TO RUN CODE :

In main.c file

1) make
2) ./outputProgram

Overview of the code :

1. `**saveToHistoryFile**`: This function saves the details of a command, such as the command itself, its process ID (PID), start time, and duration, into a file named "history.txt". It is used to keep a history of executed commands.

2. `**printHistoryFromFile**`: This function reads and prints the command history from the "history.txt" file when the user enters the "history" command.

3. `**PipeCommands**`: This function handles the execution of commands separated by pipes ('|'). It splits the input into separate commands, creates pipes for communication between them, and executes each command in its own child process. The output of one command is connected to the input of the next command through the pipe.

4. `**StringToArray**`: This function splits a string (command) into an array of strings (command and its arguments) based on whitespace. It is used for parsing user input into an array of command arguments.

5. `**ExecuteCommands**`: This function executes a single command, either one entered by the user or a command within a piped sequence. It forks a child process to run the command and waits for the child process to complete. If the command contains the '&' character, it can run the command in the background.

6. `**trim**`: This function trims leading and trailing whitespace from a string, used for cleaning up user input.

7. `**CommandWithAnd**`: This function handles commands containing the '&' character. It splits the input into separate commands and executes each command in its own child process, allowing them to run concurrently.

8. `**FileStringToArray**`: This function is similar to `StringToArray`, but it reads a command from a file and parses it into an array of strings.

9. `**read_user_input**`: This function reads a line of user input from the terminal.

10. `**ExecuteFileCommands**`: This function executes a series of commands read from a file. It forks a child process for each command, waits for the child processes to complete, and records command details in the history.

11. `**launch**`: This function initiates the execution of a command. It determines whether the command contains pipes or should be executed in the background and records command details in the history.

12. `**ClearFile**`: This function clears the command history by opening the "history.txt" file and truncating its content.

13. `**infinite_input**`: This function continuously prompts the user for input and executes commands until the user presses Ctrl+Z. It also handles the execution of the "clear" command to clear the history.

14. `**ExitTheShell**`: This function is a signal handler for Ctrl+C (SIGINT). It clears the history and exits the shell when Ctrl+C is pressed.

15. `**DoBonus**`: This function executes a series of commands from a file named "bash.sh." It reads commands from the file, parses them, and executes them sequentially.

16. `**main**`: The main function is the entry point of the program. It processes commands from the user or runs commands from the "bash.sh" file based on user input. It also sets up the signal handler for Ctrl+C (SIGINT) and enters the main shell loop.

Overall, this program provides a basic shell-like interface with some additional features, such as command history and the ability to execute commands from a file.

Commands which cannot run :

Here are logical reasons for why the provided commands might not work in your shell:

1. `**cd <directory_name>**`
   - **Reason**: The `cd` command is a shell built-in command, and executing it via `fork()` and `exec()` functions, like external commands, will not change the directory of the parent shell. You need a dedicated implementation in your shell to change its own working directory using `chdir()` function.

2. `**alias**`
   - **Reason**: The provided shell doesn't have built-in functionality to create and manage aliases. Implementing aliases would require maintaining a mapping between alias names and their corresponding command strings.

3. `**source <file_name>**`
   - Reason: The `source` command typically executes commands in the current shell environment. Your shell lacks a mechanism to read and execute commands from a given file within the current shell context.

4. `**export VAR=value**`
   - **Reason**: Your shell doesn't have an implementation for environment variable management. To support this, you would need functionality to set environment variables, which can be done using functions like `setenv()`.

5. `**unset VAR**`
   - **Reason**: Similar to the `export` command, your shell doesn't provide functionality to unset or delete environment variables.

6. `**pushd <directory_name>` & `popd**`
   - **Reason:** Both `pushd` and `popd` require maintaining a directory stack in the shell, which your current shell doesn't seem to support.

8. `**jobs**`
   - **Reason**: The `jobs` command requires maintaining a list of background jobs. Your shell currently lacks the infrastructure to keep track of and display background processes.

9. `**fg %job_number**`
   - **Reason**: Your shell does not have the capability to manage background and foreground processes in a job control fashion. Bringing a background job to the foreground requires more advanced process group management.

10. `**bg %job_number**`
   - **Reason:** Similar to `fg`, sending a job to the background and managing its execution would require an advanced job control mechanism which is not present in your shell.

In summary, for your shell to support these commands, you would need to incorporate more advanced features and handle many edge cases associated with process and job control, directory and environment variable management, and shell built-ins.

Contribution Report

Contributions:

1. **Anish**:

- **Contribution Percentage: 50%**

Details:

➔ Made a basic structure of the code, laying down the logic and components.
➔ Focused on the code implementation, from defining global variables to drifting the initial versions of the primary functions.

2. Arav:

- **Contribution Percentage: 50%**

Details:

➔ Worked intensively on error analysis.
➔ Dedicated efforts towards the optimization of the code, making it efficient, and more user friendly.

Final Breakdown:

- **Anish**: 50%
- **Arav**: 50%

GITHUB : https://github.com/ianishdev/OS-Assignment.git