OS Assignment-2

(Group-ID: 35)

Group Members:

S.No.	Name	Roll No.	Contribution
1	Debjit Banerji	2022146	• Researched on piping, file handling, system interrupt handling, exec functions and all related functions required for the assignment. • Wrote and debugged all the functions. • Added most of the comments in the program and most of the error handling code snippets. • Wrote the code for the bonus part (2). • Created and edited the explanation of the working of the SimpleShell.
2	Baljyot Singh Modi	2022133	• Researched on piping, file handling, exec functions and all related functions required for the assignment. • Wrote and debugged all the functions. • Added a few error handling code snippets and a few comments. • Wrote and debugged the code for the bonus part (1). • Debugged the code for the bonus part (2).

Working of SimpleShell:

The working of the SimpleShell can be explained in the following way:

1) Shell Loop Function:
    a. This function is called by the main method and is responsible for taking the commands as input from the user through the terminal like interface.

b. The program first checks whether it was able to read the command entered by the user properly. If yes, then the following steps are executed otherwise the program terminates with an error.
c. If the command was read properly, it checks whether the command is “fileinput”. If yes, then it redirects the program to the shell_loop2 function to execute all the commands written in the file commands.sh, otherwise executes the following lines of code.
d. The program then checks if the command entered is a valid command or not. If the command is valid then it formats and stores it in a 2D array for easy processing. If the command entered is found to be invalid, then the program asks for input from the user again.
e. The program then checks whether the command entered is “history”. If yes, then it prints the details of all the commands (like command name, start time of execution, execution duration, process ID) that have been entered at the terminal by the user until that point of time. Then, the history command is also saved in the history.txt along with its details. If the command entered was not “history” then the following lines of code are executed.
f. The program executes the entered command by redirecting the execution flow to the create_process_and_run function, and then after the execution of the command finishes, it stores all the details of the entered command in history.txt.
g. Then, if the command was executed properly, then the create_process_and_run function returns a value of 1 and the terminal asks for input from the user again. Otherwise, the create_process_and_run function returns 0 and the program exits.
h. The user can also prompt to press Ctrl-C to stop the program forcefully. In this case, the program catches the software interrupt and displays all the content of the history.txt (all the details of the all the commands executed until this point of time) before stopping the program.

2) <u>Shell_loop2 Function:</u>
   a. The shell_loop2 is called by the shell_loop function is the use has given the command “fileinput” to the terminal.
   b. The program first checks whether it was able to read the command in the commands.sh file properly. If yes, then the following steps are executed otherwise the program terminates with an error.
   c. The program then checks if the command entered is a valid command or not. If the command is valid then it formats and stores it in a 2D array for easy processing. If the command entered is found to be invalid, then the program asks for input from the user again.
   d. The program then checks whether the command entered is “history”. If yes, then it prints the details of all the commands (like command name, start time of execution, execution duration, process ID) that have been executed at the terminal by the user until that point of time. Then, the history command is also saved in the history.txt along with its details. If the command entered was not “history” then the following lines of code are executed.

e. The program executes the command pointed to by the file pointer in the file "commands.sh" by redirecting the execution flow to the create_process_and_run function, and then after the execution of the command finishes, it stores all the details of the entered command in history.txt.
f. Then, if the command was executed properly, then the create_process_and_run function returns a value of 1 and the terminal executes the next command in the file. Otherwise, the create_process_and_run function returns 0 and the program exits.
g. The user can also prompt to press Ctrl-C to stop the program forcefully. In this case, the program catches the software interrupt and displays all the content of the history.txt (all the details of the all the commands executed until this point of time) before stopping the program.
h. Once all the commands of the file have been executed the execution flow returns to the shell_loop function and now it asks for user input again.

3) History function:
   a. The history function puts all the command details stored in the character array "line" in the history.txt, repositions the file pointer to the starting of the file and prints the entire content of the history.txt on the terminal.
   b. It then empties the line variable to store more commands and returns to the shell_loop or shell_loop2 function.

4) Get_time Function:
   This function sets the time-zone, creates a time variable, stored the local time in the variable, and then converts the time stored in variable to a readable format and then finally stores it in the global variable "timeofexec".

5) Sys_call Handler Function:
   a. This function catches the Ctrl-C software interrupt, and executes the following lines of code in the function which includes printing the history in a formatted manner.

6) Split Function:
   a. This function firstly checks if the command entered is valid or not. If its valid, then it continues to execute the following lines of code, otherwise it returns 0 to the shell_loop or shell_loop2 function.
   b. The function then stores the command passed to the terminal in a 2D array which each column contains a word of the command and each row contains a command separated by pipes. If there are no pipes in the given command, the 2d array has only one row. The last element is set to be NULL, to execute properly using execvp function in the create_process_and_run function.
   c. The no of rows of the 2D array is also stored in a global variable called "rows" which is used in the create_process_and_run function.

7) Create_process_and_run Function:

a. It first stores the start time of execution of the command in a global variable called “start_time_of_exec” using clock_gettime inbuilt function.
b. Then, the function checks whether the entered command only has one row (which means the entered command does not use piping).
c. If the entered command does not use piping, then it enters the “if block”, creates and child process and executes the command using the execvp function in the child process.
d. Then, the parent process records the process ID of the child process in the global variable called “pid” and also records the end time of execution of the command in the global variable “end_time_of_exec” using clock_gettime inbuilt function, and then return to shell_loop or shell_loop2 function.
e. If the entered command uses pipes, then the function enters in the “else block”, initializes file descriptors for each pipe in the command, to be piped between the child processes, runs a for loop for the number of rows in the 2d array, pipes the file descriptor and creates a child process for every iteration of the loop.
f. Now, if the child process is the first piped command, then it does not direct the input of the process to the STDIN_FILENO, otherwise, it does.
g. After this, if the child process is the last piped command, then it does not direct the output of the process to the STDOUT_FILENO, otherwise, it does.
h. After these two steps, all the read and write ends of all the file descriptors are closed.
i. Then, the execvp executes the jth command and stores the output in the STDOUT_FILENO (except for the last command).
j. Then, the next command reads the output of the previous command from the STDIN_FILENO of the previous file descriptor (except for the first command) and executes its command and again stores it in the STDOUT_FILENO (except for the last command).
k. This process continues till the last piped command is reached, which read the input from the STDIN_FILENO of the previous file descriptor and prints the output to the terminal.
l. After, this entire process finishes, the parent process records the end time of execution in the global variable “end_time_of_exec” using clock_gettime inbuilt function, and returns 1 to the shell_loop or shell_loop2 function.

Commands that have not been handled:

- Features like navigating through previously entered commands using arrow keys.
- Built in commands involving “cd” command cannot be implemented with the “execvp” type of the “exec” function that we have used in our program.
- Commands like “fg”, “bg” and “jobs” cannot be used in the SimpleShell program.
- Complex shell operations like input/output redirection using ‘<’, ‘>’, ‘>>’ and process grouping (done using the ‘()’) have not been handled.

Bonus Parts:

1) Bonus Part 1:
   We have essentially added one more condition in the create_process_and_run function, where we have checked 2nd last string of each command line and if it is an '&' symbol, then inside the child process we have created another child of the child (grandchild). Then, we have immediately exited the child process, now the grandchild gets adopted by the mother of all processes, that is the "init" process, which runs the command (inside the grandchild) in the background, allowing us to start the next process.

2) Bonus Part 2:
   The code added for the bonus part has already been described above in the "Shell_loop2" function description above.

GitHub Repository Link:
https://github.com/baljyot25/CSE231_OS_35/tree/main/OS_Assignment_2