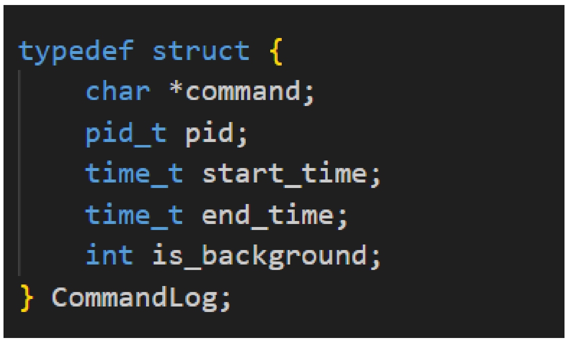
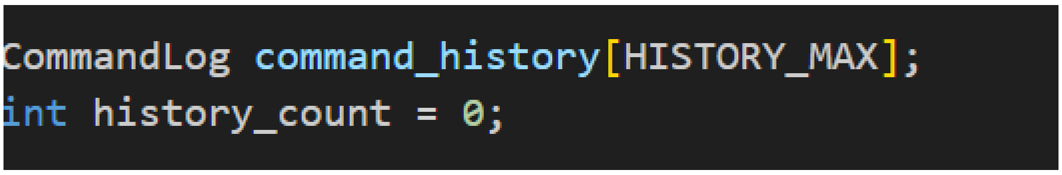
# Design document

## Explanation of code



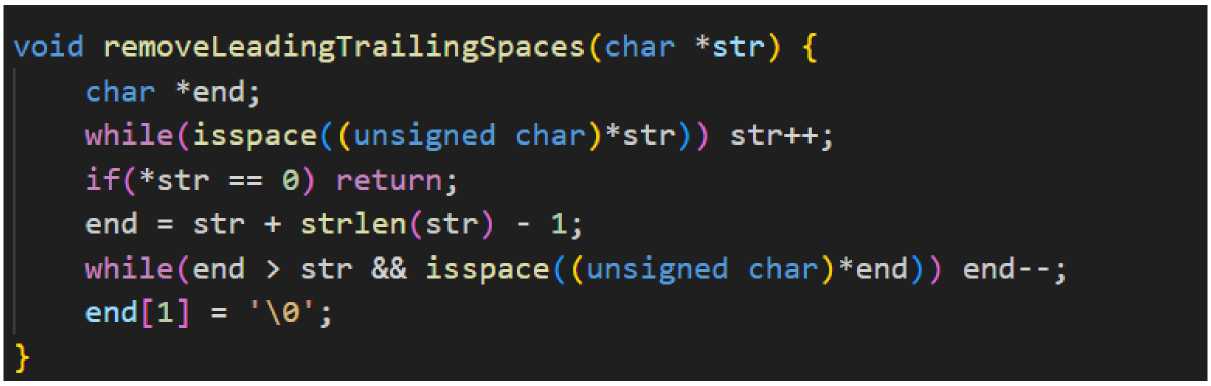
```c
typedef struct {
    char *command;
    pid_t pid;
    time_t start_time;
    time_t end_time;
    int is_background;
} CommandLog;
```

This defines a structure to log information about executed commands.



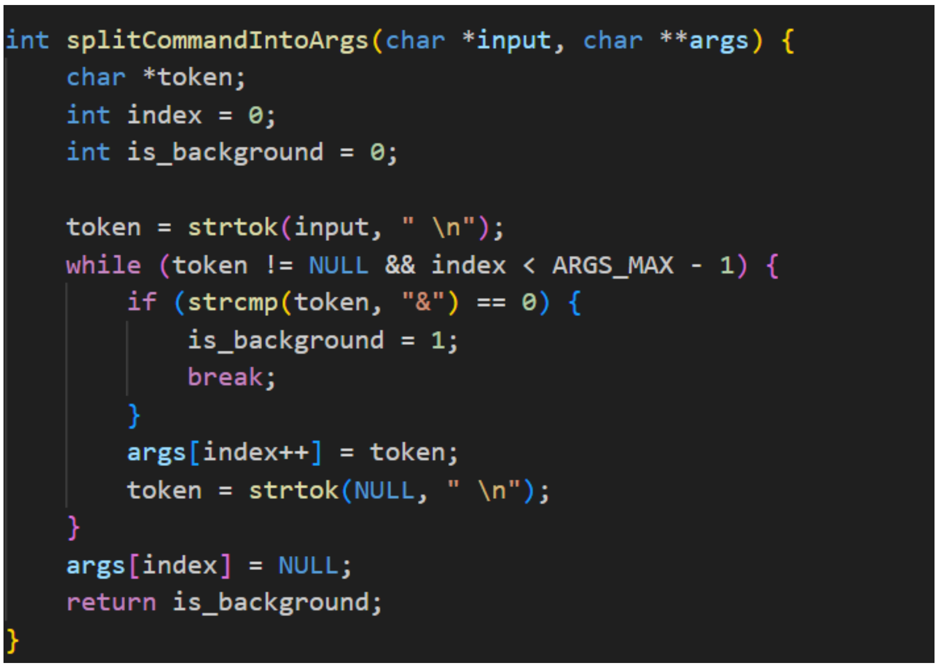
```c
CommandLog command_history[HISTORY_MAX];
int history_count = 0;
```

This creates an array to store command history and a counter for the number of commands.



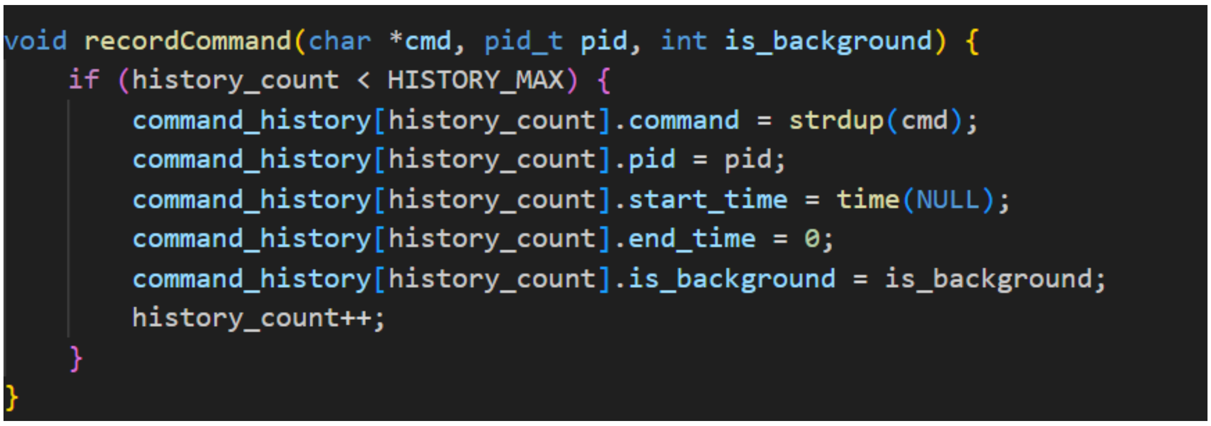
```c
void removeLeadingTrailingSpaces(char *str) {
    char *end;
    while(isspace((unsigned char)*str)) str++;
    if(*str == 0) return;
    end = str + strlen(str) - 1;
    while(end > str && isspace((unsigned char)*end)) end--;
    end[1] = '\0';
}
```

This function removes leading and trailing whitespace from a string.



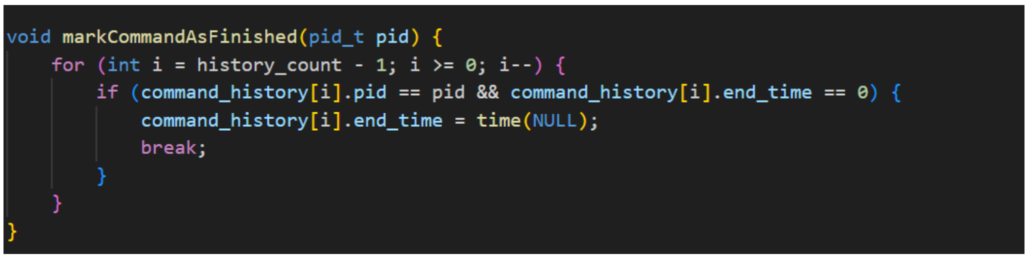
```c
int splitCommandIntoArgs(char *input, char **args) {
    char *token;
    int index = 0;
    int is_background = 0;

    token = strtok(input, " \n");
    while (token != NULL && index < ARGS_MAX - 1) {
        if (strcmp(token, "&") == 0) {
            is_background = 1;
            break;
        }
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL;
    return is_background;
}
```

This function splits the input string into arguments and checks for background execution.



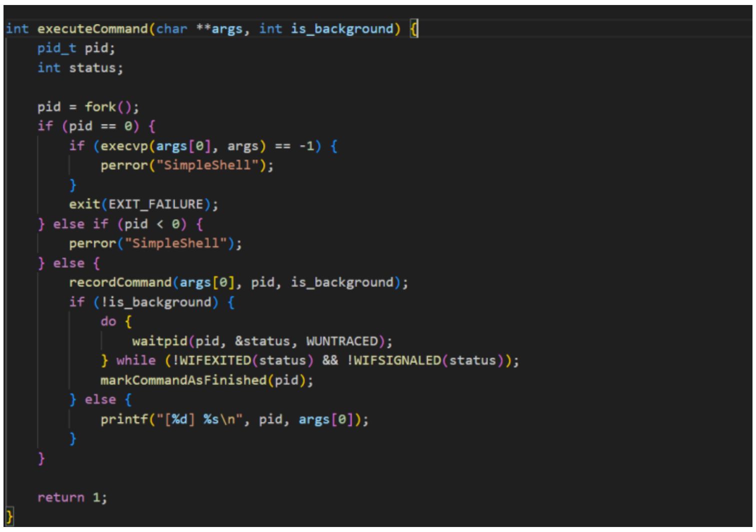
```c
void recordCommand(char *cmd, pid_t pid, int is_background) {
    if (history_count < HISTORY_MAX) {
        command_history[history_count].command = strdup(cmd);
        command_history[history_count].pid = pid;
        command_history[history_count].start_time = time(NULL);
        command_history[history_count].end_time = 0;
        command_history[history_count].is_background = is_background;
        history_count++;
    }
}
```

This function records information about executed commands in the history.



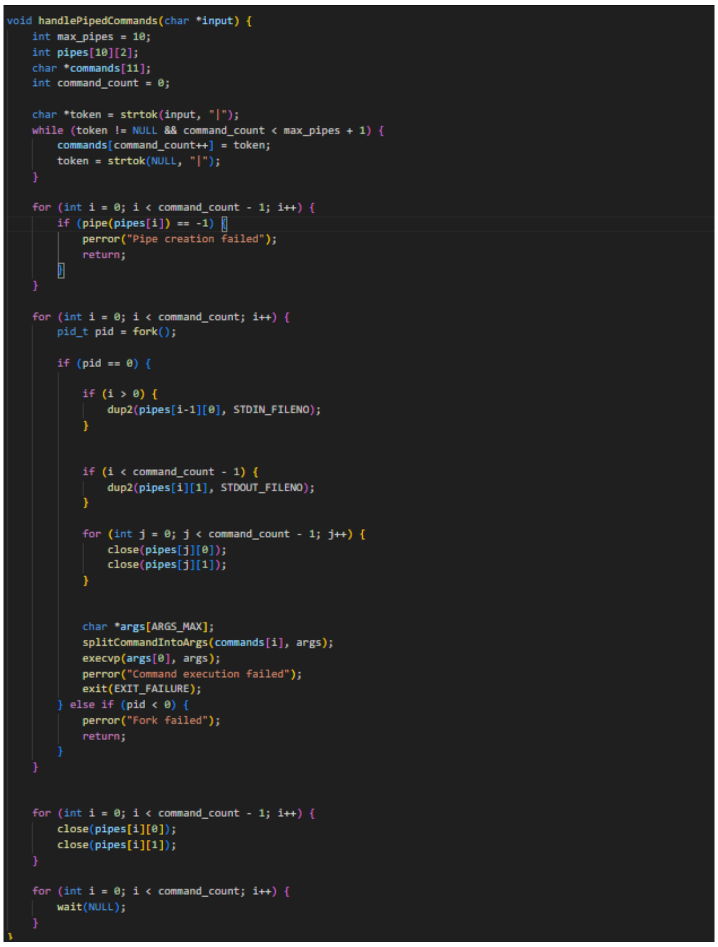
```c
void markCommandAsFinished(pid_t pid) {
    for (int i = history_count - 1; i >= 0; i--) {
        if (command_history[i].pid == pid && command_history[i].end_time == 0) {
            command_history[i].end_time = time(NULL);
            break;
        }
    }
}
```

This function updates the end time of a command in the log.



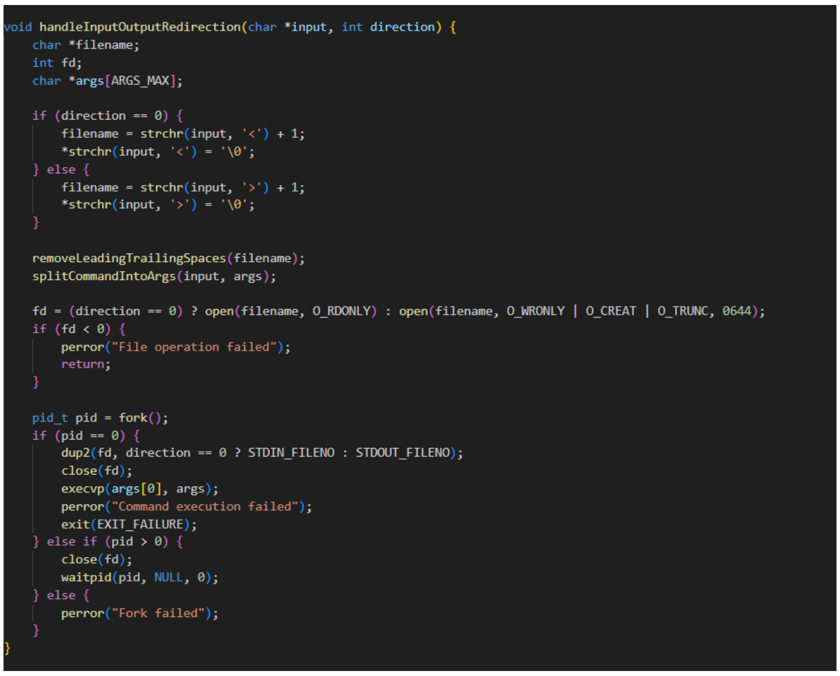
```c
int executeCommand(char **args, int is_background) {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        if (execvp(args[0], args) == -1) {
            perror("SimpleShell");
        }
        exit(EXIT_FAILURE);
    } else if (pid < 0) {
        perror("SimpleShell");
    } else {
        recordCommand(args[0], pid, is_background);
        if (!is_background) {
            do {
                waitpid(pid, &status, WUNTRACED);
            } while (!WIFEXITED(status) && !WIFSIGNALED(status));
            markCommandAsFinished(pid);
        } else {
            printf("[%d] %s\n", pid, args[0]);
        }
    }

    return 1;
}
```

This function forks a new process to execute a command, handling both foreground and background execution.



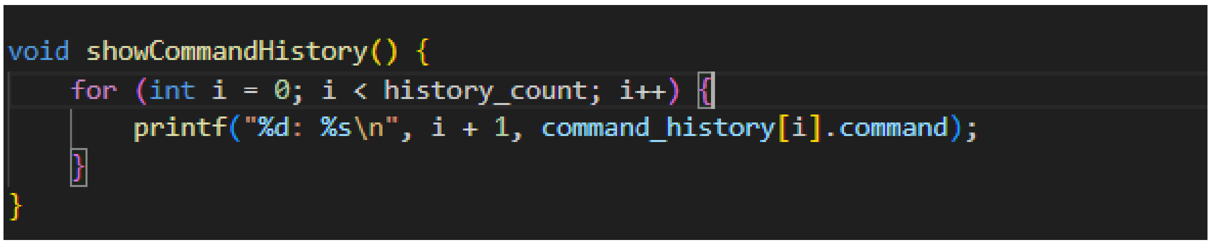
```c
void handlePipedCommands(char *input) {
    int max_pipes = 10;
    int pipes[10][2];
    char *commands[11];
    int command_count = 0;

    char *token = strtok(input, "|");
    while (token != NULL && command_count < max_pipes + 1) {
        commands[command_count++] = token;
        token = strtok(NULL, "|");
    }

    for (int i = 0; i < command_count - 1; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("Pipe creation failed");
            return;
        }
    }

    for (int i = 0; i < command_count; i++) {
        pid_t pid = fork();

        if (pid == 0) {

            if (i > 0) {
                dup2(pipes[i-1][0], STDIN_FILENO);
            }


            if (i < command_count - 1) {
                dup2(pipes[i][1], STDOUT_FILENO);
            }

            for (int j = 0; j < command_count - 1; j++) {
                close(pipes[j][0]);
                close(pipes[j][1]);
            }


            char *args[ARGS_MAX];
            splitCommandIntoArgs(commands[i], args);
            execvp(args[0], args);
            perror("Command execution failed");
            exit(EXIT_FAILURE);
        } else if (pid < 0) {
            perror("Fork failed");
            return;
        }
    }


    for (int i = 0; i < command_count - 1; i++) {
        close(pipes[i][0]);
        close(pipes[i][1]);
    }

    for (int i = 0; i < command_count; i++) {
        wait(NULL);
    }
}
```

This function implements piping between multiple commands.



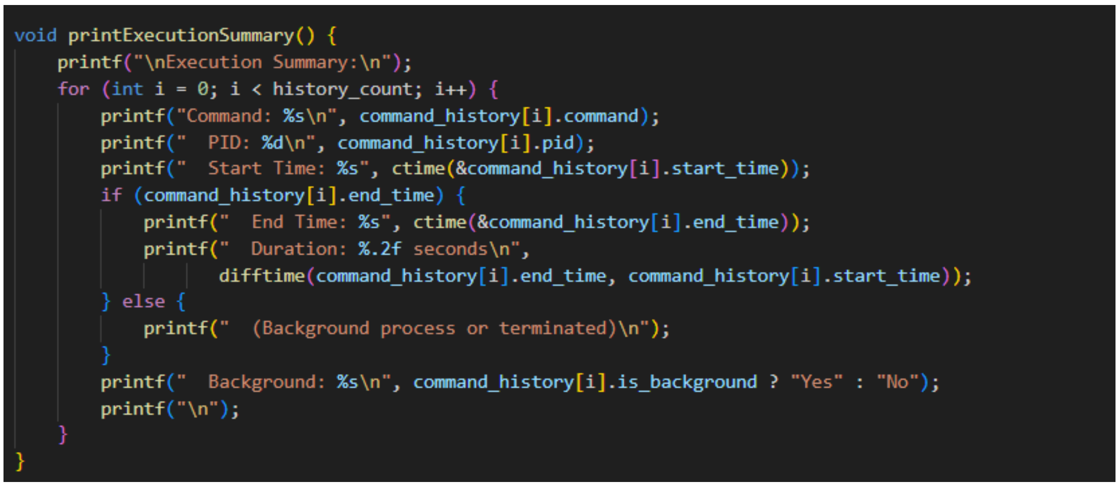
```c
void handleInputOutputRedirection(char *input, int direction) {
    char *filename;
    int fd;
    char *args[ARGS_MAX];

    if (direction == 0) {
        filename = strchr(input, '<') + 1;
        *strchr(input, '<') = '\0';
    } else {
        filename = strchr(input, '>') + 1;
        *strchr(input, '>') = '\0';
    }

    removeLeadingTrailingSpaces(filename);
    splitCommandIntoArgs(input, args);

    fd = (direction == 0) ? open(filename, O_RDONLY) : open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("File operation failed");
        return;
    }

    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd, direction == 0 ? STDIN_FILENO : STDOUT_FILENO);
        close(fd);
        execvp(args[0], args);
        perror("Command execution failed");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        close(fd);
        waitpid(pid, NULL, 0);
    } else {
        perror("Fork failed");
    }
}
```

This function handles input and output redirection.



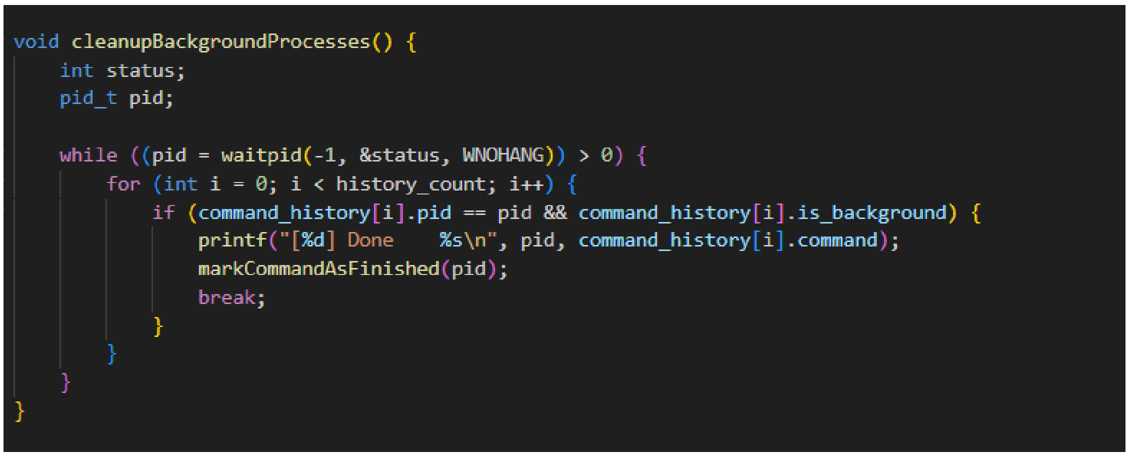
```c
void showCommandHistory() {
    for (int i = 0; i < history_count; i++) {
        printf("%d: %s\n", i + 1, command_history[i].command);
    }
}
```

This function displays the command history.



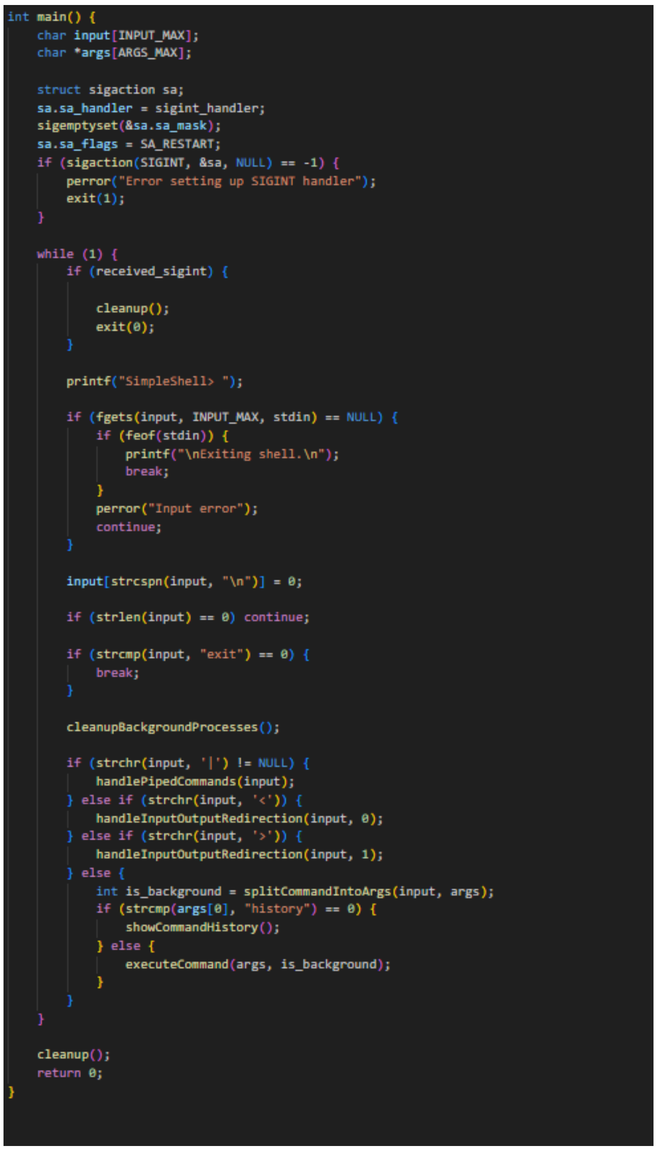
```c
void printExecutionSummary() {
    printf("\nExecution Summary:\n");
    for (int i = 0; i < history_count; i++) {
        printf("Command: %s\n", command_history[i].command);
        printf("  PID: %d\n", command_history[i].pid);
        printf("  Start Time: %s", ctime(&command_history[i].start_time));
        if (command_history[i].end_time) {
            printf("  End Time: %s", ctime(&command_history[i].end_time));
            printf("  Duration: %.2f seconds\n",
                   difftime(command_history[i].end_time, command_history[i].start_time));
        } else {
            printf("  (Background process or terminated)\n");
        }
        printf("  Background: %s\n", command_history[i].is_background ? "Yes" : "No");
        printf("\n");
    }
}
```

This function provides a detailed summary of executed commands.



```c
void cleanupBackgroundProcesses() {
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        for (int i = 0; i < history_count; i++) {
            if (command_history[i].pid == pid && command_history[i].is_background) {
                printf("[%d] Done    %s\n", pid, command_history[i].command);
                markCommandAsFinished(pid);
                break;
            }
        }
    }
}
```

This function checks and reports on completed background processes.



```c
int main() {
    char input[INPUT_MAX];
    char *args[ARGS_MAX];

    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Error setting up SIGINT handler");
        exit(1);
    }

    while (1) {
        if (received_sigint) {

            cleanup();
            exit(0);
        }

        printf("SimpleShell> ");

        if (fgets(input, INPUT_MAX, stdin) == NULL) {
            if (feof(stdin)) {
                printf("\nExiting shell.\n");
                break;
            }
            perror("Input error");
            continue;
        }

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

        if (strlen(input) == 0) continue;

        if (strcmp(input, "exit") == 0) {
            break;
        }

        cleanupBackgroundProcesses();

        if (strchr(input, '|') != NULL) {
            handlePipedCommands(input);
        } else if (strchr(input, '<')) {
            handleInputOutputRedirection(input, 0);
        } else if (strchr(input, '>')) {
            handleInputOutputRedirection(input, 1);
        } else {
            int is_background = splitCommandIntoArgs(input, args);
            if (strcmp(args[0], "history") == 0) {
                showCommandHistory();
            } else {
                executeCommand(args, is_background);
            }
        }
    }

    cleanup();
    return 0;
}
```

This is the main function that implements the shell's main loop. It repeatedly prompts for input, parses it, and executes the appropriate action based on the command.

Contribution – Both Anshul Rawat (2023104) and Darsh Gupta(2023185) contributed equally.

GitHub link repository - https://github.com/Anshul1734/OS-simple-shell

**Commands not working**

**1) cd -** It requires changing in shells internal state or environment which cant be done as we are running command in child process.

**2) Environment variable manipulation - export**

These involve manipulating the shell's environment, which again requires built in functionality rather than external commands.