Indian Institute of Technology Bombay

CS 744 Autumn 2022 Semester, Project

Report On

Implementation of UNIX Shell

Submitted by: Shamik Kumar De(22m0822)

1 Introduction

Overview of UNIX Shells: A UNIX shell is a command-line interpreter that provides a user interface to the operating system's services. It allows users to execute commands, manage files, and control processes.

Project Objective: The primary objective of this project was to develop a simple UNIX shell in C++ that supports essential shell functionalities, including command execution in both foreground and background, handling signals such as SIGINT, and managing process control.

Learning Outcomes: This project provided practical experience with system calls, process management, and signal handling in UNIX-like systems. It enhanced the understanding of how shells operate and interact with the kernel.

2 Design and Implementation

System Architecture: The shell operates in an infinite loop, waiting for user input. Upon receiving input, it tokenizes the command, determines whether the command should be run in the foreground or background, and then creates a child process to execute the command. The parent process either waits for the child to finish (foreground) or continues accepting new commands (background).

Command Parsing: The `tokenize()` function splits the user input into individual tokens based on spaces, tabs, and newlines. These tokens represent the command and its arguments, which are later used in process creation.

Foreground and Background Execution: The shell differentiates between foreground and background commands using the `checkAmpersand()` function. If an ampersand (&) is found, the command is executed in the background by forking a child process without the parent waiting for it to finish. Otherwise, the command runs in the foreground, and the parent process waits for the child to complete.

Signal Handling: The shell uses the `signal()` function to handle SIGINT (Ctrl+C) for terminating foreground processes. The `sig_handler()` function is triggered when SIGINT is received, killing the currently running foreground process.

3 Key Features

Process Control: The shell allows for both foreground and background process execution. Foreground processes block the shell until they finish, while background processes allow the shell to continue accepting new commands. Background processes are tracked in an array (`procs[]`), and the shell checks for any finished background processes in each loop iteration.

Signal Handling: The SIGINT signal handling ensures that the shell remains responsive and user-friendly by allowing users to terminate foreground processes with

Ctrl+C. This is a crucial feature for any interactive shell, as it provides control over running processes without requiring manual intervention.

Error Handling and Edge Cases: The shell checks for invalid commands using `execvp()`, and if the execution fails, it prints an error message. Additionally, the shell handles cases where no command is entered, or an incorrect command is used with built-in commands like `cd` or `exit`.

4 Challenges and Solutions

Command Execution and System Calls: One challenge was ensuring the shell could correctly execute a wide range of UNIX commands using `execvp()`. The solution involved careful parsing of user input and proper handling of arguments passed to system calls.

Signal Handling: Implementing robust signal handling required ensuring that SIGINT only affected the intended process (the current foreground process) and did not cause the shell itself to terminate. This was addressed by associating the signal handler only with the foreground child process.

Memory Management: The shell dynamically allocates memory for storing command tokens. Proper memory management was crucial to avoid leaks, especially in a long-running shell session. The solution involved freeing allocated memory for tokens after each command execution.

5 Conclusion

Project Summary: The UNIX shell developed in this project successfully implemented essential shell functionalities, including foreground and background process management, command execution, and signal handling. The shell provides a simplified but functional interface for interacting with the operating system.

Reflections: This project provided deep insights into how UNIX shells operate and the underlying system calls that facilitate command execution and process management. It also enhanced the understanding of signal handling in a concurrent environment.

Future Work: Future enhancements could include adding support for advanced features such as piping (`|`), redirection (`>`, `<`), command history, and more sophisticated error handling. These additions would bring the shell closer to a fully functional UNIX shell.