Implementation of a Simple Shell in C

Introduction

In this assignment, I implemented a simple command-line shell using the C programming language. The shell is capable of executing basic Linux commands, handling background processes, and managing the execution of multiple commands in both serial and parallel modes. The assignment was divided into several parts, each focusing on a specific aspect of shell functionality, including process management, signal handling, and user command parsing.

Implementation Details

Part A: Simple Shell

The initial part of the implementation focused on creating a basic shell that can execute simple Linux commands such as ls, cat, echo, and sleep. The shell uses the fork(), exec(), and wait() system calls to create child processes, execute commands, and reap dead processes. The shell continuously accepts user inputs, tokenizes the input string, and executes the corresponding command.

Key aspects of this implementation include:

Command Parsing: The input string is tokenized into an array of strings, where the first element is the command and the subsequent elements are its arguments.

Command Execution: The shell uses fork() to create a new child process and exec() to execute the command in the child process. The parent process waits for the child to finish using wait().

Error Handling: The shell gracefully handles errors such as incorrect commands or arguments by notifying the user and continuing to prompt for the next command.

Part B: Background Execution

In the next part, the shell was extended to support background execution of commands. If a command is followed by an ampersand (&), the shell executes it in the background, allowing the user to continue interacting with the shell without waiting for the command to complete.

Key aspects of this implementation include:

Background Process Management: The shell keeps track of background processes and periodically checks if they have terminated. When a background process finishes, the shell prints a message indicating its completion.

Foreground and Background Process Reaping: The shell ensures that all child processes, whether foreground or background, are reaped to avoid zombie processes.

Part C: Exit Command

An exit command was implemented to terminate the shell. Upon receiving this command, the shell terminates all running background processes, cleans up any allocated resources, and exits the infinite loop to terminate the shell process.

Part D: Handling Ctrl+C Signal

The shell was further enhanced to handle the SIGINT signal (Ctrl+C). Instead of terminating the shell, the signal is caught and relayed to the foreground process, allowing the shell to continue running while terminating only the active command.

Key aspects of this implementation include:

Signal Handling: A custom signal handler was implemented using signal() to catch SIGINT and handle it appropriately.

Process Group Management: The shell manages process groups to ensure that only the foreground process is affected by the SIGINT signal, while background processes remain unaffected.

Part E: Serial and Parallel Foreground Execution

Finally, the shell was extended to support the execution of multiple commands in both serial and parallel modes. Commands separated by && are executed sequentially, while commands separated by &&& are executed simultaneously.

Key aspects of this implementation include:

Command Chaining: The shell parses input strings with multiple commands and executes them in the specified order, either serially or in parallel.

Error Handling in Serial Execution: If a command in a serial chain fails, the shell continues to execute the remaining commands.

Parallel Execution Management: The shell ensures that all parallel processes are correctly managed and reaped before returning to the command prompt.

Conclusion

This assignment provided a deep understanding of process management, signal handling, and user input parsing in Linux. By implementing a simple shell, I gained practical experience with system calls such as fork(), exec(), wait(), and kill(). The shell was designed to handle both foreground and background processes, manage multiple commands, and gracefully handle errors and signals. This implementation forms the foundation for more complex shell functionalities in the future.