Simple Shell

This code implements a basic Unix shell that provides several key functionalities. It allows users to execute foreground and background commands. Both types of commands are handled separately. In case of background commands, the shell doesn't wait for the process to finish and moves on to accept new prompts.
The code also provides a feature to handle piping between commands. For this, it parses the command w.r.t '|' and then creates pipes and executes the processes by enabling the output of one command to be passed as input to another. The shell also maintains a history of the command(command, process id, execution start time, total duration of execution). It uses this history when the 'history' command is called or the shell is terminated. Once the shell terminates this history is cleared. The shell also handles graceful termination by ensuring that the command history is displayed and the memory allocated for storing command data is freed before the shell exits.

The following are the components of the code:

1. store_command

This function adds a command to the command history, along with its PID, start time, and execution duration. It is called whenever a command (foreground or background) is executed. It stores the command details in a linked list (`head` and tail point to this list).

2. display_shellHistory

This function displays the command history with full details, including command name, PID, start time, and execution duration. It is called when the shell terminates (e.g., when the user presses Ctrl+C).

3. showHistory

It displays a simple list of previously entered commands. It is called when the user types the `history` command to see a list of command names that were executed.

4. free_shellHistory

This function frees the memory allocated to store the command history. It is called at shell termination to clean up memory and prevent memory leaks.

5. add_bgProcess

It adds a background process to a linked list for tracking. It stores the process's PID, start time and command string. It is called when a command is executed in the background (with &), enabling the shell to track its completion.

6. check_bgProcess

It checks whether any background processes have finished. If a background process has completed, its execution time is calculated and the command history is updated. It is called repeatedly in the shell loop to monitor the status of background processes using waitpid() with the WNOHANG option.

7. handle_SIGINT

It handles the Ctrl+C (SIGINT) signal to gracefully terminate the shell. It displays the command history and frees the memory allocated for the command history before terminating. It is installed as a signal handler and automatically called when the user presses Ctrl+C.

8. read_user_input

It reads a line of input from the user. Allocates memory to store the input and returns the input string. It is called in the shell loop to capture the command entered by the user.

9. Shell_loop

It is the main loop of the shell. It repeatedly reads user input, checks for completed background processes, parses commands, and executes them. Also handles history

commands and triggers process creation for other commands. It is called from main() to keep the shell running until the user terminates it.

10. parse_for_piping

This function parses the input string to check if there are multiple commands connected by pipes (|). Returns a set of individual commands and counts the number of pipes. It is called when a command contains pipes, enabling the shell to process each command individually.

11. Parse_command

It breaks a command into its arguments and identifies if the command should run in the background (by checking for &). Fills the `args` array with command arguments. It is called to split the input string into individual arguments, which are then passed to `execvp()` for command execution.

12. launch

It launches a command by calling create_process_and_run, determining whether the command should run in the background or foreground.It is called after parsing the command to trigger the process creation and execution.

13. create_process_and_run

It forks a new process to execute the given command. If the command is a background process, it adds it to the background process list; if it's a foreground process, it waits for it to complete and records the execution time.It is called by launch to handle process creation, command execution, and managing foreground/background processes.

14. piping

It handles commands connected by pipes. It forks multiple processes and links their input/output through pipes, enabling the output of one command to become the input for

the next. It is called when commands are separated by "|".It also manages the creation of pipes and process synchronization for piped commands.

The code implements three types of commands, i.e., command without '&'(foreground process), command with '&'(background process) and ctrl+c in the following ways:

1. Normal Commands Without & (Foreground Commands):

- The shell reads the command from the user.
- The parse_command function splits the command into arguments.
- The launch function calls create_process_and_run to fork a new process.
- In create_process_and_run:
    1. The process is forked.
    2. The parent waits for the child to complete using the waitpid function.
    3. The command is then stored in the history.

2. Commands With & (Background Commands):

- The shell reads the command from the user.
- The parse_command function detects the & symbol and marks it as a background command.
- The launch function calls create_process_and_run to fork a new process.
- In create_process_and_run:
    1. The process is forked.
    2. The parent does not wait for the child (background).
    3. The start time is recorded and the command is stored in history.
    4. The background process is added to the background process list using the add_bgProcess function.
- The shell returns to the shell_loop immediately, without waiting for the background process to finish.
- In subsequent iterations, check_bgProcess is called to check if the background process has finished, and it updates the history with the actual duration.

3. Termination (`Ctrl+C`):

- When the user presses Ctrl+C, the handle_SIGINT function is triggered.

- The command history is displayed, showing each command with its PID, start time, and execution duration.
- The memory allocated for the command history is freed using free_shellHistory.
- The shell terminates.

Following are the commands that will not be supported by our simple shell implementation:
1. cd(Change Directory): This command changes the shell's working directory. But this command affects the shell process itself. Our SimpleShell runs commands in child processes, because of which ‘cd’ would not have any effect.

2. source: This command executes a script in the current shell process which means that any variables, functions, or changes made by the script persist in the current shell environment. But our implementation creates/forks a new process for each command which would mean that any change to the environment would be local to the child process and won’t persist.

3. History for previous sessions(Ctrl+R): Our current implementation allows the current shell session’s history to be stored and displayed. It won’t be able to show commands from previous sessions due to the absence of persistent storage.

4. vi: The ‘vi’ command or any other text editor-related commands demand full access to the terminal as they have to handle screen updates, cursor actions, and other interactive features. Our current implementation doesn’t explicitly manage the terminal and hence ‘vi’ won’t be able to have full access to the terminal.

5. alias: This command is a built-in shell command and is used to create shortcuts/snippets for commands. Our current implementation doesn’t support the built-in commands and hence ‘alias’ command will not be supported.

6. trap: This command is used to catch and handle signals in the shell script. Our current implementation doesn’t have advanced management of signals because of which the ‘trap’ command wouldn’t be supported.

Contributions

Both the team members had equal contribution in completing this assignment which made the whole process easy and efficient. Individual contributions of the team members are listed below:-

Abhinav Kashyap(2023022): Implementation of main Shell_loop, piping, create_process_and_run, parse_for_piping, add_bgprocess, check_bgprocess, show_history, handle_SIGINT and debugging.

Harsh Sharma(2023233): Implementation of main Shell_loop, piping, create_process_and_run, parse_command, reading user input, display_shellhistory, store_command, free_shellhistory and debugging.

Link to private GitHub repository: https://github.com/Abhinav0821/SimpleUnixShell

Sources for additional information:

1.https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-waitpid-wait-specific-child-process-end

2. https://www.gnu.org/software/libc/manual/html_node/Process-Completion.html

3.https://www.ibm.com/docs/en/zvm/7.3?topic=descriptions-waitpid-wait-specific-child-process-end

4. Lecture 6

5. Lecture 7