we chose a linked list data structure in which each node is a struct storing background process id, command line arguments, process job id, and a pointer to the next node. When the end user runs ps_all, we loop over the linked list twice (once for running, and once for finished processes), and in each iteration we call the waitpid() with WNOHANG. Then, based on the return value (0 if running, pid if finished) of the function call, we decide whether a background process has been finished or not. If it is finished, we display it once and then remove that specific node from the linked list.

^Z

In this part, firstly, we initialized the signal, then set up the signal handler for ^Z signal. While doing this, we have given the necessary warnings by considering all the situations we may encounter such as failing to set signal handler, etc. In the signal handler function, catchCTRLZ, first we check if there is a foreground process or not. If there is a foreground process which is still running, we check if it is still running or not. If the foreground process is still running we send a stop signal to it. Also in this part, we check if there are any zombie children. If there is a foreground process but it is not still running the boolean variable, which is declared for to keep tracking if there is a foreground process, set to false and signal has no effect at this point. If there is no foreground process then again the signal has no effect.

search

We handled the search inside a separated function that takes three parameters as “pwd” which holds the relative path of current working directory, “isRecursive" which will be 1 when “-r” option is given, otherwise it will be 0 and program will only run for current directory, and lastly “keyword” holds the user specified argument to search. If the program finds a folder and -r option is given, the sub-folder will be searched recursively while traversing inside the current directory. On the other hand, the program will automatically jump to the next file and check the extension of each file in order to search only eligible ones. For searching a single file, simply we have created a line counter and a buffer that contains each matched line.

bookmark

The bookmark built-in function is used to store and later run some of the users’ most used commands. There are several functionalities to bookmark, i.e. listing (-l), executing (-i), deletion (-d), ... In the beginning, we start by determining which functionality of the bookmark needs to be done. A command without -l, -i, and -d means adding that command to the list of existing bookmarked commands. We use a linked list data structure in order to keep track of bookmarks. Each node of the linked list is a struct which stores necessary information. If the

user wants to list the bookmarks, we simply loop over all of the linked list nodes and print their required information. In case of -i, we expect the user to enter the index of the bookmarked item they want to execute. Finally, if the user wants to delete a bookmarked item, we expect that the user will provide the index, and using that index we loop over the nodes and delete it. Upon successful deletion of the linked list node, we shift the indices of the subsequent nodes so we could keep the order.

exit

The exit built-in function is used to exit from the myshell when there are no background processes left. Therefore, our goal here is to loop over all of the nodes in the background process linked list, check their most recent status, and make sure they are the parent process' child. If there are background processes still running, we warn the user about it and simply do not exit from myshell.

I/O Redirection

For this part, first of all, we declare an integer variable called redirectionMode. To understand which mode the variable is in, we assign integer values from 1 to 6 to the variable. 1 for create/truncate mode (>), 2 for create/append mode (>>), 3 for input mode (<), 4 for create/truncate and input mode (< file.out >), 5 for error mode (2>), and finally 6 for pipe mode (|). While determining the mode according to the value entered by the user, we also equated the ">, <, >>, 2>" characters to the value null because of the call to the execv function. In addition, we filled in the outputFile and inputFile character arrays that we created at the beginning, according to the desired mode. Finally, then using the appropriate modes in the switch statements we redirect the input and output.

Bonus

In the bonus part, we have used a library called a termios.h to catch, up and down arrow keys. In order to keep previous user inputs we have created a doubly linked list which contains previous argument, next argument and input line. We have 2 variables that consist of the head and tail of the linked list. At the beginning of the program we are creating a new node for tail to hold the next input inside that node. Then, in each iteration we are copying user inputs from inputBuffer to tail's line and creating a new node for tail again. Briefly, we are adding a node to tail in each input and connecting the newly created node with the previous one.