Design and Implementation of a Custom UNIX Shell

Project Team:
Vamsi G D N Riya Gupta
Naisha Dekate Arpan Tiwari

Mentored by: **Manikanta**
Department of Computer Science

1 Project Abstract

This report details the architectural design and technical implementation of a modular UNIX shell. The project explores core Operating System concepts such as process lifecycle management, signal handling, and lexical analysis. The primary goal was to develop a resilient environment that bridges the gap between user input and Kernel-level system calls, emphasizing modularity for scalability and ease of debugging.

2 Milestone 1: Foundational Shell Engine

Milestone 1 focused on constructing the "Heart" of the shell—the infrastructure required to facilitate seamless communication between the user and the operating system.

2.1 Sub-Topic 1.1: The REPL Architecture and UI

The shell operates on a robust Read-Eval-Print Loop (REPL). This sub-task involved creating a stable `main.c` engine designed for high uptime. Key features include:

- **Dynamic Prompting:** A customizable interface that awaits user commands.
- **Buffer Management:** Utilization of `getline()` to handle variable-length inputs, preventing buffer overflow vulnerabilities.
- **EOF Handling:** Detecting the `-1` return value from input streams to handle **Ctrl+D** logouts without process hanging.

2.2 Sub-Topic 1.2: Advanced Tokenization

The parser transforms raw character streams into a structured argument vector (`argv`). By implementing a custom `parser.c`, the team ensured that the shell can handle multiple delimiters (spaces, tabs, and newlines) while maintaining null-termination for system compatibility.

2.3 Sub-Topic 1.3: Process Isolation and Execution

The executor handles the critical transition from shell logic to system execution. The team implemented a "Fork-Exec-Wait" sequence to ensure process isolation:

a) **Process Spawning:** Using `fork()` to create a sandbox for command execution.

b) **Binary Execution:** Using `execvp()` to search the system `PATH` and execute binaries.

c) **Parent Synchronization:** Using `waitpid()` to prevent zombie processes and ensure the shell waits for command completion.

2.4 Sub-Topic 1.4: Signal Resilience

Under the guidance of our mentor, Manikanta, the team implemented signal trapping. By redefining `SIGINT`, the shell avoids termination upon receiving **Ctrl+C**, allowing the shell to remain active while only interrupting the foreground child process.

3 Design Methodology

The project employs a **Modular Command Pattern**. Each specific shell utility (like `ls`, `cat`, or `stat`) is decoupled from the core engine. This design choice ensures that errors in a specific command file (e.g., `cmd_ls.c`) do not affect the stability of the main executor, allowing for rapid iterative development.

4 Conclusion

The completion of Milestone 1 provides a robust framework for command execution. This foundation demonstrates the team's ability to manage low-level system calls and process states, setting the stage for advanced features in the subsequent project phases.