Shell Program Explanation

Author: Apaar(2022089) & Angadjeet Singh(2022071)
github_repo_link ANGADJEET/OS_ASSIGNMENT_2 (github.com)

Overview

This code represents a simple shell program in C. It provides a command-line interface for users to interact with a basic shell environment. The shell supports various operations, including executing single commands, handling pipes for command chaining, maintaining command history, and executing scripts.

The code is organized into several C source files and header files, each responsible for different aspects of the shell's functionality. Here, we'll provide an overview of the main components of the code.

Files and Components

`main.c`

- **Initialization:** The `main` function initializes signal handling for Ctrl+C (SIGINT) using the `sigaction` function.
- **Script Execution:** If the program is run with an argument (script filename), it executes the commands from the script file using the `shell_execute_script` function.
- **Interactive Shell:** If no argument is provided, it enters interactive shell mode using the `shell_loop` function, allowing users to enter commands and interact with the shell.

`utils.c`

- **`timespec_diff_in_ms` Function:** Calculates the time difference in milliseconds between two `timespec` structures.
- **`read_user_input` Function:** Reads user input from the command prompt using `getline` and handles errors gracefully.
- **`split_on_whiteSpaces` and `split_on_pipeSymbol` Functions:** These functions split a line into individual tokens based on whitespace characters and the pipe symbol (`|`), respectively.
- **`my_handler` Function:** Signal handler for handling Ctrl+C (SIGINT) signals. It prints the command history and terminates the shell gracefully.
- **`command_history` Structure:** Defines a structure to store command history information, including command name, whole command, PID, start time, end time, and total duration.
- **`command_history` Array:** Defines an array of `command_history` structures to store command history entries.
- **`print_command_history` Function:** Prints the command history entries.

`shell.c`

- **Command Execution Functions:** This file contains functions for executing commands both with and without pipes. It uses `fork` and `execvp` to create child processes for command execution.
- ** `read_user_input()`:** This function reads user input from the command prompt, handling dynamic memory allocation for the input buffer.It's used to capture user-entered commands and process them within the shell program.

- **`launch_with_pipes(char **commands)`:** This function handles the execution of multiple commands connected by pipes (`|`). When a command contains pipes (indicating a command chain), this function splits the commands, sets up pipes between them, and creates child processes to execute each command in the chain.

- **`launch_without_pipes(char **commands)`:** This function manages the execution of a single command without pipes.It's used when the user enters a single command without pipe symbols. This function creates a child process to execute the command.

- **`execute_with_pipes(int argc, char **argv)`:** This function executes multiple commands connected by pipes (`|`) with proper input and output redirection. It's called by `launch_with_pipes` to create child processes for each command in a pipe chain, set up pipes for communication, and execute the commands accordingly.

- **`execute_without_pipes(char **commands, int background)`:** This function handles the execution of a single command without pipes, with an option to run it in the background. It's used when the user enters a single command. This function creates a child process to execute the command, and in the case of background execution, redirects the output to `/dev/null` to allow the shell to continue accepting commands.

Shell Program Execution Flow

This document provides a detailed explanation of the execution flow of the shell program. The shell program allows users to enter commands interactively or execute scripts, managing command execution, command history, and user interactions. This guide outlines the key steps in the program's execution.

1. Initialization

- The program initializes signal handling for Ctrl+C (SIGINT) in the `main` function using the `sigaction` function.

2. Script Execution (Optional)

- If the program is run with a command-line argument (script filename), it enters script execution mode.

- The `main` function checks the number of command-line arguments (`argc`).
- If an argument is provided, the program executes the commands from the script file using the `shell_execute_script` function.
- Script execution mode allows for automated execution of commands stored in a script file.

3. Interactive Shell Mode (Default)

- If no command-line arguments are provided, the program enters interactive shell mode.
- In this mode, the program acts as a command-line interface, allowing users to enter commands interactively.

4. User Prompt

- The program displays a command prompt, typically showing the user's current working directory (e.g., `iiitd@possum:~$`).

5. Reading User Input

- The `read_user_input` function reads user input from the command prompt using the `getline` function.
- It handles errors gracefully, such as unexpected end-of-file conditions.

6. Handling "history" Command

- The program checks if the user's input is the "history" command.
- If the user enters "history" and presses Enter, the program prints the command history using the `print_command_history` function.
- The command history displays previously executed commands.

7. Command Execution

- If the user enters a command, the program proceeds with command execution.
- It checks if the command contains pipes (`|`) to determine whether it's a single command or a series of commands chained together.

8. Command Splitting

- If the command contains pipes, the program splits the command into individual commands using the `split_on_pipeSymbol` function.
- Each individual command is stored as a separate string.

9. Command Execution with Pipes

- If the command has pipes, the program executes the commands with pipes using functions in `command_execution.c`.
- Child processes are created for each command in the chain using `fork`, and commands are executed with `execvp`.
- Output of one command is connected to the input of the next using pipes, allowing for command chaining.

10. Command Execution without Pipes

- If the command does not contain pipes, it is considered a single command.
- The program executes the single command using the `execute_without_pipes` function.
- Child processes are created using `fork`, and the command is executed with `execvp`.

11. Recording Command Execution Time

- For each executed command (single or part of a pipe chain), the program records the start and end times.
- It uses the `clock_gettime` function to obtain precise timestamps.

12. Updating Command History

- The program maintains a history of executed commands, including command name, whole command, PID, start time, end time, and total duration.
- If the command history is not full (limited by `MAX_COMMAND_HISTORY`), the program updates the history with command details.
- If the history is full, a message is displayed indicating that no more commands can be stored.

13. Looping and User Interaction

- The program operates within a loop, allowing users to enter multiple commands.
- After executing each command, it returns to the command prompt, awaiting the next user input.
- Users can continue to enter commands until they decide to exit the shell.

14. Graceful Termination

- Users can exit the shell by either entering a specific exit command or by pressing Ctrl+C (SIGINT).
- In the event of a Ctrl+C signal, the program gracefully terminates the shell, printing the command history before exiting.

Features

- Command execution with and without pipes.

- Command history tracking and printing.
- Script execution from a file.
- Graceful handling of errors and signals.
- User-friendly command-line interface.

Limitations of the programme

There are certain limitations of the code that is, not working of the specific commands because of the limitations of logic and as required in the assignment we are mentioning some of them below:

1. **Alias:**

   - The code does not implement custom alias handling. Aliases are not recognized or expanded because the code lacks the necessary logic to identify and process custom alias definitions.

2. **Exit:**

   - The code does not provide an option to exit the shell itself. While it processes user commands continuously, there is no built-in "exit" command to terminate the shell program. Executing "exit" simply returns to the command prompt.

3. **Return:**

   - The code does not execute functions or scripts."Return" is used within functions or scripts to indicate values to return. Since your shell does not execute functions or scripts, there is no context for the "return" keyword.

4. **Break:**

   - The code does not include loop constructs."Break" is used within loops to exit them prematurely. Your code primarily focuses on executing single commands and lacks loop constructs.

5. **Unset:**

   - The code does not implement variable management.The "unset" command, used to remove variables, is not supported as the code lacks logic for variable management.

6. **Export:**

- The code does not implement variable management. "Export" is used to make variables available to child processes. This requires custom logic for managing variables and their export status, which is not present in the code.

7. **Redirections (e.g., `>`, `>>`, `<`):**

- The code does not support input/output redirection.Redirection operators like `>`, `>>`, and `<` are used to redirect input or output streams. The code does not include logic to interpret or process these operators, limiting its ability to perform file input/output operations.

8. **Conditional Execution (e.g., `&&`, `||`):**
- The code does not implement conditional execution of commands.Conditional execution operators like `&&` (AND) and `||` (OR) allow commands to be executed based on the success or failure of previous commands. The code does not include logic to evaluate and execute commands conditionally, making it unable to handle such constructs.

* The "cd" command should also not run for the similar reasons but we have implemented the code for working of cd command.*

Contributions

Apaar(2022089): 50%

- Made basic structure of the code and worked intensively on error analysis.
- Completed the bonus part of the assignment and report writing.

Angadjeet Singh(2022071): 50%

- Focused on the code implementation.
- Did all the logic framework and defined the required system calls and variables.