POSIX SHELL

Team: We_4

Introduction:-

POSIX Shell is a command line shell for computer operating system which was introduced by IEEE Computer Society. POSIX stands for ***Portable Operating System Interface***.
POSIX defines the application programming interface (API), along with Unix command line shells and utility interfaces. This ensure software compatibility with flavors of Unix and other operating systems.
The POSIX shell is implemented for many UNIX like operating systems. The POSIX standard is designed to be used by both application programmers and system administrators.
In Linux, a shell offers an interface for a Unix system that allows you to execute commands or utilities more easily. A shell collects an input from a user and executes a program according to that input.
In this project we have created a working POSIX compatible shell with a subset of feature support of our default shell.

Description:-

To develop a working POSIX compatible shell.
Following features have been implemented:-

- Basic shell commands (ls,echo,touch,mkdir,grep,pwd,cd,cat,head,tail,chmod,exit,history,clear, cp)
- Tab autocompletion
- Generic piping support (For any number of pipes)
  Example: cat main.cpp | head -10 | tail -4 | sort
- History : Stores all the valid commands entered by user
- Alarm: (Example: alarm k message : This command will remind about the message after k second)
- Maintain a configuration file (.bashrc) which our program reads on startup and sets the environment accordingly.
- This file contains alias and default applications that we use to open any file.
- Association of “~” with the HOME variable.
- Prompt look via PS1 is handled.
- Export: This command exports the variable locally for the current session.
  Example: export college=IIIT

- I/O redirections (IO redirection with '>>' and '>' will be done for one source and one destination only. Example: cat main.cpp > myfile.txt)

Work Distribution:-

1) **Ujjwal Prakash**

- Parsing user input and calling appropriate functions for its execution
- Implementing IO redirection with '>>' and '>'
- Changing prompt for normal and root user
- Handle support for '$$' and '$?'
- Maintained a configuration file ".bashrc" which handles setup of all the environment variables on startup
- Did the implementation of recording as script command, namely 'record start' and 'record stop'
- Did the integration of the project as a whole by bundling up all the code written by the other team members.

2) **Priyank Mahour**

- History buffer - searchable via - **TRIE** . This is a list of all commands executed ever.
- The List of History is limited by the **HISTSIZE** variable in .bashrc file. Invoked as **history.**
- **TRIE - for tab completion is implemented**. All the directories which are in mentioned in **PATH** variable.
- Then we populate the TRIE by scanning all the directories mentioned in the **Environment PATH variable .**
- Partially helped to implement "record start" and "record stop" using dup2().

3) **Prashant Kumar**

- Implemented **Alias** functionality.
- Setup a generic **piping** approach.
- Setup all environment variable maintained by **.bashrc** file
- Handled look of prompt via PS1

4) **Rishabh Gupta**

- An **alarm** feature in shell is implemented.
- Did the implementation of **open**.
- Implemented exporting variables with **'export'** command.
- Did the implementation of **'setenv'** and **'unsetenv'** command

Solution Approach:-

- We've parsed the input in two ways, namely text input without any special character and text input with special characters.
- For generic command which are given through text based input we are simply executing system commands in the /bin folder using fork and execl.
- For commands, with special characters like pipe (|) or redirection (>> or >) we are using different functions.
- Trie has been used to implement history and tab auto completion.
- For record start and stop, we have used dup2() which duplicates an open file descriptor.

Problems Faced & Shortcomings:-

- While implementing 'history' command we faced some challenges in maintaining previous session history. Ultimately we resolved it by loading the previous session history in trie data structure with the help of a global count and finally saving the context while quitting the session.
- While implementing the piping functionality, we faced some challenges in storing the intermediate result of one operation before forwarding it to the next operation. This was handled using dup2().
- While implementing the alarm functionality, placing the cursor after the message was displayed has been a challenge.
- Implementation of bg and fg jobs has been a challenge for us.

Learnings:-

- Got practical understanding of fork() and exec() system calls.
- Got information about various signal handling commands.
- Learnt about dup2() and pipe() calls.
- Got to know the difference between raw and canonical mode.
- Usage of some default environment variables which LINUX uses.
- Got an intuition of trie data structure.

Conclusion:-

- While doing this project, we got a great understanding of the various system calls() and method through which we get a lot of flexibility to make our own commands.
- The beauty of LINUX is that it is very close to developers through which they can break into it to come up with their own custom commands.

Github Link:-

https://github.com/WeFourIIIT/POSIX-Shell
