T.C. MARMARA UNIVERSITY

FACULTY of ENGINEERING COMPUTER ENGINEERING DEPARTMENT

CSE 3033 - Operating Systems

Programming Assignment 2 Report

Group Members

150120031 - Şükrü Can MAYDA

150121545 - Nurbetül ÇAKIR

150116837 – Oğuz Kaan NAS

Introduction

The purpose of this project is to develop a terminal application (shell) in C. The shell is expected to process user commands, execute external programs, manage processes, and provide additional functionality such as command history, process control, and input/output redirection. The program mimics the behavior of standard shells while incorporating specific features detailed in the project requirements.

Part A: Command Execution and Process Management

In this part of the program, the shell creates a new process for each user command. It performs the following steps:

1. Path Search:
- The program retrieves directories listed in the 'PATH' environment variable using the 'getenv' function.
- Each directory is searched for the command using the 'access' function to ensure it exists and is executable.

2. Process Creation:
- The 'fork' system call is used to create a child process.
- If 'fork' returns a value less than 0, an error message is displayed.
- The child process executes the program using the 'execv' function. If the execution fails, an appropriate error message is displayed.

3. Foreground and Background Execution:
- Commands ending with '&' are executed as background processes. The shell does not wait for their completion.
- For foreground processes, the parent process waits for the child process to finish using the 'waitpid' function.

Part B: History and Process Control

In this part, the shell provides command history functionality and process management features:

1. Command History:
- The program stores the last 10 commands entered by the user.
- The 'history' command displays the stored commands.
- The 'history -i <index>' command allows the user to re-execute a specific command from the history.

2. Background Process List:
- Background process IDs are stored in a list.
- The 'fg %<pid>' command brings a background process to the foreground.

3. Exit Command:
- When the user enters 'exit', the shell checks for running background processes. If any are found, the shell prompts the user to terminate them before exiting.

Part C: Input/Output Redirection

In this part, the shell supports I/O redirection as follows:

1. Standard Output Redirection:
- 'command > file.out' writes the output of a command to a file, overwriting its contents.
- 'command >> file.out' appends the output to the end of the file.

2. Standard Input Redirection:
- 'command < file.in' reads input for the command from a file.

4. Combined Redirection:

- ‘command < file.in > file.out’ uses a file as input and writes the output to another file.

Implemented Methods

setup

The setup method processes user input from stdin, breaking it down into arguments for execution. It accepts raw input, a string array of arguments, and a background flag, referred to as inputBuffer, args, and background, respectively. The method reads the input into inputBuffer, splits the input string into distinct arguments, and supports background processes by identifying a flag for the character. It also handles null termination of the input to ensure proper formatting. If the input comes from history, it uses strlen() to determine its length; otherwise, it reads directly from stdin.

```
void setup(char inputBuffer[], char *args[], int *background) {
    int length, i, start, ct;
    ct = 0;
    if (isFromHistory == 1) {
        length = strlen(inputBuffer) + 1;
    } else {
        length = read(STDIN_FILENO, inputBuffer, MAX_LINE);
    }
    if (length == 0) exit(0);
    if ((length < 0) && (errno != EINTR)) {
        perror("Error reading command");
        exit(-1);
    }
    start = -1;
    *background = 0;
    for (i = 0; i < length; i++) {
        if ((i + 1 == length) && (inputBuffer[i] != '\n')) {
            if (start != -1) {
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            args[ct] = NULL;
            continue;
        }
        switch (inputBuffer[i]) {
            case ' ':
            case '\t':
                if (start != -1) {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0';
                start = -1;
                break;
            case '\n':
                if (start != -1) {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0';
                args[ct] = NULL;
                break;
            default:
                if (start == -1) start = i;
                if (inputBuffer[i] == '&') {
                    *background = 1;
                    inputBuffer[i] = '\0';
                }
        }
    }
    args[ct] = NULL;
}
```

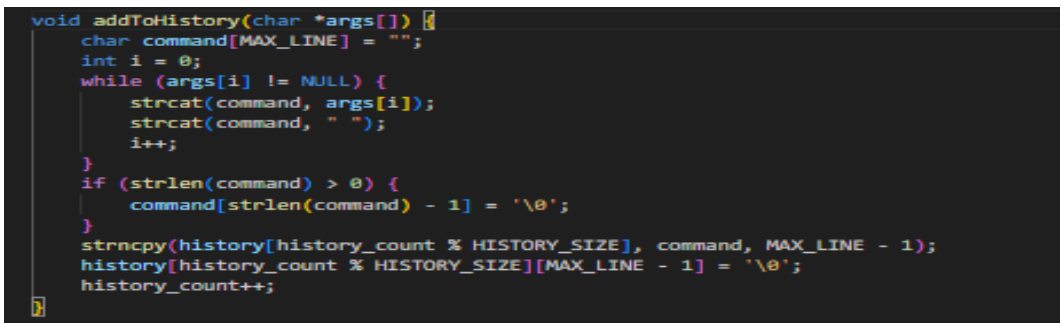
countWords

The countWords method counts the number of words in a given string. It takes a string argument called buffer and works by distinguishing between whitespace and non-whitespace characters while keeping track of whether it is currently within a word using the inWord flag.

```
int countWords(const char *buffer) {
    int inWord = 0;
    int wordCount = 0;
    while (*buffer != '\0') {
        if (isspace((unsigned char)*buffer)) {
            inWord = 0;
        } else if (!inWord) {
            inWord = 1;
            wordCount++;
        }
        buffer++;
    }
    return wordCount;
}
```

addToHistory

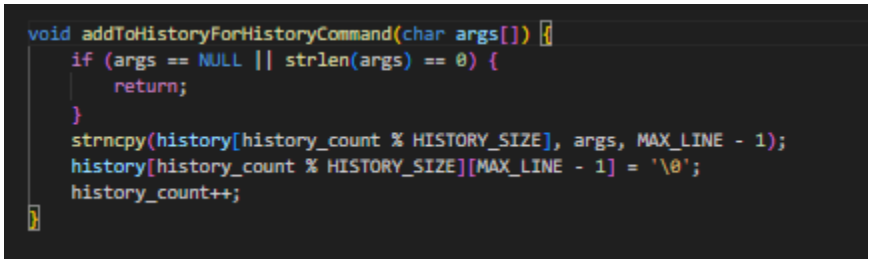
The addToHistory method appends a command to the history array. It takes a string array argument called args and constructs a single string command by concatenating all arguments with spaces. To maintain a fixed size for the history, it overwrites older entries using modulo indexing and increments history_count to keep track of the total number of commands entered.



```
void addToHistory(char *args[]) {
    char command[MAX_LINE] = "";
    int i = 0;
    while (args[i] != NULL) {
        strcat(command, args[i]);
        strcat(command, " ");
        i++;
    }
    if (strlen(command) > 0) {
        command[strlen(command) - 1] = '\0';
    }
    strncpy(history[history_count % HISTORY_SIZE], command, MAX_LINE - 1);
    history[history_count % HISTORY_SIZE][MAX_LINE - 1] = '\0';
    history_count++;
}
```

addToHistoryForHistoryCommand

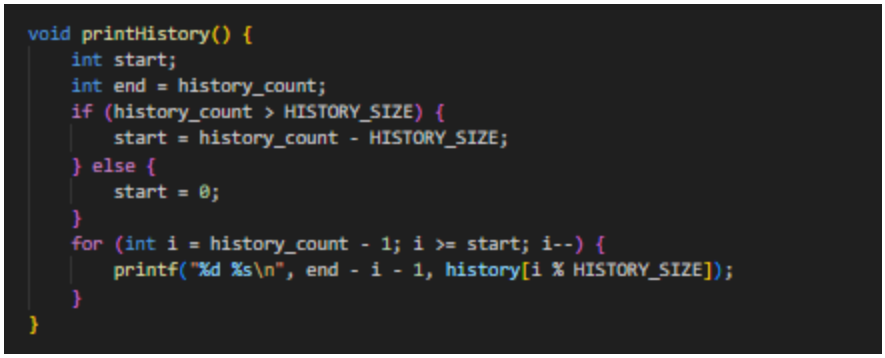
The addToHistoryForHistoryCommand method adds a string command to the history. It takes a string array argument called args and functions similarly to the addHistory methods, but it directly accepts a string command instead of an array of arguments.



```
void addToHistoryForHistoryCommand(char args[]) {
    if (args == NULL || strlen(args) == 0) {
        return;
    }
    strncpy(history[history_count % HISTORY_SIZE], args, MAX_LINE - 1);
    history[history_count % HISTORY_SIZE][MAX_LINE - 1] = '\0';
    history_count++;
}
```

printHistory

The printHistory method displays the last HISTORY_SIZE commands stored in the history. It does not require any arguments. This method calculates the start index based on the current history_count, iterates through the history from the most recent to the oldest, and prints each command along with its index relative to the current session.



```
void printHistory() {
    int start;
    int end = history_count;
    if (history_count > HISTORY_SIZE) {
        start = history_count - HISTORY_SIZE;
    } else {
        start = 0;
    }
    for (int i = history_count - 1; i >= start; i--) {
        printf("%d %s\n", end - i - 1, history[i % HISTORY_SIZE]);
    }
}
```

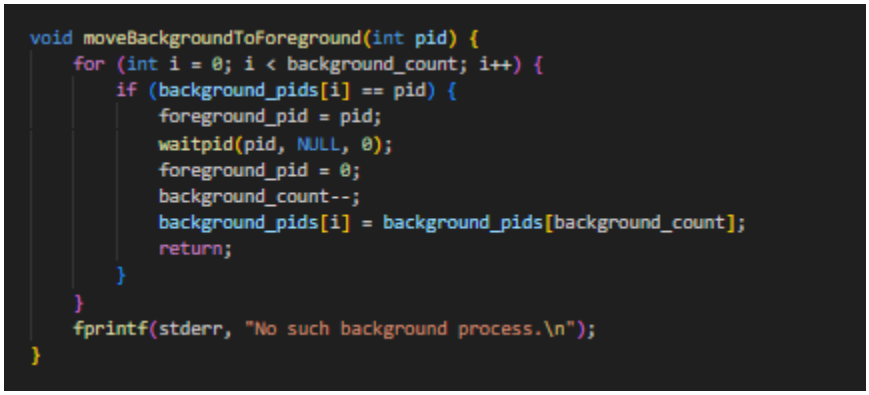
historyCommand

The historyCommand method executes a command from the history based on its index. It accepts an integer for the index, a string array for the input buffer, another string array for arguments, and an integer for background processing. The method validates the index against the history array, retrieves the corresponding command string, sets it as the current input buffer, adds the command to the history, and then uses fork to create a child process to execute the command.

```c
void historyCommand(int index, char inputBuffer[], char *args[], int *background) {
    int start;
    int end = history_count;
    isFromHistory = 1;
    if (history_count > HISTORY_SIZE) {
        start = history_count - HISTORY_SIZE;
    } else {
        start = 0;
    }
    int command_Index = history_count - 1 - index;
    if (command_Index < start || command_Index >= end) {
        fprintf(stderr, "Invalid history index.\n");
        return;
    }
    char *command = history[command_Index % HISTORY_SIZE];
    printf("Executing command from history: %s\n", command);
    strncpy(inputBuffer, command, MAX_LINE - 1);
    inputBuffer[MAX_LINE - 1] = '\0';
    addToHistoryForHistoryCommand(command);
    setup(inputBuffer, args, background);
    isFromHistory = 0;
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        return;
    }
    if (pid == 0) {
        handleRedirection(args);
        char *pathEnv = getenv("PATH");
        if (!pathEnv) {
            fprintf(stderr, "PATH not set\n");
            exit(1);
        }
        char *pathDir = strtok(pathEnv, ":");
        while (pathDir) {
            snprintf(commandPath, sizeof(commandPath), "%s/%s", pathDir, args[0]);
            if (access(commandPath, X_OK) == 0) {
                execv(commandPath, args);
                perror("execv failed");
                exit(1);
            }
            pathDir = strtok(NULL, ":");
        }
        fprintf(stderr, "Command not found: %s\n", args[0]);
        exit(1);
    } else {
        if (!*background) {
            waitpid(pid, NULL, 0);
        } else {
            printf("Background process started with PID %d\n", pid);
        }
    }
}
```

moveBackgroundToForeground

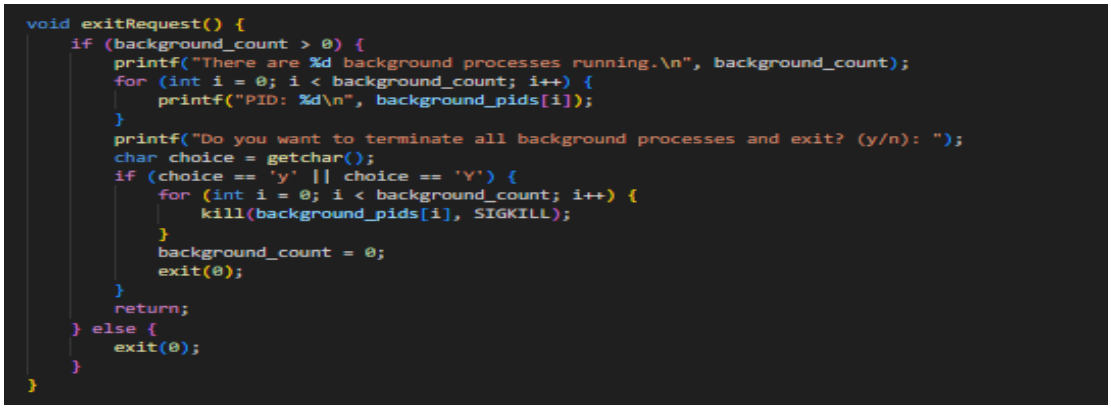
The moveBackgroundToForeground method brings a background process to the foreground for user interaction. It takes an integer argument for the process ID (PID). The method searches for the specified PID in the background-pids array, sets it as the foreground_pid, waits for the process to finish, and removes it from the background list.



```
void moveBackgroundToForeground(int pid) {
    for (int i = 0; i < background_count; i++) {
        if (background_pids[i] == pid) {
            foreground_pid = pid;
            waitpid(pid, NULL, 0);
            foreground_pid = 0;
            background_count--;
            background_pids[i] = background_pids[background_count];
            return;
        }
    }
    fprintf(stderr, "No such background process.\n");
}
```

exitRequest

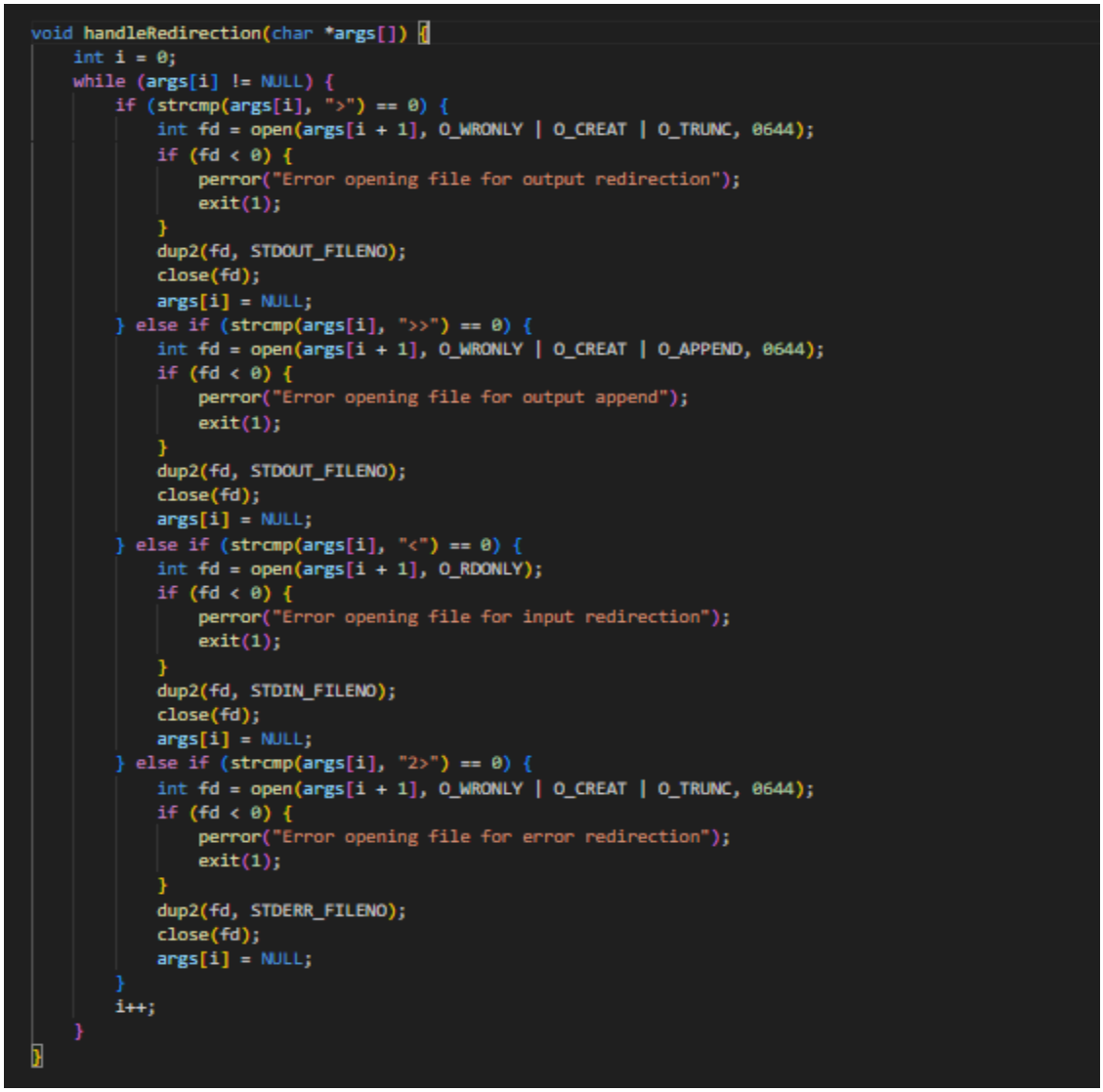
The exitRequest method manages shell termination and cleans up background processes. It does not require any arguments. The method checks for any running background processes, displays their PIDs, and prompts the user to terminate them if necessary. If the user agrees, it kills all background processes; if there are no background processes, it simply exits the shell.



```
void exitRequest() {
    if (background_count > 0) {
        printf("There are %d background processes running.\n", background_count);
        for (int i = 0; i < background_count; i++) {
            printf("PID: %d\n", background_pids[i]);
        }
        printf("Do you want to terminate all background processes and exit? (y/n): ");
        char choice = getchar();
        if (choice == 'y' || choice == 'Y') {
            for (int i = 0; i < background_count; i++) {
                kill(background_pids[i], SIGKILL);
            }
            background_count = 0;
            exit(0);
        }
        return;
    } else {
        exit(0);
    }
}
```

handleRedirection

The handleRedirection method manages input and output for commands. It takes a string array as an argument. The method searches the array for special operators like >, >>, <, and 2>, replaces the file descriptors with those of the specified files, and removes the redirection operators and file names. It utilizes system calls such as open, dup2, and close for file handling.



```
void handleRedirection(char *args[]) {
    int i = 0;
    while (args[i] != NULL) {
        if (strcmp(args[i], ">") == 0) {
            int fd = open(args[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (fd < 0) {
                perror("Error opening file for output redirection");
                exit(1);
            }
            dup2(fd, STDOUT_FILENO);
            close(fd);
            args[i] = NULL;
        } else if (strcmp(args[i], ">>") == 0) {
            int fd = open(args[i + 1], O_WRONLY | O_CREAT | O_APPEND, 0644);
            if (fd < 0) {
                perror("Error opening file for output append");
                exit(1);
            }
            dup2(fd, STDOUT_FILENO);
            close(fd);
            args[i] = NULL;
        } else if (strcmp(args[i], "<") == 0) {
            int fd = open(args[i + 1], O_RDONLY);
            if (fd < 0) {
                perror("Error opening file for input redirection");
                exit(1);
            }
            dup2(fd, STDIN_FILENO);
            close(fd);
            args[i] = NULL;
        } else if (strcmp(args[i], "2>") == 0) {
            int fd = open(args[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (fd < 0) {
                perror("Error opening file for error redirection");
                exit(1);
            }
            dup2(fd, STDERR_FILENO);
            close(fd);
            args[i] = NULL;
        }
        i++;
    }
}
```

terminateRunningProcess

The terminateRunningProcess method handles the termination of the current foreground process when the user sends a SIGTSTP signal. It takes an integer argument for the signal number. The method sends a SIGKILL signal to the active foreground process identified by foreground_pid and confirms its termination.

```
void terminateRunningProcess(int signum) {
    if (foreground_pid > 0) {
        kill(-foreground_pid, SIGKILL);
        printf("Foreground process %d and its group terminated.\n", foreground_pid);
        foreground_pid = 0;
    }
}
```

main

The main method serves as the core loop of the shell and does not accept any arguments. It manages several operations, including:

• Registering SIGTSTP to invoke the terminateRunningProcess method,

• Prompting the user for input,

• Parsing the input and calling the setup method,

• Displaying or executing a specific history entry through the history command,

• Terminating the shell after processing commands with the exit command,

• Bringing a background process to the foreground using fg,

• Forking a child process, and

• Keeping track of background processes in background_pids.

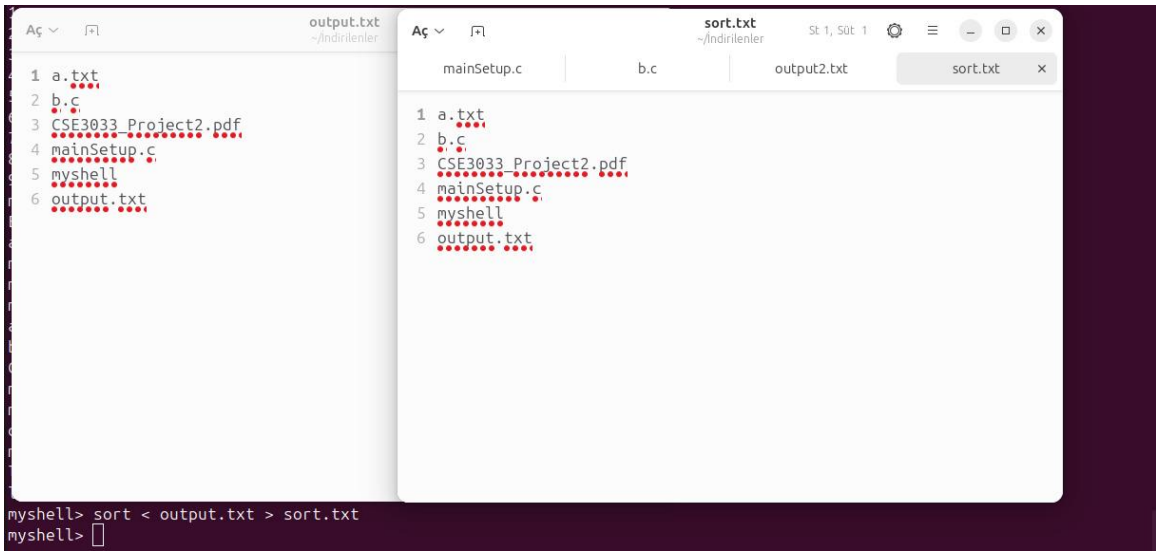
```c
int main(void) {
    char inputBuffer[MAX_LINE];
    int background;
    char *args[MAX_LINE / 2 + 1];
    signal(SIGTSTP, terminateRunningProcess);
    while (1) {
        background = 0;
        printf("myshell> ");
        fflush(stdout);
        setup(inputBuffer, args, &background);
        if (args[0] == NULL) continue;
        if (args[0] != NULL && strcmp(args[0], "history") != 0) {
            addToHistory(args);
        }
        if (strcmp(args[0], "history") == 0) {
            if (args[1] && strcmp(args[1], "-i") == 0) {
                if (args[2]) {
                    int index = atoi(args[2]);
                    historyCommand(index, inputBuffer, args, &background);
                    continue;
                } else {
                    fprintf(stderr, "Usage: history -i <index>\n");
                }
            } else {
                printHistory();
            }
            continue;
        } else if (strcmp(args[0], "exit") == 0) {
            exitRequest();
            continue;
        } else if (strcmp(args[0], "fg") == 0) {
            if (args[1] && args[1][0] == '%') {
                int pid = atoi(&args[1][1]);
                moveBackgroundToForeground(pid);
            } else {
                fprintf(stderr, "Usage: fg %%<pid>\n");
            }
            continue;
        }
        pid_t pid = fork();
        if (pid < 0) {
            perror("Fork failed");
            continue;
        }
        if (pid == 0) {···
        } else {
            if (!background) {
                foreground_pid = pid;
                waitpid(pid, NULL, 0);
                foreground_pid = 0;
            } else {
                background_pids[background_count++] = pid;
                printf("Background process started with PID %d\n", pid);
            }
        }
    }
}
```

Screenshots:

```
oguz@oguz-VMware-Virtual-Platform:~/İndirilenler$ gcc -o myshell mainSetup.c
oguz@oguz-VMware-Virtual-Platform:~/İndirilenler$ ./myshell
myshell> sleep 2
myshell> sleep 2 &
Background process started with PID 5496
```

```
-rw-rw-r-- 1 oguz oguz 134031 Ara 15 17:06 CSE3033_Project2.pdf
-rw-rw-r-- 1 oguz oguz  13625 Ara 15 21:09 mainSetup.c
-rwxrwxr-x 1 oguz oguz  21976 Ara 16 16:18 myshell
-rw-r--r-- 1 oguz oguz     74 Ara 16 16:21 output2.txt
-rw-r--r-- 1 oguz oguz     62 Ara 16 16:20 output.txt
-rw-r--r-- 1 oguz oguz     14 Ara 16 16:31 out.txt
-rw-r--r-- 1 oguz oguz     62 Ara 16 16:23 sort.txt
myshell> who
oguz     seat0        2024-12-16 14:31 (login screen)
oguz     tty2         2024-12-16 14:31 (tty2)
myshell> ps -a
    PID TTY          TIME CMD
   2331 tty2     00:00:00 gnome-session-b
   6643 pts/0    00:00:00 myshell
   7733 pts/0    00:00:00 ps
myshell> chmod 777 a.txt
myshell> history
0 chmod 777 a.txt
1 ps -a
2 who
3 ls -l
4 ls
5 ps
6 clear
7 echo World! >> out.txt
8 echo Hello, > out.txt
9 clear
myshell> history -i 4
Executing command from history: ls
a.txt  b.c  CSE3033_Project2.pdf  mainSetup.c  myshell  output2.txt  output.txt  out.txt  sort.txt
myshell> exit
oguz@oguz-VMware-Virtual-Platform:~/İndirilenler$
```

```
myshell> echo Hello, > out.txt
myshell> echo World! >> out.txt
myshell>
```


Aç
output.txt
~/İndirilenler
1 a.txt
2 b.c
3 CSE3033_Project2.pdf
4 mainSetup.c
5 myshell
6 output.txt
Aç
sort.txt
~/İndirilenler
St 1, Süt 1
mainSetup.c
b.c
output2.txt
sort.txt
1 a.txt
2 b.c
3 CSE3033_Project2.pdf
4 mainSetup.c
5 myshell
6 output.txt
myshell> sort < output.txt > sort.txt
myshell>