# POSIX SHELL

Advanced Operating Systems

Project Report

**Team number:** 27

**Team name:** sQuad

**Members:**

1. Sarthak Verma *(2021201002)*
2. Aditya Sharma *(2021201003*)
3. Ashish Rai *(2021201060)*
4. Kishan Faladu *(2021202003)*

# INTRODUCTION

The ‘shell’ is a command line interpreter that can be used to interact with the operating system.

POSIX (Portable Operating System Interface) is a family of standards defined by IEEE to allow compatibility of a piece of software or program with various operating systems. In simpler terms, a system following these standards is ‘portable’.

The POSIX shell is a command line interpreter that can work on all POSIX compliant operating systems, including all variants of unix.

# PROBLEM

The given problem is to construct a POSIX compliant shell that incorporates a subset of the features found in a real POSIX shell. These include:

1. Basic working: Support for shell commands like ls, echo, touch, mkdir, grep, pwd, cd, cat, head, tail, chmod, exit, history, clear and cp.
2. IO Redirection using ‘>’ or ‘>>’ for one source and destination only
3. Usage of ‘|’ symbol to implement the pipe feature. At least 3 commands can be piped successfully.
4. ‘&’ symbol at the end of a command to run it in the background, and the ability to use ‘fg’ command to bring a specific process to foreground.
5. Storing entered commands in a history, also searchable via trie.
6. Recording entered commands into an output file.
7. Alarm feature for specific date/time.
8. Aliases.
9. Autocompletion of commands using trie.
10. Support for environment variables.

# SOLUTION APPROACH

This project has been approached by breaking it into two parts:

1. A normal text editor

2. Passing/inputting commands, tokenizing them, setting their

precedences and executing them

For the text editor, a simple text editor has been made where users can enter and execute commands. It will have some basic

commands of its own like Clearline function which will clear the

screen and hotkeys to alter text on the screen.

For input command execution, alias resolution has been done first followed by parsing, command execution and output generation. Fork requests have been handled by keeping track of background and foreground processes and halting the parent process (foreground) until the completion of the child process (so the child process is now treated as foreground).

The commands themselves are executed using the ‘execvp’ system call to get the pre-implemented commands from /bin directory.

Dup2, fork and pipe system calls have been used to implement the piping feature. A new child process is made for general execution, and within that the dup2 call is used to redirect outputs of piped segments to a pipe, followed by another dup2 for next segment to read inputs from the pipe instead of stdin.

For the record command, a file pointer is used which will keep track of all changes and write them into an output file while the command is on.

We will be maintaining 3 tries which will be store the following:

1. Paths to directories/files (Trie1)

2. All executable commands (Trie2)

For history commands, they will be written to a .history file on every execution up to a cap defined by HISTSIZE in .myrc config file.

For auto-complete functionality in paths/commands, “command_trie” and “directory_trie” respectively will be used where users will be able to get

different autocomplete suggestions(closest to farthest) by pressing the `Tab` key. If there is only one suggestion, it will be autocomplete in place.

For the alarm feature, the ‘alarm’ POSIX compliant function will be used. A separate background process will be made for each alarm that ends when the alarm signal goes off. When the parent detects that the alarm process has ended, it will generate the notification.

A “.myrc” file will be maintained which will have config settings for the shell such as history trie size and mapping of extensions to the default application to be used for the same and integrate it with the shell. A .filemaprc file will be used to keep track of mappings for specific extensions.

# WORK DISTRIBUTION

| MEMBER | FEATURES |
| --- | --- |
| Sarthak Verma | 1. Command execution<br>2. Piping<br>3. Foreground/Background<br>4. Process data structures management<br>5. Command parsing |
| Aditya Sharma | 1. Text editor/Command input<br>2. Autocompletion<br>3. Command parsing<br>4. Environment Variables |
| Ashish Rai | 1. Aliasing<br>2. History<br>3. Record<br>4. Export<br>5. Environment Variables |
| Kishan Faladu | 1. Alarm<br>2. Documentation<br>3. Environment Variables<br>4. Command parsing<br>5. Command execution |

# PROBLEMS FACED AND SHORTCOMINGS

## COMMAND EXECUTION

Understanding pipes was a difficult part, especially how the iteration could be designed to continuously send the output of the previous piped segment of the command to the next.

Not ending the child process appropriately often leads to duplicate instances of the same command being executed.

How the data structures for foreground/background processes could be maintained was a bit tricky to figure out in the beginning, since global variables are not shared amongst processes. So adding/removing entries required trial and error.

Some commands required manual implementation and thus their own error handling as well, which took up some time.

During file execution, we had to define a separate .rc file to store its appropriate variables, since parsing was found to be very messy to implement.

## TEXT EDITOR

This part was particularly interesting and at the same time vital for the overall functioning of the shell. The only way for users to interact would be through this , so presentation was extremely important.

This included using non-canonical mode to our advantage and giving users basic interaction through text. At the same time as

rest of our team would be using the functions defined in the text editor, The helper functions had to be easy to understand and versatile.

The Challenge here was making versatile helper functions and loading appropriate information on the terminal in a presentable manner.

## AUTOCOMPLETION

This feature, although very minor, is extremely useful when used in the right way. Auto Completion not only saves time but also gives users a basic idea of the commands and directories available to them.

The autocomplete feature in Bash is very versatile , so it was important to us to make it just as much if not more versatile.

Our autocomplete function not only distinguishes between commands and directories but also considers different operations like pipes when suggesting the users. All this is done through our custom trie implementation.

Also based on the number of suggestions available it automatically changes the commands or prints suggestions.

This function was a bit tedious to implement because of the versatility offered but turned out to be a very fun feature while using our terminal and thus deserves a highlight.

One minor shortcoming would be the fact that directory auto completion only works for files in the current directory, and not for any directories in a partially written path.

## ENVIRONMENT VARIABLES

Storing environment variables is a core shell functionality, but still it was a relatively easier part to implement. It requires basic knowledge of file reading and writing and also string parsing.

We store all the variables declared during the shell session in the ".myrc" file and load them as required.

In particular, the "PS1" variable is extracted from the ".myrc" file every time and thus is editable as the user requires.

# ALARM

For alarms, I had to understand how to extract the current system time and get the difference between provided time and current system time and setting alarms accordingly. I handled it by making a structure of my own and inputting the string from user according to given format (DD/MM/YYYY::HH:MM:SS::Message) and storing them into respective variables and extracting the difference between current system time and provided alarm time in seconds.

Another issue was  to how to check if an alarm previously set would go off at session start, which was handled by calculated difference in seconds after reading alarms from file(alarms.txt) and calculating seconds difference, if it's negative we directly print the message otherwise we initiate an alarm and push it into alarm map against process id.

Using the alarm() function was a good way to set alarms but the problem was that only one SIGALRM signal can be present for a process at a time, so if we tried to set multiple alarms at the same time, the newer alarm call would replace the previous one already present in the system. We tackled this by using the fork command and generating a new process for each alarm which will keep running in the background and end when it returns the alarm signal which will then trigger the print function for its corresponding message.

# LEARNINGS

The biggest takeaways from this project were some basics regarding POSIX compliance, forking and parent/child processes, as well as the structure and working of a POSIX compliant shell.

Usage of signals also helped us learn about some useful signals and when they are generated, as well as how we can manage information regarding signals. Pipes also helped us build a keener awareness of how file descriptors work and how we can redirect I/O between exec() calls (and processes in the future if need be).

We also got to put into practice how we can use the non-canonical input mode and provide more versatile input options to the user. Thus we also provide an extensive feature set to the user through this mode.

# CONCLUSION

Using POSIX compliant functions and system calls, a POSIX compliant shell with required features has been successfully created.