Project report

Group Members:

Abubakar Waziri -4220056

Muhammad Alansari -4312897

Overview

We implement a simple shell command executor in C. It allows users to execute various shell commands, maintain a history of executed commands, and execute two commands simultaneously using the **join** command.

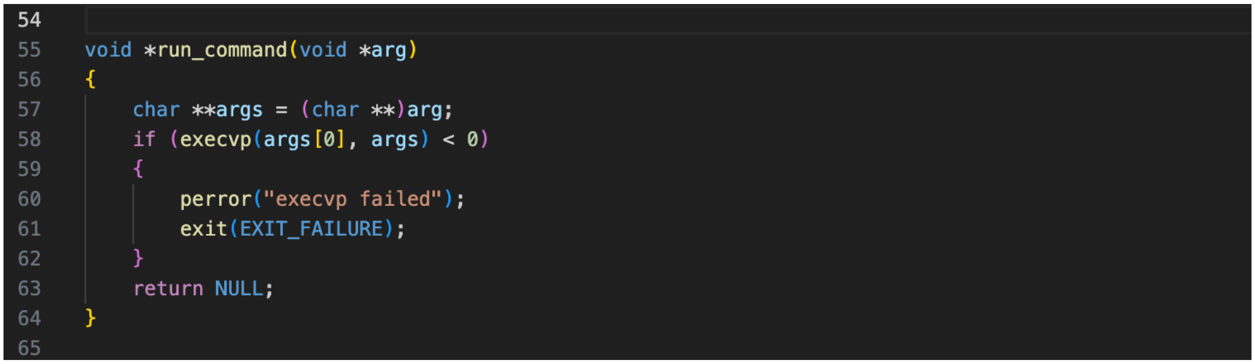
Key Features

1. **Command Execution**: Users can execute shell commands like **ls**, **pwd**, **whoami**, etc.
2. **Command History**: The shell maintains a history of the last 100 commands executed, which can be displayed or cleared.
3. **Join Command**: Users can execute two commands in simultaneously.
4. **Thread Safety**: The command history is protected with a mutex to ensure thread safety when accessed by multiple threads.

Important Code Sections

Command Execution

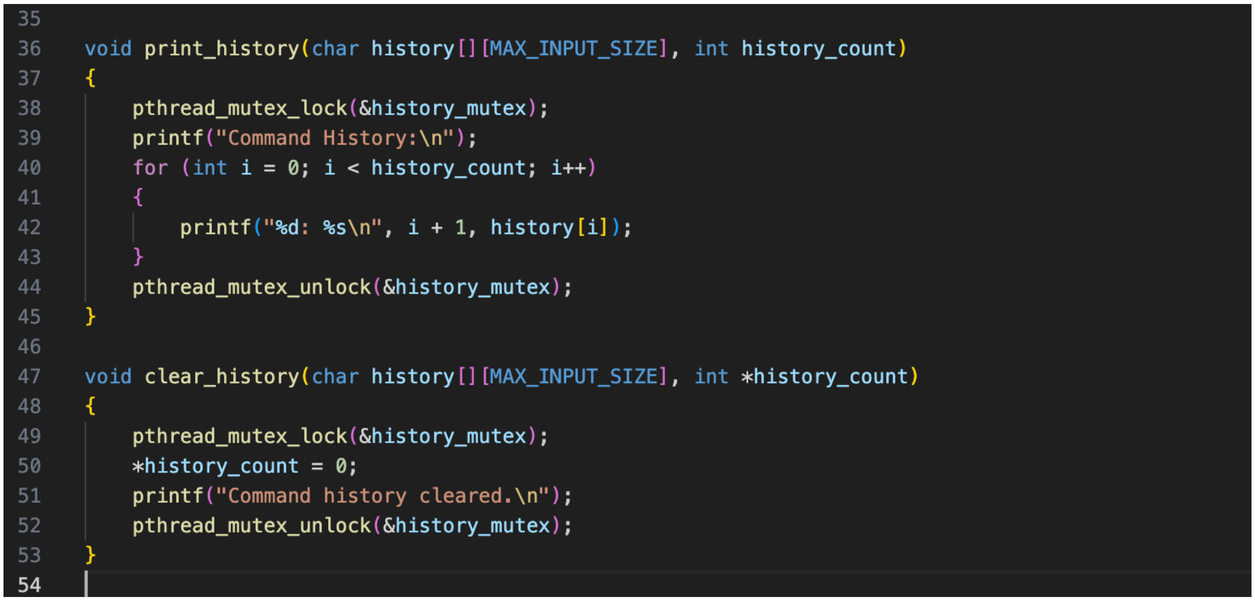
The main logic for executing commands is handled in the **run_command** function. This function uses **execvp** to execute commands passed as arguments.



```c

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

```

Command History Management

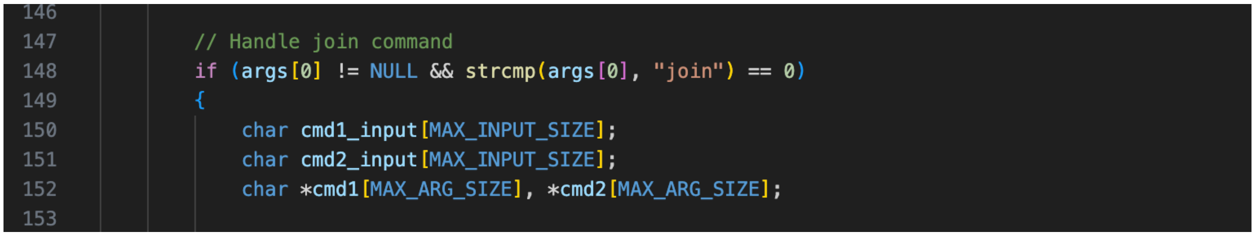
The command history is managed with a **mutex** to ensure that it is accessed safely in a multithreaded environment.



```c

void print_history(char history[][MAX_INPUT_SIZE], int history_count)
{
    pthread_mutex_lock(&history_mutex);
    printf("Command History:\n");
    for (int i = 0; i < history_count; i++)
    {
        printf("%d: %s\n", i + 1, history[i]);
    }
    pthread_mutex_unlock(&history_mutex);
}

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

```

Join Command Implementation

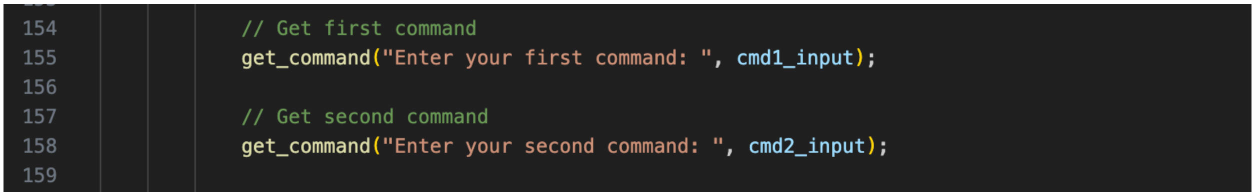
The **join** command allows users to execute two commands concurrently. The implementation forks two processes, one for each command.:



```c

        // 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];

```

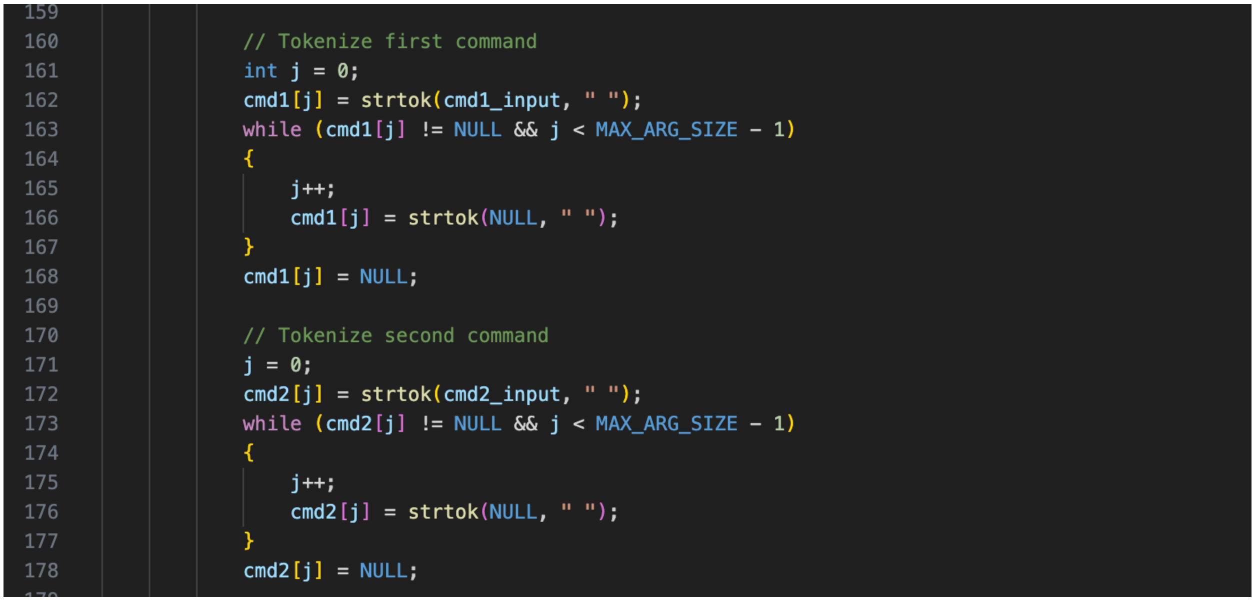
- **Condition Check**: The code first checks if the first argument (**args[0]**) is not **NULL** and if it equals **"join"**. *This determines if the user wants to execute the **join** command.*

- **Variable Declarations(characters)**: It declares variables to hold the input commands (**cmd1_input** and **cmd2_input**) and arrays to store the tokenized arguments for each command (**cmd1** and **cmd2**).



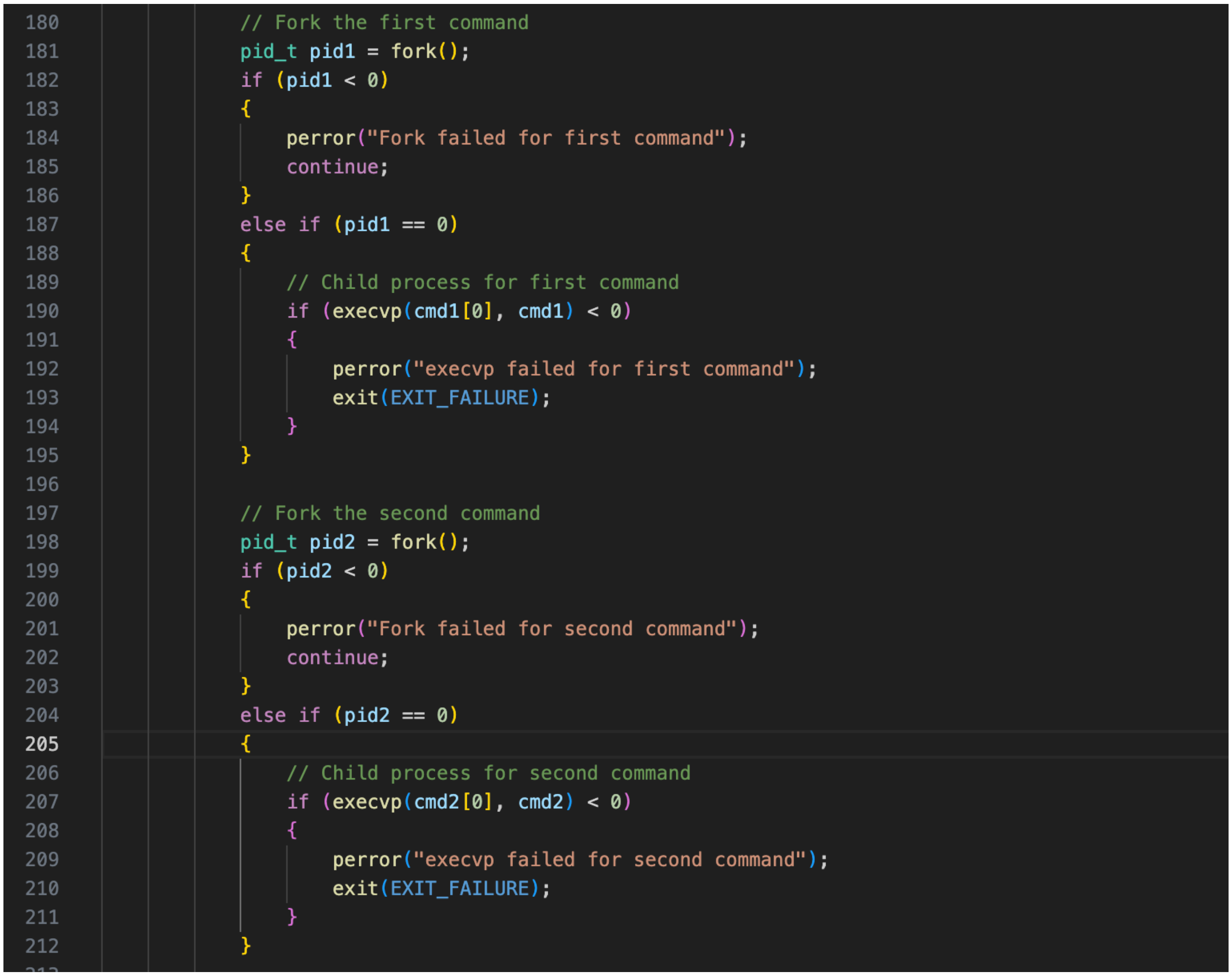
```c
            // Get first command
            get_command("Enter your first command: ", cmd1_input);

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

```

- **Get Commands**: The **get_command** function prompts the user to enter two commands. The first command is stored in **cmd1_input**, and the second in **cmd2_input**.



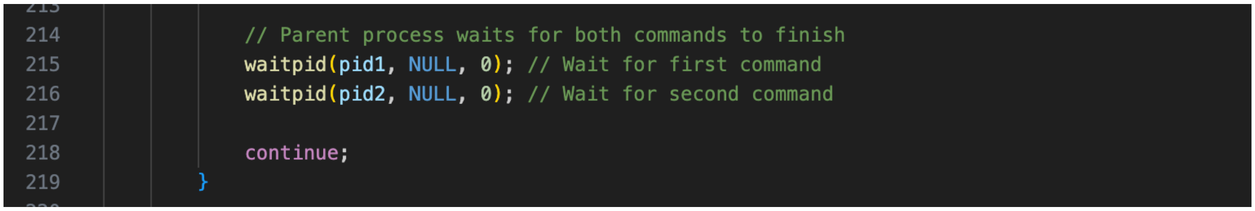
```c

            // 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;
```

- **Tokenization of the First & Second Command**:
    - The **strtok** function is used to split **cmd1_input** into tokens based on spaces.
    - The first token is assigned to **cmd1[0]**, and subsequent tokens are assigned in the loop until there are no more tokens.
    - The last element of **cmd1** is set to **NULL**, which is necessary for the **execvp** function to know where the argument list ends.



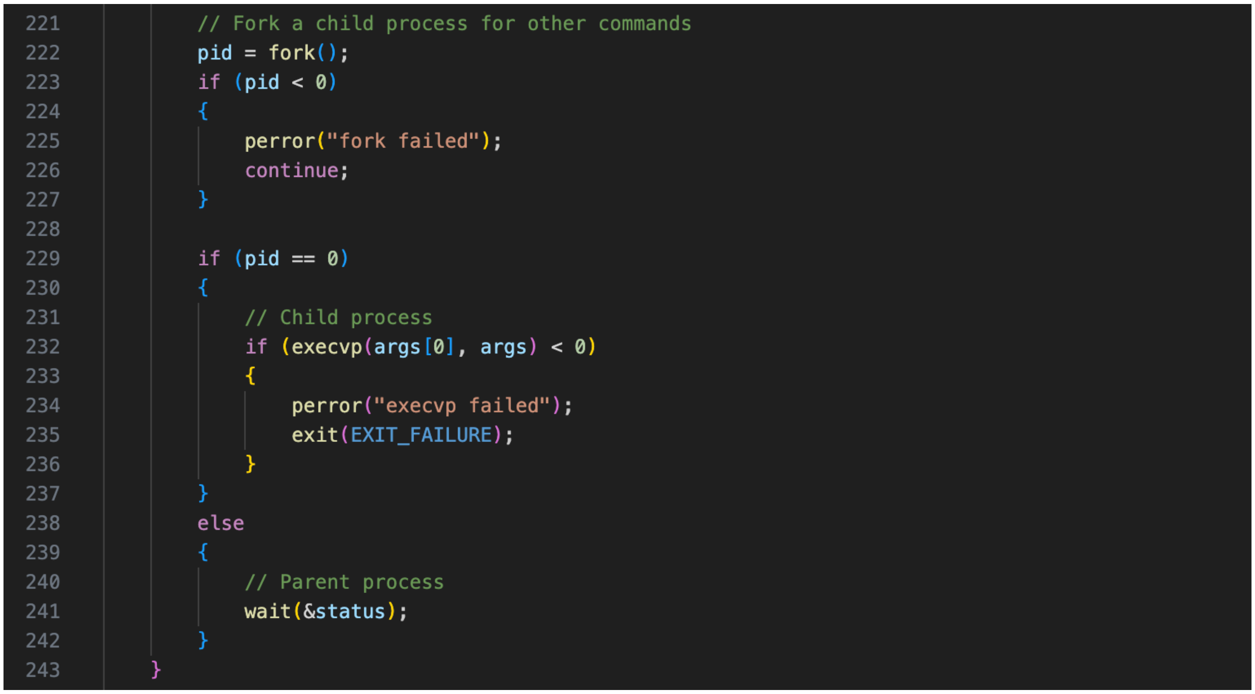
```c
            // Fork the first command
            pid_t pid1 = fork();
            if (pid1 < 0)
            {
                perror("Fork failed for first command");
                continue;
            }
            else if (pid1 == 0)
            {
                // Child process for first 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);
                }
            }
```

- **Forking for the First Command**:
    - A new process is created using **fork()**. If **fork()** fails, an error message is printed.
    - If the process is the child (**pid1 == 0**), it attempts to execute the first command using **execvp**. If **execvp** fails, it prints an error message and exits.
- **Forking for the Second Command:** same thing as the first command's forking logic. It creates a second child process for executing the second command.



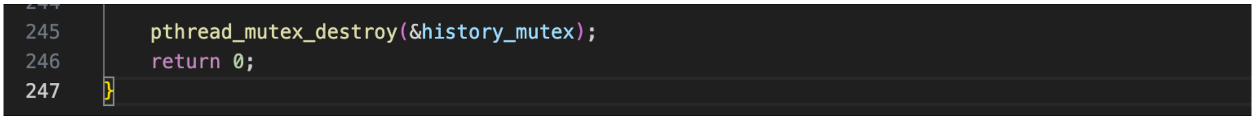
```c
            // Parent process waits for both commands to finish
            waitpid(pid1, NULL, 0); // Wait for first command
            waitpid(pid2, NULL, 0); // Wait for second command

            continue;
        }
```

- **Waiting for Child Processes**: The parent process waits for both child processes to finish executing their respective commands using **waitpid()**. This ensures that the parent does not proceed until both commands have completed.
- **Continue Statement**: This statement ensures that if the **join** command was executed, the shell will skip any further processing and wait for the next user input.



```c
        // 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);
        }
    }
```

- **Handling Other Commands**: If the command is not **join**, the code forks a new child process for executing other commands.
- **Forking Logic**: similar to the other one., it checks for errors during **fork()**. If successful, the child process attempts to execute the command using **execvp**. If it fails, an error message is printed, and the child exits.
- **Parent Process Wait**: The parent process waits for the child to finish executing the command using **wait()**, ensuring proper synchronization.



```c
    pthread_mutex_destroy(&history_mutex);
    return 0;
}
```

1. **pthread_mutex_destroy(&history_mutex);**: This function call cleans up and frees resources associated with the **history_mutex**, which is used to synchronize access

to shared data (like command history) among threads. It should only be called when the mutex is no longer needed.

Discussion on Results

The shell command executor was successfully implemented and tested

The results show that the shell can execute multiple commands and maintain a history without issues. The implementation of the **join** command works as intended, allowing for concurrent execution of commands.

Conclusion

The project successfully demonstrates the ability to create a basic shell command executor with essential features like command history and concurrent execution. Future improvements could include adding more built-in commands and enhancing user experience with better error handling and input validation.

Keywords:

1. **Headers**: Files included to use standard library functions and types.
    - **stdio.h**: Input/output functions.
    - **stdlib.h**: General utilities like memory allocation.
    - **unistd.h**: Unix standard functions (e.g., fork, exec).
    - **sys/types.h**: Data types used in system calls.
    - **sys/wait.h**: Macros for process termination.
    - **string.h**: String handling functions.
    - **pthread.h**: POSIX thread (threading) functions.
2. **Macros**: Constants defined for easy reference.
    - **MAX_INPUT_SIZE**: Maximum size for input strings.
    - **MAX_ARG_SIZE**: Maximum number of command arguments.
    - **MAX_HISTORY_SIZE**: Maximum number of commands to store in history.
3. **Data Types**:

   - **pthread_mutex_t**: Data type for mutex locks in threading.
   - **char**: Character data type.
   - **char ****: Pointer to a pointer of characters (array of strings).
   - **int**: Integer data type.
   - **pid_t**: Data type for process IDs.

4. **Functions**: Blocks of code that perform specific tasks.
   - **print_prompt()**: Displays the shell prompt.
   - **print_help()**: Lists available commands.
   - **print_history()**: Displays command history.
   - **clear_history()**: Clears the command history.
   - **run_command()**: Executes a command in a new thread.
   - **get_command()**: Reads user input.
   - **main()**: Entry point of the program.
5. **Mutex Operations**: Functions for thread synchronization.
   - **pthread_mutex_lock()**: Locks a mutex.
   - **pthread_mutex_unlock()**: Unlocks a mutex.
   - **pthread_mutex_init()**: Initializes a mutex.
   - **pthread_mutex_destroy()**: Destroys a mutex.
6. **Process Control**: Functions for managing processes.
   - **fork()**: Creates a new process.
   - **execvp()**: Executes a program with arguments.
   - **wait()**: Waits for process termination.
   - **waitpid()**: Waits for a specific process to terminate.
7. **Input Handling**: Functions for reading and processing input.
   - **fgets()**: Reads a line of input.
   - **strcspn()**: Finds the length of a substring.

- **strtok()**: Tokenizes a string.
- **strcpy()**: Copies a string.

8. **Control Structures**: Constructs for controlling the flow of execution.
   - **while (1)**: Infinite loop.
   - **if (condition)**: Conditional statement.
   - **else if (condition)**: Alternative conditional statement.
   - **for (initialization; condition; increment)**: Loop with initialization, condition, and increment.
9. **Error Handling**: Techniques for managing errors.
   - **perror()**: Prints a descriptive error message.
   - **exit(EXIT_FAILURE)**: Exits the program with a failure status.
10. **Commands**: Built-in commands for the shell.
    - **help**: Displays help information.
    - **ls**: Lists directory contents.
    - **ps**: Displays current processes.
    - **pwd**: Prints working directory.
    - **date**: Shows the current date and time.
    - **whoami**: Displays the current user.
    - **uname**: Shows system information.
    - **df**: Displays disk space usage.
    - **history**: Shows command history.
    - **clearhistory**: Clears the command history.
    - **join**: Combines two commands.
    - **exit**: Exits the shell.

Key Changes for the new updated version:

- Added a function ***'is_supported_command'*** to check if the entered command is in the list of supported commands, if it’s not then it will return and error text saying that the command not supported.
- Before forking a child process, the program now verifies if the command is supported. If not, it informs the user and continues to the next prompt.