OPERATING SYSTEMS
UCS303 - 2323SUMSUM

C-SHELL

Submitted to **Mr. Shashank Singh**

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be Univerrsity)

Submitted by -

Aryan Tyagi	-	**102117184**
Arihant Chaudhary	-	**102117139**

TABLE OF CONTENT

Abstract

This project involves the development of a simple Unix-like shell, named "osh", which demonstrates key features and functionalities of a command-line interface commonly found in operating systems. The shell provides a robust environment for executing user commands, managing process control, and handling job scheduling with functionalities similar to those of established shells like Bash.

The shell supports various built-in commands for file operations, process management, and system information retrieval, including my_ls, my_create, my_delete, my_ps, my_kill, my_meminfo, and my_sysinfo. It also includes support for traditional commands such as cd for changing directories and exit for terminating the shell session. Notably, the shell implements job control features, allowing users to manage background and foreground processes effectively. Users can list active jobs, bring jobs to the foreground using fg, and continue jobs in the background using bg.

One of the key aspects of this shell is its job control mechanism. It maintains a list of active jobs, each associated with a unique process ID (PID) and command. The shell can add jobs to this list, remove completed jobs, and provide a listing of all currently active jobs. Users can interact with these jobs by bringing them to the foreground or continuing them in the background, using the appropriate commands.

The project also includes history management, which allows users to track and recall previously executed commands, enhancing the usability and efficiency of the shell. The history mechanism ensures that users can revisit and reuse past commands easily.
In summary, this project encapsulates the fundamental concepts of shell programming and job control in Unix-like systems. It provides a practical demonstration of process management, signal handling, and command execution, offering users a versatile and functional command-line environment. The implementation serves as a valuable educational tool for understanding the inner workings of command-line interfaces and process management in operating systems.

Introduction

In the realm of operating systems, the command-line interface (CLI) represents a critical component of user interaction, enabling direct and efficient communication with the system. This project focuses on the development of a custom Unix-like shell, named "osh," to explore and demonstrate essential features of shell programming and job control mechanisms. By creating a shell from scratch, we delve into the intricate workings of process management, signal handling, and job control, thereby gaining a deeper understanding of how modern operating systems handle command execution and resource management.

A shell serves as a command interpreter, translating user inputs into system calls and commands that the operating system can execute. Traditionally, shells such as Bash, Zsh, and KornShell offer users a way to interact with the system through textual commands. These shells provide not only the ability to execute commands but also advanced features like job control, command history, and scripting capabilities. By developing "osh," we aim to replicate and understand these fundamental functionalities, providing a hands-on experience with shell operations.

This project begins with the basic functionalities of a command-line shell, including command execution, argument parsing, and process forking. The implementation includes essential built-in commands such as cd for changing directories and exit for terminating the shell session. These basic features lay the groundwork for more complex functionalities.

A significant aspect of this project is the implementation of job control features. Job control allows users to manage multiple processes simultaneously, giving them the ability to run commands in the background or foreground. This is achieved through the implementation of job lists, job addition, and job removal mechanisms. The shell supports commands such as jobs to list active jobs, fg to bring a job to the foreground, and bg to continue a job in the background. By integrating these features, the shell mirrors the functionality of established Unix-like systems, providing users with robust process management capabilities.

Additionally, the project incorporates a signal handling mechanism to manage child processes. The SIGCHLD signal handler is utilized to detect when a child process terminates, allowing the shell to clean up its job list and ensure accurate process tracking. This feature is crucial for maintaining the shell's responsiveness and reliability.

he shell also includes a history management feature that tracks and stores previously executed commands. This allows users to review and reuse past commands, enhancing the efficiency of command-line operations. The integration of history management underscores the importance of user convenience and accessibility in shell environments.

Overall, this project serves as an educational exploration of shell programming and job control in Unix-like systems. By implementing "osh," we gain practical insights into the inner workings of command-line interfaces and process management, while also developing a tool that mirrors many of the features found in conventional shells. Through this endeavor, we enhance our understanding of operating system principles and the essential role of shells in user-system interaction.

Objective

- The primary objective of this project is to develop a custom Unix-like shell, referred to as "osh," to explore and implement fundamental concepts of shell programming and job control mechanisms. This project aims to achieve the following specific objectives:

- Implement Basic Shell Functionality: Develop a functional command-line shell capable of executing basic commands, parsing user input, and handling arguments. The shell should support essential built-in commands such as cd for changing directories and exit for terminating the shell session.

- Integrate Job Control Features: Implement job control mechanisms to manage processes running in the background and foreground. This includes developing features to:
  - Start processes in the background and track their status.
  - List active jobs with a jobs command.
  - Bring a background job to the foreground using the fg command.
  - Continue a stopped job in the background using the bg command.

- Implement Signal Handling: Develop a signal handling mechanism to manage child processes and ensure proper cleanup. Specifically, handle the SIGCHLD signal to detect when child processes terminate and update the job list accordingly.

- Develop Command History Management: Incorporate a command history feature that allows users to view and reuse previously executed commands. This includes storing and managing a limited number of commands in the history and providing a history command to display them.

- Enhance Process Management: Implement process management functionalities such as handling process creation, monitoring, and termination. This includes forking child processes, executing commands, and managing process states.

- Provide a User-Friendly Interface: Design a user interface that is intuitive and responsive, providing users with clear prompts and error messages. Ensure that the shell behaves reliably and handles edge cases gracefully.

By achieving these objectives, the project aims to provide a comprehensive understanding of shell programming and job control, offering practical insights into the management of processes and user interactions within an operating system environment. Through the development of "osh," the project seeks to enhance the user's knowledge of operating system principles and the functionality of command-line interfaces.

OS Concepts Used

- Processes and Process Management:
  - Forking Processes: The fork() system call is used to create a new process (child process) from the current process (parent process). This allows the shell to execute commands in a new process.
  - Process Execution: The execvp() function replaces the current process image with a new process image, executing the command specified by the user.
  - Process Control: The waitpid() function is used to wait for the completion of a child process. It also handles stopping and resuming jobs using the WUNTRACED flag.
  - Signal Handling: The signal() function is used to set up a handler (sigchld_handler) to handle SIGCHLD signals. This signal is sent to the parent process when a child process terminates, allowing the parent to clean up and remove the job from the job list.

- Job Control:
  - Job Management: The implementation includes managing background and foreground jobs. Jobs are tracked using a job list (jobs[]), where each job includes process ID, command, and status.
  - Foreground and Background Execution: The program supports bringing jobs to the foreground (fg) and continuing jobs in the background (bg). This is handled by sending SIGCONT signals to the respective jobs and updating their status.

- File System Operations:
  - File Manipulation: Functions like my_createFile(), my_deleteFile(), and directory listing (my_listFiles()) demonstrate file system operations. These include creating, deleting files, and reading directory contents.
  - Directory Operations: The opendir(), readdir(), and closedir() functions are used to open, read, and close directories, respectively.

- System Information and Resource Management:
  - System Information: Functions such as my_memoryInfo() and my_systemInfo() provide information about system resources, including memory usage (sysinfo()) and kernel version (uname).
  -
- Signal Handling:

  Handling Signals: The sigchld_handler() function processes signals sent by child processes, such as termination or stop signals, and performs necessary actions such as cleaning up the job list.

- Command History:

  History Management: The add_history() and show_history() functions manage a history of previously executed commands. This allows users to view and recall previous commands.

- Process Synchronization:

  Handling Background Processes: The program distinguishes between foreground and background processes. Background processes are handled differently by not waiting for their completion immediately and by using job control commands.

These concepts are integral to the functionality of a shell-like program, providing a robust interface for interacting with the operating system.

Methodology

1. Requirement Analysis:
   - Identify the core features of a basic shell program, including command execution, job control, and command history.
   - Determine the commands and features to be supported, such as cd, my_ls, my_create, my_delete, my_ps, my_kill, my_meminfo, my_sysinfo, history, jobs, fg, and bg.
2. Design:
   - Structure the shell program to handle user input, parse commands, and manage processes.
   - Design the job management system to handle foreground and background processes.
   - Plan for signal handling to manage child process termination and job control.
3. Implementation:
   - Main Loop: Implement a loop to continuously prompt the user for commands, read user input, and process commands.
   - Command Parsing: Use strtok to split the input string into command and arguments.
   - Built-in Commands: Implement built-in command handlers for cd, my_ls, my_create, my_delete, my_ps, my_kill, my_meminfo, my_sysinfo, history, jobs, fg, and bg.
   - Process Management: Use fork to create new processes for command execution and execvp to replace the process image.
   - Job Control: Manage background and foreground processes using a job list and provide job control functionality with kill, waitpid, and signal handling.
   - Signal Handling: Implement a signal handler for SIGCHLD to clean up child processes and update the job list.
4. Testing and Debugging:
   - Test individual components such as command parsing, built-in command execution, and process management.
   - Conduct integration testing to ensure all components work together seamlessly.
   - Debug issues related to process synchronization, signal handling, and command execution.
5. Documentation:
   - Document the code with comments explaining the purpose of functions and key code segments.
   - Prepare a user manual or guide explaining how to use the shell program and its features.

Framework

The framework of the shell program can be described as follows:

- Main Loop:
  - Continuously prompts the user for input and reads commands.

- Input Handling:
  - Reads user input using fgets and removes the newline character.

- Command Parsing:
  - Splits the input string into command and arguments using strtok.

- Built-in Command Execution:
  - Checks if the command matches a built-in command and executes the corresponding function.

- Process Management:
  - Uses fork to create a new process for command execution.
  - Replaces the process image using execvp for external commands.

- Job Control:
  - Manages jobs using a job list.
  - Handles background and foreground processes, allowing them to be brought to the foreground or continued in the background.

- Signal Handling:
  - Implements a signal handler for SIGCHLD to handle child process termination and update the job list.

- Command History:
  - Maintains a history of executed commands.
  - Provides functionality to display and recall previous commands.

- File and System Operations:
  - Implements functions for file creation, deletion, directory listing, process listing, and system information retrieval.

By following this methodology and framework, the shell program is designed and implemented to provide robust functionality, efficient process management, and user-friendly command execution.

Conclusion

The development of our custom shell, osh, has been a comprehensive project that delved into several core operating system concepts. Throughout this project, we explored the intricacies of process creation, management, and communication, gaining a deeper understanding of how operating systems handle user commands and manage resources.

Our shell implementation successfully incorporates essential features such as command execution, job control, and command history management, reflecting a functional and efficient user interface. The built-in commands for file and process management, along with the handling of system information, provide users with a versatile tool for interacting with their system.

The methodology involved in designing and implementing this shell highlighted the importance of careful planning and iterative development. From parsing user inputs to managing background and foreground processes, each component was meticulously developed and tested to ensure robust performance. The integration of signal handling and job control mechanisms, such as bringing jobs to the foreground and continuing jobs in the background, added to the complexity and utility of our shell.

Moreover, this project has reinforced our understanding of Unix-based operating systems, particularly in terms of system calls, process synchronization, and inter-process communication. By implementing features like the command history and job management, we also delved into data structures and memory management, further broadening our practical knowledge.

In summary, the creation of osh has been a rewarding experience that not only solidified our grasp of operating system concepts but also provided a tangible and useful application. This project exemplifies how theoretical knowledge can be applied to develop practical tools that enhance user interaction with the system. Moving forward, this foundation can be expanded to include more advanced features and optimizations, ensuring continued learning and development in the field of operating systems.

References

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). Operating System Concepts (10th ed.). Wiley.
- Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.
- Robbins, A., & Beebe, N. (2005). Classic Shell Scripting. O'Reilly Media.
- Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment (3rd ed.). Addison-Wesley.
- Bovet, D. P., & Cesati, M. (2005). Understanding the Linux Kernel (3rd ed.). O'Reilly Media.