Shell Command Executor in C

Group Members:

Abubakar Waziri - 4220056

Muhammad Alansari - 4312897

Overview

Implemented a shell command executor in C with features including:

- Command execution.
- Command history management.
- Simultaneous execution of commands using the join() command.

Key Features

- **Command Execution:** Run shell commands such as ls, pwd, whoami.
- **Command History:** Maintain and display the last 100 commands.
- **Join Command:** Execute two commands concurrently.
- **Thread Safety:** Protect command history using a mutex.

```
        printf("Exiting...\n");
        break;
    }

    // Check if the command is supported
    if (args[0] != NULL && !is_supported_command(args[0]))
    {
        printf("Error: '%s' is not a supported command.\n", args[0]);
        continue;
    }

    // Handle help command
    if (args[0] != NULL && strcmp(args[0], "help") == 0)
    {
        print_help();
        continue;
    }

    // Handle history command
    if (args[0] != NULL && strcmp(args[0], "history") == 0)
    {
        print_history(history, history_count);
        continue;
    }

    // Handle clearhistory command
    if (args[0] != NULL && strcmp(args[0], " clearhistory") == 0)
```

```c
void print_histo[][MAX_INP
{
    pthread_mut[his]ry_mutex);
    pr[mand Hi]n");
    for [i = 0; i < ]ory_count;
    {
        printf("%d: %s\n", i + 1, hist
    }
    pthread_mutex_unlock(&history_mutex);
}

void clear_history(char history[][MAX_INPUT_SIZE], int *h
{
    pthread_mutex_lock(&history_mutex);
    *history_count = 0;
    printf("Command history cleared.\n");
    pthread_mutex_unlock(&history_mutex);
}

void *run_command(void *arg)
{
    char **args = (char **)arg;
    if (execvp(args[0], args) < 0)
    {
        perror("execvp failed");
        exit(EXIT_FAILURE);
    }
    return NULL;
}
```

Command Execution

Core Function: run_command()

- Executes commands using execvp().
- Forks a new process for each command.
- Handles errors if command execution fails.

Command History Management

History Storage: Keeps a record of the last 100 commands.

Mutex Protection: Ensures thread safety for concurrent access.

Functions: print_history(): Displays command history. And clear_history(): Clears stored history.

Resource Cleanup: pthread_mutex_destroy(&history_mutex); frees mutex resources when no longer needed.

ase3.c > main()

```c
int main()
    while (1)

        // Check if the command is supported
        if (args[0] != NULL && !is_supported_command(args[0]))
        {
            printf("Error: '%s' is not a supported command.\n", args[0]);
            continue;
        }

        // Handle help command
        if (args[0] != NULL && strcmp(args[0], "help") == 0)
        {
            print_help();
            continue;
        }

        // Handle history command
        if (args[0] != NULL && strcmp(args[0], "history") == 0)
        {
            print_history(history, history_count);
            continue;
        }

        // Handle clearhistory command
        if (args[0] != NULL && strcmp(args[0], " clearhistory") == 0)
        {
            clear_history(history, &history_count);
            continue;
        }

        // Handle join command
        if (args[0] != NULL && strcmp(args[0], "join") == 0)
        {
            char cmd1_input[MAX_INPUT_SIZE];
            char cmd2_input[MAX_INPUT_SIZE];
            char *cmd1[MAX_ARG_SIZE], *cmd2[MAX_ARG_SIZE];

            // Get first command
            get_command("Enter your first command: ", cmd1_input);

            // Get second command
            get_command("Enter your second command: ", cmd2_input);
```

```
{
    {
        {
            get_command("Enter your first command: ", cmd1

            // Get second command
            get_command("Enter your second command: ", cmd2_inpu

            // Tokenize first command
            int j = 0;
            cmd1[j] = strtok(cmd1_input, " ");
            while (cmd1[j] != NULL && j < MAX_ARG_SIZE - 1)
            {
                j++;
                cmd1[j] = strtok(NULL, " ");
            }
            cmd1[j] = NULL;

            // Tokenize second command
            j = 0;
            cmd2[j] = strtok(cmd2_input, " ");
            while (cmd2[j] != NULL && j < MAX_ARG_SIZE - 1)
            {
                j++;
                cmd2[j] = strtok(NULL, " ");
            }
            cmd2[j] = NULL;

            // Fork the first command
            pid_t pid1 = fork();
            if (pid1 < 0)
            {
                perror("Fork failed for first command");
                continue;
            }
            else if (pid1 == 0)
            {
                // Child process for first command
                if (execvp(cmd1[0], cmd1) < 0)
                {
                    perror("execvp failed for first command");
                    exit(EXIT_FAILURE);
                }
```

Join Command Implementation

What It Does:

Allows users to execute two commands simultaneously.

Implementation Steps:

Condition Check:

- Verify the first argument is join.

Input Commands:

- Use get_command() to capture two commands.

Tokenization:

- Split commands into tokens using strtok(). And Prepare arguments for execvp().

Fork Processes:

- Create two child processes using fork(). And Each process runs one command with execvp().

Parent Process Wait:

- Wait for both processes to complete using waitpid().

Threading and Process Control

Thread Safety:

Mutex (pthread_mutex_t) ensures safe access to shared data.

Process Management:

- fork() creates new processes.
- execvp() executes commands.
- wait() and waitpid() synchronize processes.

```
        // Child proce
        if (execvp(cmd2                < 0)

            perror("execv led for second command");
            exit(EXIT_FAILURE);


        // Parent process waits for both commands to finish
        waitpid(pid1, NULL, 0); // Wait for first command
        waitpid(pid2, NULL, 0); // Wait for second command

        continue;
    }

    // Fork a child process for other commands
    pid = fork();
    if (pid < 0)
    {
        perror("fork failed");
        continue;
    }

    if (pid == 0)
    {
        // Child process
        if (execvp(args[0], args) < 0)
        {
            perror("execvp failed");
            exit(EXIT_FAILURE);
        }
    }
    else
    {
        // Parent process
        wait(&status);
    }
```

```c
#define MAX_INPUT_SIZE 1024
#define MAX_ARG_SIZE 100
#define MAX_HISTORY_SIZE 100

pthread_mutex_t history_mutex;

void print_prompt()
{
    printf("phase3-shell> ");
}

void print_help()
{
    printf("Available commands:\n");
    printf("  help\n");
    printf("  ls\n");
    printf("  ps\n");
    printf("  pwd\n");
    printf("  date\n");
    printf("  whoami\n");
    printf("  uname\n");
    printf("  df\n");
    printf("  history\n");
    printf("  clearhistory\n");
    printf("  join\n");
    printf("  exit\n");
}

void print_history(char history[][MAX_INPUT_SIZE], in
{
```

Key Functions

Core Functions:

- print_prompt(): Displays the shell prompt.
- get_command(): Reads user input.
- run_command(): Executes a command in a child process.

Utility Functions:

- is_supported_command(): Verifies if a command is supported.
- print_help(): Lists available commands.

Error Handling

Common Errors Managed:

- Invalid commands or unsupported operations.
- Failure during forking or command execution.

Error Messages:

- Inform users of issues like unsupported commands or execution failures using perror().

```
    j++;
    cmd2[j] = strtok(NULL, " ");

cmd2[j] = NULL;

// Fork the first command
pid_t pid1 = fork();
if (pid1 < 0)
{
    perror("Fork failed for first command");
    continue;
}
else if (pid1 == 0)
{
    // Child process for first command
    if (execvp(cmd1[0], cmd1) < 0)
    {
        perror("execvp failed for first command");
        exit(EXIT_FAILURE);
    }
}

// Fork the second command
pid_t pid2 = fork();
if (pid2 < 0)
{
    perror("Fork failed for second command");
    continue;
}
else if (pid2 == 0)
{
    // Child process for second command
    if (execvp(cmd2[0], cmd2) < 0)
    {
        perror("execvp failed for second command");
        exit(EXIT_FAILURE);
    }
}
```

```
(base) aiwaziri@MacBookPro PROJECT % gcc -
(base) aiwaziri@MacBookPro PROJECT % ./phase
phase3-shell> help
Available commands:
  help
  ls
  ps
  pwd
  date
  whoami
  uname
  df
  history
  clearhistory
  join
  exit
phase3-shell> sp
Error: 'sp' is not a supported command.
phase3-shell> ls
CS221 - Project (Fall 2024-2025).pdf    phas
Project report-final-os.docx            phas
Project report-final-os.pdf             phase
phase-1-OS-project.png                  phase2
phase-2-OS-project.png                  phase3
phase-3-OS-project.png                  phase3.c
phase3-shell> join
Enter your first command: whoami
Enter your second command: uname
Darwin
aiwaziri
phase3-shell> exit
Exiting...
(base) aiwaziri@MacBookPro PROJECT %
```

Discussion on Results

Successfully implemented and tested the shell command executor.

Results:

- Executed multiple commands, maintained history without errors.
- Verified join command for concurrent execution of commands.

Key Changes in the Updated Version

Added is_supported_command() to verify commands before execution.

Improved error handling for unsupported commands.

```
        }
        input[strcspn(input, "\n")] = 0;
    }

    bool is_supported_command(char *command)
    {
        const char *supported_commands[] = {
            "help", "ls", "ps", "pwd", "date", "whoami", "uname", "df", "history", "clearhistory", "join", "exit"};
        for (int i = 0; i < sizeof(supported_commands) / sizeof(supported_commands[0]); i++)
        {
            if (strcmp(command, supported_commands[i]) == 0)
            {
                return true;
            }
```

Conclusion

Achievements:

- Implemented a functional shell command executor.
- Supported key features: history, threading, and concurrent execution.

Future Improvements:

- Add more built-in commands.
- Enhance error handling and input validation.
- Improve user interface for better usability.

phase3.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <pthread.h>
#include <stdbool.h>

#define MAX_INPUT_SIZE 1024
#define MAX_ARG_SIZE 100
#define MAX_HISTORY_SIZE 100

pthread_mutex_t history_mutex;

void print_prompt()
{
    printf("phase3-shell> ");
}
```

Keywords and Terminology

Key Headers: stdio.h, stdlib.h, unistd.h, etc.

Threading: pthread_mutex_t and mutex operations.

Process Control: fork(), execvp(), waitpid().

Error Handling: perror(), exit(EXIT_FAILURE).

Shell Commands: help, ls, history, join, exit. etc

Questions & Answers