Course Project Report:

Implementation of Linux Shell (C | Shell | System Calls) (Aug'22-Dec'22)

Course: CS 744: Design and Engineering of Computing Systems
Instructor: Prof. Mythili Vutukuru

Project Overview

This project involved the design and implementation of a basic UNIX shell terminal in C, capable of executing user commands. The shell mimics the functionality of a real shell by supporting various built-in commands, **handling foreground and background processes**, and managing user inputs efficiently. The shell is built using system calls like **fork, exec, exit, and wait,** which are fundamental to process management in UNIX-like operating systems.

Key Features

1. **Command Execution**:
    - o The shell can execute a variety of user commands like cd, echo, cat, sleep, and chdir. These commands are handled by invoking the appropriate system calls within the shell.
    - o For example, the cd command changes the current working directory using the chdir() system call.
2. **Process Management**:
    - o The shell supports both foreground and background process execution.
    - o Foreground processes are executed such that the shell waits for their completion before accepting new commands.
    - o Background processes are handled differently, allowing the shell to continue accepting new commands while the background processes execute concurrently. The & operator is used to denote background execution.
3. **Signal Handling**:
    - o The shell can handle SIGINT (generated by pressing Ctrl + C), which is typically used to terminate foreground processes. The signal handling function ensures that only the foreground process is terminated, while background processes continue running.
    - o A custom signal handler is implemented to manage the shell's exit process (SIGUSR1), ensuring that all background processes are properly terminated when the shell exits.

4. **Memory Management**:
   - The shell allocates memory dynamically for storing and processing user inputs and commands.
   - Proper memory deallocation is performed after each command is executed to avoid memory leaks.
5. **Job Control**:
   - The shell maintains an array of background processes and reaps them upon completion to prevent zombie processes.

Detailed Code Explanation

1. **Tokenization of User Input**:
   - The tokenize function is responsible for splitting the user input string into individual command tokens. This is achieved by iterating over the input string and breaking it down based on spaces and newline characters.
2. **Change Directory (cd) Implementation**:
   - The cd command is implemented using the chdir() system call, which changes the current working directory to the one specified by the user.
   - If the command fails (e.g., due to an invalid directory), an error message is displayed.
3. **Executing Commands**:
   - The shell uses the fork() system call to create a new child process for command execution. The child process then uses execvp() to replace its image with the program specified by the user command.
   - If the command ends with &, the shell sets the process to run in the background.
4. **Handling Background Processes**:
   - Background processes are managed using process groups, allowing the shell to distinguish between foreground and background tasks.
   - The shell reaps terminated background processes to clean up resources and prevent the accumulation of zombie processes.
5. **Signal Handling**:
   - The SIGINT_Handler function is used to handle SIGINT signals, terminating only the foreground process.

    - The handle_sigusr1 function deals with custom exit signals, ensuring that all background processes are terminated before the shell itself exits.
    -

Challenges and Learning Outcomes

- **Process Management**: Implementing process management using fork, exec, and wait system calls was a key learning experience. Understanding how UNIX processes are created and managed provided deeper insights into operating systems.
- **Signal Handling**: Managing signals like SIGINT and custom signals required careful consideration to ensure that the shell behaved correctly in different scenarios, such as terminating only the intended processes and cleaning up resources.
- **Memory Management**: Handling dynamic memory allocation for user inputs and ensuring proper deallocation was a critical aspect of the project. This helped reinforce the importance of memory management in C programming.

Conclusion

This project successfully demonstrated the principles of process management, system calls, and basic UNIX shell functionality. By building a custom shell, the fundamental concepts of operating systems, such as process control, signal handling, and inter-process communication, were effectively explored and applied.

This project was completed as part of the course **CS 744: Design and Engineering of Computing Systems** under the guidance of **Prof. Mythili Vutukuru** during the August-December 2022 semester.