CSE-344 Homework 1 Report

Süleyman Burak Yaşar
1901042662

Getting Input from User

A simple CLI was created using an infinite loop. User input was taken using fgets() and converted into a string.

```c
while(1){

        char string[100];
        printf("Command (type 'exit' to exit): ");
        char *check = fgets(string, 100, stdin);

        if (check == NULL)
        {
            perror("fgets");
            exit(1);
        }


        string[strlen(string) - 1] = '\0';
```

Match the user input with system commands

The command received from the user was compared with the existing commands in the system. If an existing command was entered, the arguments were divided into appropriate sections using a tokenizer for further processing.

```c
753   if(strncmp(string, "addStudentGrade", 15) == 0){
```

```c
824   else if(strncmp(string, "searchStudent", 13) == 0){
```

Tokenize the String

Using the ***divide_string()*** function, the string received from the user is split into tokens.

After being split into tokens, the number of tokens is checked. If the number of tokens does not match the number of arguments required by the entered command, an error is printed, and the system waits for input from the user again.

```c
int divide_string(char *string, char **tokens){

    char *token = strtok(string, " ");
    tokens[0] = token;
    int i = 1;
    while(token != NULL){
        token = strtok(NULL, "\"");
        if (token == NULL){
            break;
        }
        tokens[i] = token;
        token = strtok(NULL, "\"");
        i++;
    }
    return i;
}
```

Example of tokenize for ***sortAll*** command

sortAll “grades.txt” “name” “a”
tokens[0] = sortall
tokens[1] = grades.txt
tokens[2] = name
tokens[3] = a

```c

char **tokens = (char **)malloc(100 * sizeof(char *));
int num_of_tokens = divide_string(string, tokens);

if (num_of_tokens != 4){
    printf("Incorrect number of arguments! Expected: 3\n");
    free(tokens);
    logging(command, "fail", "Incorrect number of arguments! Expected: 3");
    free(command);
    continue;
}

char *file_name = tokens[1];
char *option = tokens[2];
char *order = tokens[3];
```

Fork the process

Once token processing is complete, a new child process is forked to execute the entered command. The parent process then pauses, awaiting the completion of the child process's task. This ensures that the command is executed in a separate process, allowing the parent to manage or control the execution flow as needed.

Screenshoot taken from
gtuStudentGrade “<file_name>” command

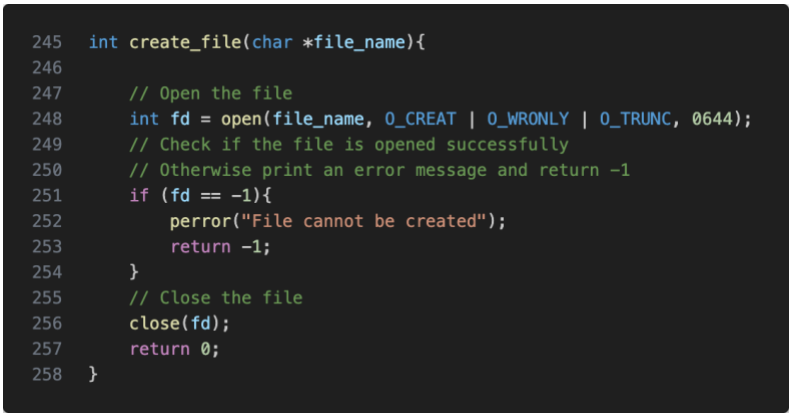
All commands are forked in the same manner, with the parent process waiting for the child process to complete its execution before continuing.

```c

pid_t pid = fork();

if (pid < 0){
    perror("fork");
    logging(command, "fail", "Fork failed");
    continue;
}

else if (pid == 0){
    int return_value = create_file(file_name);

    if (return_value == -1){
        printf("File cannot be created\n");
        logging(command, "fail", "File cannot be created");
        free(command);
        exit(1);
    }

    printf("File created\n");
    logging(command, "success", "File created");
    free(command);
    exit(0);
}

else{
    waitpid(-1, NULL, 0);
}
```

Command Processes

gtuStudentGrades “<file_name>”

A file is created using the file name. The ***`open`*** function is used to open the file with options for creation ***`O_CREAT`***, opening for writing ***`O_WRONLY`***, and truncating existing content if the file exists ***`O_TRUNC`***. If the file cannot be opened successfully, an error message is printed, and the function returns a value of ***`-1`***. If the file is opened successfully, it is closed with the ***`close`*** function, and the function successfully returns a value of ***`0`***.



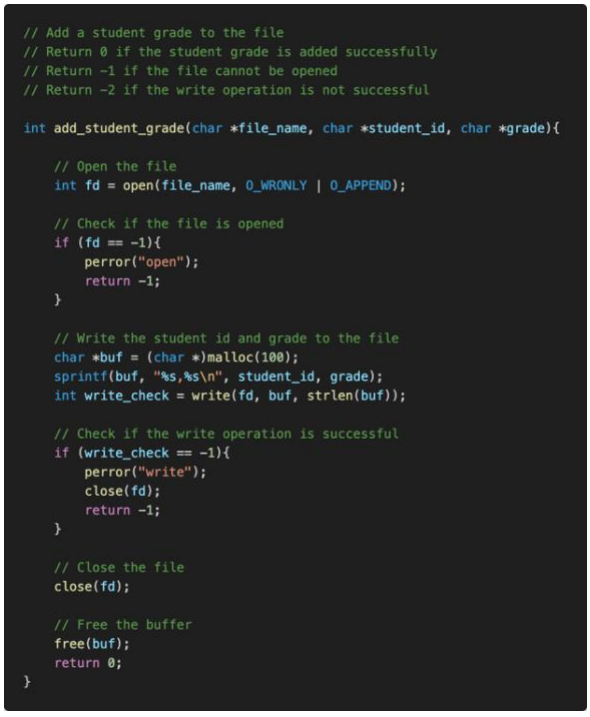
```c
int create_file(char *file_name){

    // Open the file
    int fd = open(file_name, O_CREAT | O_WRONLY | O_TRUNC, 0644);
    // Check if the file is opened successfully
    // Otherwise print an error message and return -1
    if (fd == -1){
        perror("File cannot be created");
        return -1;
    }
    // Close the file
    close(fd);
    return 0;
}
```

addStudentGrade "<file_name>" "<student_name>" "<student_grade>"

The provided ***`file_name`*** is used to add a new student grade using ***`student_id`*** and ***`grade`***. The file is attempted to be opened in write-only (***`O_WRONLY`***) and append (***`O_APPEND`***) modes. If the file cannot be successfully opened, an error message is printed, and the process is terminated by returning ***`-1`***.

Once the file is successfully opened, dynamic memory is allocated to create a text string (***`buf`***) containing the student ID and grade, and its content is formatted using the ***`sprintf`*** function. This text string is written to the file using the ***`write`*** function. If the writing process fails, an error message is printed, the file is closed, and the process is terminated by returning ***`-2`***.

If the writing process is successful, the opened file is closed, and the memory allocated for the created text string is freed. Finally, the process is completed by returning a ***`0`*** value to indicate that the student grade was successfully added to the file**.**



```c
// Add a student grade to the file
// Return 0 if the student grade is added successfully
// Return -1 if the file cannot be opened
// Return -2 if the write operation is not successful

int add_student_grade(char *file_name, char *student_id, char *grade){

    // Open the file
    int fd = open(file_name, O_WRONLY | O_APPEND);

    // Check if the file is opened
    if (fd == -1){
        perror("open");
        return -1;
    }

    // Write the student id and grade to the file
    char *buf = (char *)malloc(100);
    sprintf(buf, "%s,%s\n", student_id, grade);
    int write_check = write(fd, buf, strlen(buf));

    // Check if the write operation is successful
    if (write_check == -1){
        perror("write");
        close(fd);
        return -1;
    }

    // Close the file
    close(fd);

    // Free the buffer
    free(buf);
    return 0;
}
```

searchStudent “<file_name>” “<Name Surname>”

The file is opened in read mode using the file name provided by the user. If the file cannot be opened successfully, an error message is printed, and the process is terminated by returning a value of ***`-1`***. If the file is opened successfully, it reads character by character from the file to construct lines containing student information. Each line is converted into a student structure using the ***`create_student_from_line`*** function. If the name and surname of the student in this structure match the student being searched for, the student information is printed using the ***`print_student`*** function, and the function successfully ends by returning a value of ***`0`***. If the searched student is not found, the function returns a value of ***`-3`***. If an error occurs while reading the file, an error message is printed, and the function returns a value of ***`-2`***. After the operations are completed, the file is closed.

```c
// Search for a student's grade and display the student name and grade
// Return 0 if the student grade is found
// Return -1 if the file cannot be opened
// Return -2 if the read operation is not successful
// Return -3 if the student grade cannot be found


int search_student(char *file_name, char *student_name_surname){

    int fd = open(file_name, O_RDONLY);
    if (fd == -1){
        perror("open");
        return -1;
    }


    int n;
    char buffer[100];
    char line[100];
    int lineIndex = 0;
    while ((n = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        for (int i = 0; i < n; i++) {
            char c = buffer[i];
            if (c == '\n' || c == '\0') {
                line[lineIndex] = '\0'; // Null-terminate the line

                Student *student = create_student_from_line(line);
                if (strcmp(student->name, student_name_surname) == 0){
                    print_student(student);
                    free_student(student);
                    close(fd);
                    return 0;
                }

                free_student(student);


                lineIndex = 0;          // Reset the line index
                continue;
            }
            line[lineIndex++] = c; // Add the character to the line
        }
    }

    if (n < 0) {
        perror("Failed to read the file");
        return -2;
    }

    close(fd);
    return -3;
}
```

Student Datatype

To more efficiently handle entries recorded in a file, a ***`Student`*** structure was defined to store a student's name and grade. This structure and its auxiliary functions were created to facilitate the handling of student information.

```c
typedef struct Student{
    char *name; // Name and surname of the student
    char *grade; // Grade of the student
} Student;

// Create a new Student
Student *create_student(char *name, char *grade){
    Student *student = (Student *)malloc(sizeof(Student));
    student->name = strdup(name);
    student->grade = strdup(grade);
    return student;
}

// Create a new Student from line of the file
Student *create_student_from_line(char *line){
    char *token = strtok(line, ",");
    char *name = token;
    token = strtok(NULL, ",");
    char *grade = token;
    return create_student(name, grade);
}


// Free the memory of the StudentGrade
void free_student(Student *student){
    free(student->name);
    free(student->grade);
    free(student);
}

// Print the StudentGrade
void print_student(Student *student){
    printf("Name: %s, Grade: %s\n", student->name, student->grade);
}
```

create_student
The ***`create_student`*** function creates a new *`Student`* object with a given name and grade. The name and grade are copied into dynamic memory space (using ***`strdup`***), and this new student object is returned.

create_student_from_line
The ***`create_student_from_line`*** function analyzes a line read from a file. This line consists of two parts separated by a comma: the student's name and grade. The function separates this information and uses the ***`create_student`*** function to create a new ***`Student`*** object.

free_student
The ***`free_student`*** function releases the memory allocated for a ***`Student`*** object. This includes freeing the memory allocated for the student's name and grade, and finally, the memory allocated for the Student object itself.

The ***`print_student`*** function prints a student's name and grade in a simple format:
Name: [name]
Grade: [grade]

sortAll “<file_name>” “<option>” “<order>”

The file is opened in read mode using the file name provided by the user, and returns a value of ***`-1`*** with an error message in case of failure.

If the file opens successfully, it reads lines containing student information from the content and converts these lines into student objects using the ***`create_student_from_line`*** function, adding them to a student vector.

This student vector is sorted according to the specified option ***`option`*** and sorting order ***`order`*** using the ***`sort_student_vector`*** function.

After the sorting process is completed, the ***`print_student_vector`*** function prints the information of students in the student vector.

Once the operations are finished, the memory allocated for the student vector is released using the ***`free_student_vector`*** function, and the file is closed with the `close` function.

The function returns a value of ***`0`*** if the operations are successfully completed.

Options: ***grade:*** Sort the grades or ***name:*** Sort the student names
Order: ***a***:Sort the grades in ascending order or ***d***: Sort the grades in descending order

```c
// Sort the students in the file
// Return 0 if the students are sorted
// Return -1 if the file cannot be opened
// Return -2 if the read operation is not successful

int sort_student(char *file_name, char *option, char *order){

    int fd = open(file_name, O_RDWR);
    if (fd == -1){
        perror("open");
        return -1;
    }

    printf("option: %s, order: %s\n", option, order);

    // Read the file
    int n;
    char buffer[100];
    char line[100];
    int lineIndex = 0;

    // Create a StudentVector
    StudentVector *student_grades = create_student_vector();

    while ((n = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        for (int i = 0; i < n; i++) {
            char c = buffer[i];
            if (c == '\n' || c == '\0') {

                line[lineIndex] = '\0';

                // Add the student grade to the vector
                add_student_to_vector(student_grades, create_student_from_line(line));

                lineIndex = 0;
                continue;
            }
            // Add the character to the line
            line[lineIndex++] = c;
        }
    }

    if (n < 0) {
        perror("Failed to read the file");
        free_student_vector(student_grades);
        return -2;

    }

    // Sort the StudentVector
    sort_student_vector(student_grades, option, order);

    print_student_vector(student_grades);

    //free the memory of the StudentVector
    free_student_vector(student_grades);


    // Close the file
    close(fd);

    return 0;

}
```

StudentVector Datatype

This structure was created for the management of a dynamic array of student records, encapsulated within a ***`StudentVector`*** struct.

The struct tracks a list of student pointers (***`Student **students`***), the current number of students (***`size`***), and the allocated capacity (***`capacity`***) to efficiently manage memory and allow for dynamic resizing.

```c
typedef struct StudentVector{
    Student **students;
    int size;
    int capacity;
} StudentVector;
```

The key functionalities include creating, adding to, sorting, and deallocating the ***StudentVector:***

`create_student_vector` allocates memory for a ***`StudentVector`*** and initializes it with a capacity for 100 students, setting the initial size to 0. This setup prepares a dynamic array for storing pointers to student records.

```c
// Create a new StudentVector
StudentVector *create_student_vector(){
    StudentVector *student_vector = (StudentVector *)malloc(sizeof(StudentVector));
    student_vector->size = 0;
    student_vector->capacity = 100;
    student_vector->students = (Student **)malloc(student_vector->capacity * sizeof
    return student_vector;
}
```

`add_student_to_vector` involves adding a new ***`Student`*** to the vector. If the capacity is reached, it doubles the capacity and reallocates memory to accommodate more student pointers, ensuring the array can scale dynamically.

```c
// Add a new Student to the StudentVector
void add_student_to_vector(StudentVector *student_vector, Student *student){
    if (student_vector->size == student_vector->capacity){
        student_vector->capacity *= 2;
        student_vector->students = (Student **)realloc(student_vector->student
    }
    student_vector->students[student_vector->size] = student;
    student_vector->size++;
}
```

Comparison functions like ***`compare_grades_ascending`***, ***`compare_grades_descending`***, ***`compare_names_ascending`***, and ***`compare_names_descending`*** are employed by the ***`qsort`*** sorting algorithm. These functions enable the sorting of students by either grades or names in either ascending or descending order, offering flexibility in organizing student records.

```c
// Compare the grades in ascending order
// Type: Student
int compare_grades_ascending(const void *a, const void *b){
    Student *student_a = *(Student **)a;
    Student *student_b = *(Student **)b;
    return strcmp(student_a->grade, student_b->grade);
}

// Compare the grades in descending order
// Type: Student
int compare_grades_descending(const void *a, const void *b){
    Student *student_a = *(Student **)a;
    Student *student_b = *(Student **)b;
    return strcmp(student_b->grade, student_a->grade);
}

// Compare the names in ascending order
// Type: Student
int compare_names_ascending(const void *a, const void *b){
    Student *student_a = *(Student **)a;
    Student *student_b = *(Student **)b;
    return strcmp(student_a->name, student_b->name);
}

// Compare the names in descending order
// Type: Student

int compare_names_descending(const void *a, const void *b){
    Student *student_a = *(Student **)a;
    Student *student_b = *(Student **)b;
    return strcmp(student_b->name, student_a->name);
}
```

`sort_student_vector` leverages the ***`qsort`*** function from the C standard library to sort the students within the vector according to the specified ***`option`*** (grade or name) and ***`order`*** (ascending or descending). It dynamically selects the appropriate comparison function based on these criteria, illustrating the adaptability of sorting operations on the student vector.

```c
//Sort the StudentVector
void sort_student_vector(StudentVector *student_vector, char *option, char *order){
    if (strcmp(option, "grade") == 0){
        if (strcmp(order, "a") == 0){
            qsort(student_vector->students, student_vector->size, sizeof(Student *), compare_grades_ascending);
        }
        else if (strcmp(order, "d") == 0){
            qsort(student_vector->students, student_vector->size, sizeof(Student *), compare_grades_descending);
        }
    }
    else if (strcmp(option, "name") == 0){
        if (strcmp(order, "a") == 0){
            qsort(student_vector->students, student_vector->size, sizeof(Student *), compare_names_ascending);
        }
        else if (strcmp(order, "d") == 0){
            qsort(student_vector->students, student_vector->size, sizeof(Student *), compare_names_descending);
        }
    }
}
```

`free_student_vector` is responsible for deallocating all dynamically allocated memory associated with the ***`StudentVector`***, including the memory for each individual student and the array of student pointers. This function ensures the prevention of memory leaks by systematically freeing each student record, the pointer array, and finally, the ***`StudentVector`*** struct itself.

```c
// Free the memory of the StudentVector
void free_student_vector(StudentVector *student_vector){
    for (int i = 0; i < student_vector->size; i++){
        free_student(student_vector->students[i]);
    }
    free(student_vector->students);
    free(student_vector);
}
```

`print_student_vector` iterates through the student vector, displaying each student's information with the ***`print_student`*** function. This facilitates a straightforward method for listing the students and their respective grades.

```c
void print_student_vector(StudentVector *student_vector){
    for (int i = 0; i < student_vector->size; i++){
        print_student(student_vector->students[i]);
    }
}
```

showAll “<file_name>”

The given file is opened in read mode using the provided file name, and all student records in the file are displayed.

If the file cannot be opened successfully, an error message is printed, and the function terminates by returning a value of ***-1.***

Once the file is successfully opened, it is read line by line, and a ***Student*** object is created from each line using the ***create_student_from_line*** function. This student information is printed using the ***print_student*** function, and then the memory allocated for the student object is released using the ***free_student*** function.

If an error occurs while reading from the file, an error message is printed, and the function returns a value of ***-2***.

After all operations are successfully completed, the file is closed, and the function terminates by returning a value of ***0***, indicating successful completion.

```c
// Display all student grades stored in the file
// Return 0 if the student grades are displayed
// Return -1 if the file cannot be opened
// Return -2 if the read operation is not successful
int show_all(char *file_name){
    int fd = open(file_name, O_RDONLY);
    if (fd == -1){
        perror("open");
        return -1;
    }
    int n;
    char buffer[100];
    char line[100];
    int lineIndex = 0;

    while ((n = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        for (int i = 0; i < n; i++) {
            char c = buffer[i];
            if (c == '\n' || c == '\0') {
                line[lineIndex] = '\0'; // Null-terminate the line

                Student *student = create_student_from_line(line);
                print_student(student);
                free_student(student);

                lineIndex = 0;          // Reset the line index
                continue;
            }
            line[lineIndex++] = c; // Add the character to the line
        }
    }

    if (n < 0) {
        perror("Failed to read the file");
        return -2;
    }

    close(fd);
    return 0;
}
```

listGrades “<file_name>”

The file is opened in read mode using the file name provided by the user. The function reads and displays student records line by line from the file.

If the file cannot be opened successfully, an error message is printed, and the function terminates by returning a value of ***-1***.

After successfully opening the file, the function reads the file line by line. For each line, a ***`Student`*** object is created using the ***`create_student_from_line`*** function. The information of the created student object is printed using the ***`print_student`*** function, and then the memory allocated for this object is released using the ***`free_student`*** function.

This process is repeated for a maximum of 5 students.

After processing, the file is closed, and the function terminates by returning a value of ***0***.

If an error occurs while reading from the file, an error message is printed, the file is closed, and ***-2*** is returned.

If there are fewer than 5 student records in the file and the operation cannot be completed, and is returned ***-3***.

```c
int list_grades(char *file_name){

    int fd = open(file_name, O_RDONLY);

    if (fd == -1){
        perror("open");
        return -1;
    }

    int n;
    char buffer[100];
    char line[100];
    int lineIndex = 0;
    int j = 0;

    while ((n = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        for (int i = 0; i < n; i++) {
            char c = buffer[i];
            if (c == '\n' || c == '\0') {
                line[lineIndex] = '\0'; // Null-terminate the line

                Student *student = create_student_from_line(line);
                print_student(student);
                free_student(student);

                lineIndex = 0;          // Reset the line index
                j++;
                if (j == 5){
                    close(fd);
                    return 0;
                }
                continue;
            }
            line[lineIndex++] = c; // Add the character to the line
        }
    }

    if (n < 0) {
        perror("Failed to read the file");
        close(fd);
        return -2;
    }

    if( j < 5){
        close(fd);
        return -3;
    }

    close(fd);
    return 0;
}
```

listSome “<file_name>” “<numofEntries>” “<pageNumber>”

Entries within a specific range in a file are listed according to the number of entries ***`num_of_entries`*** and page number ***`page_number`*** parameters specified by the user. The file is opened in read mode; if this fails, an error message is printed and it terminates by returning ***`-1`***.

After the file is successfully opened, it is read line by line, and these lines are converted into student entries. Only entries within a certain range are processed, based on the ***`page_number`*** and ***`num_of_entries`*** parameters. This range starts at ***`(page_number * 5 - 1)`*** and ends at ***`(first + num_of_entries)`.***

If the entries within the specified range are read, the student information is printed, and then freed from memory. After all the necessary entries are successfully listed, the function terminates by returning ***`0`***. If an error occurs during reading from the file, an error message is printed and ***`-2`*** is returned. If there are not enough entries in the file (i.e., it is not possible to reach the specified ***`last`*** index), it terminates by returning ***`-3`***. Finally, the file is closed, and this closure action is performed in any case.

```c
// List the entries of the file
// Return 0 if the student grades are listed
// Return -1 if the file cannot be opened
// Return -2 if the read operation is not successful
// Return -3 if there are not enough entries in the file

int list_some(char *file_name, int num_of_entries, int page_number){
    int fd = open(file_name, O_RDONLY);
    if (fd == -1){
        perror("open");
        return -1;
    }
    int n;
    char buffer[100];
    char line[100];
    int lineIndex = 0;
    int first = page_number * 5 - 1;
    int last = first + num_of_entries;
    int j = 0;

    while ((n = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        for (int i = 0; i < n; i++) {
            char c = buffer[i];
            if (c == '\n' || c == '\0') {
                line[lineIndex] = '\0'; // Null-terminate the line

                if (j >= first && j < last){
                    Student *student = create_student_from_line(line);
                    print_student(student);
                    free_student(student);
                }

                if (j >= last){
                    close(fd);
                    return 0;
                }

                lineIndex = 0;          // Reset the line index
                j++;
                continue;
            }
            line[lineIndex++] = c; // Add the character to the line
        }
    }

    if (n < 0) {
        perror("Failed to read the file");
        return -2;
    }

    if( j < last){
        close(fd);
        return -3;
    }

    close(fd);
    return -1;
}
```

gtuStudentGrades

The user able to display all of the available commands by calling gtuStudentGrades without an argument.

Logging

The ***`logging`*** function is called at the end of each command processing, regardless of whether the command succeeds or fails. It takes three parameters:

`command`: The command received from the user.
`status`: Indicates whether the operation was successful or not.
`result`: The result of the operation.

Here's how the function works:

`time(&t)`: Retrieves the current time and stores it in the variable ***`t`***.
`fork()`: Creates a copy of the current process.

In the child process:

Opens the file ***"app.log"*** in write-only mode ***(`O_WRONLY`)***, appends to the end of the file ***(`O_APPEND`)***, and creates the file if it doesn't exist ***(`O_CREAT`)***. If opening the file fails, it prints an error message and exits the child process with ***`exit(1)`***.

```c
int logging(char *command, char *status, char *result){

    time_t t;   // not a primitive datatype
    time(&t);


    pid_t pid = fork();

    if (pid < 0){
        perror("fork");
        return -1;
    }

    else if (pid == 0){

        int fd = open("app.log", O_WRONLY | O_APPEND | O_CREAT, 0644);

        if (fd == -1){
            perror("Log file cannot be opened");
            exit(1);
        }

        char *buf = (char *)malloc(200);
        sprintf(buf, "%s,%s,%s,%s\n", command, status, result, ctime(&t));
        free(command);
        int write_check = write(fd, buf, strlen(buf));

        free(buf);

        if (write_check == -1){
            perror("write");
            close(fd);
            exit(1);
        }

        close(fd);
        exit(0);
    }
    else{
        waitpid(-1, NULL, 0);
        return 0;
    }

    return 0;
}
```

Formats the ***`command`***, ***`status`***, ***`result`***, and system time using ***`sprintf`*** into a buffer and writes it to the file.

If the writing operation fails, it prints an error message and exits the child process with ***`exit(1)`***.

Upon successful writing, dynamically allocated memory for ***`command`*** and ***`buf`*** is freed using ***`free()`***, and the file is closed with ***`close(fd)`***. Subsequently, the child process exits successfully with ***`exit(0)`***.

The parent process waits for the child process to complete with ***`waitpid`*** and then terminates successfully by returning ***`0`***.

```
if (num_of_tokens != 2){
    printf("Incorrect number of arguments! Expected: 1\n");
    free(tokens);
    logging(command, "fail", "Incorrect number of arguments! Expected: 1");
    continue;
}
```

```
if (return_value == -1){
    printf("File cannot be created\n");
    logging(command, "fail", "File cannot be created");
    free(command);
    exit(1);
}
```

```
printf("File created\n");
logging(command, "success", "File created");
free(command);
exit(0);
```

Test Part

There is a file named input.md in the ***/additional_folder***. It contains sample inputs along with their explanations. The test was conducted using the inputs from this file. All ***.txt*** files within, including ***app.log***, are related to the test.

gtuStudentGrades "<file_name>"

Valid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): gtuStudentGrades "computer_science_students.txt"
File created
Command (type 'exit' to exit): gtuStudentGrades "2024_spring.txt"
File created
```

Log File

```
gtuStudentGrades "computer_science_students.txt",success,File created,Wed Mar 20 22:26:06 2024
gtuStudentGrades "2024_spring.txt",success,File created,Wed Mar 20 22:26:11 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       23978  0.0  0.0   2880  1408 pts/1    S+   22:29   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output:

```
Command (type 'exit' to exit): gtuStudentGrades "computer_science_students.txt" "aa"
Incorrect number of arguments! Expected: 1
```

Log File

```
gtuStudentGrades "computer_science_students.txt" "aa",fail,Incorrect number of arguments! Expected: 1,Wed Mar 20 22:27:36 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       23981  0.0  0.0   2880  1536 pts/1    S+   22:31   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

addStudentGrade “<file_name>” “<Name Surname>” “<grade>”

Valid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt" "John Smith" "BA"
Student grade added
Command (type 'exit' to exit): addStudentGrade "2024_spring.txt" "Emily Davis" "CB"
Student grade added
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt" "Michael Brown" "AA"
Student grade added
Command (type 'exit' to exit):
```

Log File

```
addStudentGrade "computer_science_students.txt" "John Smith" "BA",success,Student grade added,Wed Mar 20 22:35:04 2024
addStudentGrade "2024_spring.txt" "Emily Davis" "CB",success,Student grade added,Wed Mar 20 22:35:11 2024
addStudentGrade "computer_science_students.txt" "Michael Brown" "AA",success,Student grade added,Wed Mar 20 22:35:18 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24006  0.0  0.0   2880  1536 pts/1    S+   22:37   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt" "John Doe" "AA" "aa"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt" "John Doe"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): addStudentGrade "computer_science_students.txt" "AA"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): addStudentGrade
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit):
```

Log File

```
addStudentGrade "computer_science_students.txt" "John Doe" "AA" "aa",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 22:41:44 2024
addStudentGrade "computer_science_students.txt",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 22:42:00 2024
addStudentGrade "computer_science_students.txt" "John Doe",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 22:42:06 2024
addStudentGrade "computer_science_students.txt" "AA",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 22:42:11 2024
addStudentGrade,fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 22:42:14 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24104  0.0  0.0   2880  1536 pts/1    S+   22:45   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

searchStudent “<file_name>” “<Name Surname>”

Valid Input:

Terminal input and output:

```
Command (type 'exit' to exit): searchStudent "computer_science_students.txt" "John Smith"
Name: John Smith, Grade: BA
Command (type 'exit' to exit): searchStudent "2024_spring.txt" "Emily Davis"
Name: Emily Davis, Grade: CB
Command (type 'exit' to exit): searchStudent "computer_science_students.txt" "Michael Brown"
Name: Michael Brown, Grade: AA
Command (type 'exit' to exit): searchStudent "2024_spring.txt" "Jessica Taylor"
Name: Jessica Taylor, Grade: BB
Command (type 'exit' to exit): searchStudent "computer_science_students.txt" "Daniel Miller"
Name: Daniel Miller, Grade: CC
Command (type 'exit' to exit):
```

Log File

```
searchStudent "computer_science_students.txt" "John Smith",success,Student grade found,Wed Mar 20 22:49:29 2024
searchStudent "2024_spring.txt" "Emily Davis",success,Student grade found,Wed Mar 20 22:49:35 2024
searchStudent "computer_science_students.txt" "Michael Brown",success,Student grade found,Wed Mar 20 22:49:41 2024
searchStudent "2024_spring.txt" "Jessica Taylor",success,Student grade found,Wed Mar 20 22:49:48 2024
searchStudent "computer_science_students.txt" "Daniel Miller",success,Student grade found,Wed Mar 20 22:49:54 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24117  0.0  0.0   2880  1536 pts/1    S+   22:51   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Valid Input but not Found:

Terminal input and output:

```
Command (type 'exit' to exit): searchStudent "computer_science_students.txt" "Unknown Student"
Student grade cannot be found
Command (type 'exit' to exit): searchStudent "2024_spring.txt" "Unknown Student"
Student grade cannot be found
Command (type 'exit' to exit):
```

Log File

```
searchStudent "computer_science_students.txt" "Unknown Student",fail,Student grade cannot be found,Wed Mar 20 22:52:15 2024
searchStudent "2024_spring.txt" "Unknown Student",fail,Student grade cannot be found,Wed Mar 20 22:52:21 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24124  0.0  0.0   2880  1408 pts/1    S+   22:53   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): searchStudent "computer_science_students.txt" "John Doe" "AA"
Incorrect number of arguments! Expected: 2
Command (type 'exit' to exit): searchStudent "computer_science_students.txt"
Incorrect number of arguments! Expected: 2
```

Log File

```
searchStudent "computer_science_students.txt" "John Doe" "AA",fail,Incorrect number of arguments! Expected: 2,Wed Mar 20 22:54:29 2024
searchStudent "computer_science_students.txt",fail,Incorrect number of arguments! Expected: 2,Wed Mar 20 22:54:36 2024
```

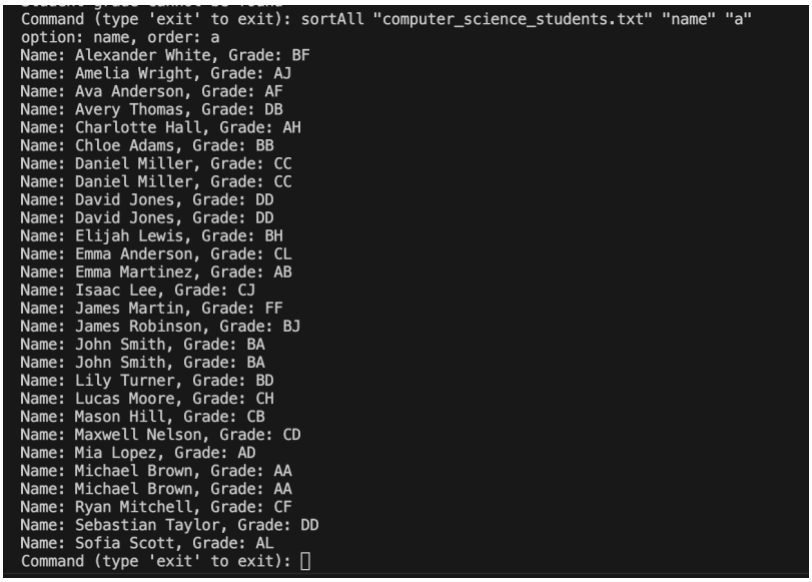
Zombie Process Control

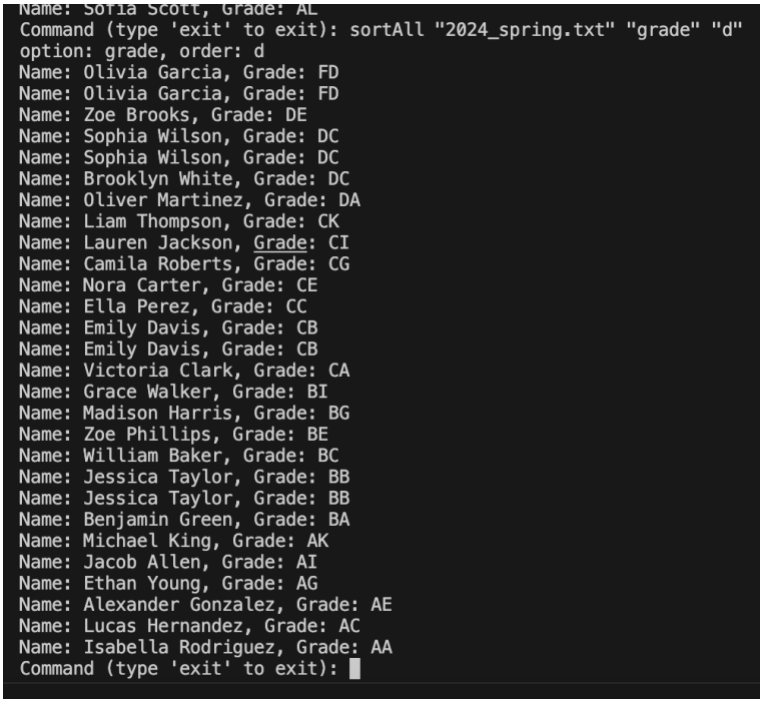
```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       24131  0.0  0.0   2880  1408 pts/1    S+   22:55   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

sortAll "<file_name>" "<name|grade>" "<a|d>"

Valid Input:

Terminal input and output**:**





Log File

```
sortAll "computer_science_students.txt" "name" "a",success,Student grades sorted,Wed Mar 20 22:56:45 2024
sortAll "2024_spring.txt" "grade" "d",success,Student grades sorted,Wed Mar 20 22:59:07 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24138  0.0  0.0   2880  1536 pts/1    S+   23:00   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): sortAll "computer_science_students.txt" "name" "a" "d"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): sortAll "computer_science_students.txt" "name" "aa"
Invalid order
Command (type 'exit' to exit): sortAll
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit):
```

Log File

```
sortAll "computer_science_students.txt" "name" "a" "d",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 23:01:02 2024
sortAll "computer_science_students.txt" "name" "aa",fail,Invalid order,Wed Mar 20 23:01:11 2024
sortAll,fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 23:01:21 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24144  0.0  0.0   2880  1536 pts/1    S+   23:02   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

showAll "<file_name>"

Valid Input:

Terminal input and output:

```
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): showAll "computer_science_students.txt"
tokens[0]: showAll
tokens[1]: computer_science_students.txt
Name: John Smith, Grade: BA
Name: Michael Brown, Grade: AA
Name: Daniel Miller, Grade: CC
Name: David Jones, Grade: DD
Name: John Smith, Grade: BA
Name: Michael Brown, Grade: AA
Name: Daniel Miller, Grade: CC
Name: David Jones, Grade: DD
Name: James Martin, Grade: FF
Name: Emma Martinez, Grade: AB
Name: Mia Lopez, Grade: AD
Name: Ava Anderson, Grade: AF
Name: Charlotte Hall, Grade: AH
Name: Amelia Wright, Grade: AJ
Name: Sofia Scott, Grade: AL
Name: Chloe Adams, Grade: BB
Name: Lily Turner, Grade: BD
Name: Alexander White, Grade: BF
Name: Elijah Lewis, Grade: BH
Name: James Robinson, Grade: BJ
Name: Mason Hill, Grade: CB
Name: Maxwell Nelson, Grade: CD
Name: Ryan Mitchell, Grade: CF
Name: Lucas Moore, Grade: CH
Name: Isaac Lee, Grade: CJ
Name: Emma Anderson, Grade: CL
Name: Avery Thomas, Grade: DB
Name: Sebastian Taylor, Grade: DD
Command (type 'exit' to exit):
```

Log File

```
showAll "computer_science_students.txt",success,Student grades displayed,Wed Mar 20 23:03:12 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24149  0.0  0.0   2880  1408 pts/1    S+   23:05   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): showAll "computer_science_students.txt" "argument2"
Incorrect number of arguments! Expected: 1
Command (type 'exit' to exit): showAll
Incorrect number of arguments! Expected: 1
Command (type 'exit' to exit):
```

Log File

```
showAll "computer_science_students.txt" "argument2",fail,Incorrect number of arguments! Expected: 1,Wed Mar 20 23:06:24 2024
showAll ,fail,Incorrect number of arguments! Expected: 1,Wed Mar 20 23:06:29 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24156  0.0  0.0   2880  1536 pts/1    S+   23:07   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

listGrades "<file_name>"

Valid Input:

Terminal input and output:

```
Command (type 'exit' to exit): listGrades "computer_science_students.txt"
num_of_tokens: 2
Name: John Smith, Grade: BA
Name: Michael Brown, Grade: AA
Name: Daniel Miller, Grade: CC
Name: David Jones, Grade: DD
Name: John Smith, Grade: BA
Command (type 'exit' to exit): listGrades "2024_spring.txt"
num_of_tokens: 2
Name: Emily Davis, Grade: CB
Name: Jessica Taylor, Grade: BB
Name: Sophia Wilson, Grade: DC
Name: Olivia Garcia, Grade: FD
Name: Emily Davis, Grade: CB
Command (type 'exit' to exit):
```

Log File

```
listGrades "computer_science_students.txt",success,Student grades listed,Wed Mar 20 23:08:22 2024
listGrades "2024_spring.txt",success,Student grades listed,Wed Mar 20 23:08:47 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24163  0.0  0.0   2880  1280 pts/1    S+   23:10   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output:

```
Command (type 'exit' to exit): listGrades "small.txt"
num_of_tokens: 2
Name: Emily Davis, Grade: CB
There are less than 5 entries in the file
Command (type 'exit' to exit):
```

```
Command (type 'exit' to exit): listGrades "computer_science_students.txt" "argument2"
num_of_tokens: 3
Incorrect number of arguments! Expected: 1
Command (type 'exit' to exit): listGrades
num_of_tokens: 1
Incorrect number of arguments! Expected: 1
Command (type 'exit' to exit):
```

Log File

```
listGrades "small.txt",fail,There are less than 5 entries in the file,Wed Mar 20 23:11:20 2024
listGrades "computer_science_students.txt" "argument2",fail,Incorrect number of arguments! Expected: 1,Wed Mar 20 23:12:44 2024
listGrades ,fail,Incorrect number of arguments! Expected: 1,Wed Mar 20 23:12:48 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24170  0.0  0.0   2880  1536 pts/1    S+   23:14   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

listSome "<file_name>" "<num_of_entries>" "page_number"

Valid Input:

Terminal input and output:

```
Command (type 'exit' to exit): listSome "computer_science_students.txt" "5" "1"
Name: John Smith, Grade: BA
Name: Michael Brown, Grade: AA
Name: Daniel Miller, Grade: CC
Name: David Jones, Grade: DD
Name: James Martin, Grade: FF
Command (type 'exit' to exit): listSome "2024_spring.txt" "3" "2"
Name: Lucas Hernandez, Grade: AC
Name: Alexander Gonzalez, Grade: AE
Name: Ethan Young, Grade: AG
Command (type 'exit' to exit):
```

Log File

```
listSome "computer_science_students.txt" "5" "1",success,Student grades listed,Wed Mar 20 23:17:16 2024
listSome "2024_spring.txt" "3" "2",success,Student grades listed,Wed Mar 20 23:17:25 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24177  0.0  0.0   2880  1408 pts/1    S+   23:18   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Input:

Terminal input and output:

```
Command (type 'exit' to exit): listSome "small.txt" "10" "5"
There are not enough entries in the file
Command (type 'exit' to exit): listSome "computer_science_students.txt" "5" "1" "argument4"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): listSome "computer_science_students.txt" "5"
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit): listSome
Incorrect number of arguments! Expected: 3
Command (type 'exit' to exit):
```

Log File

```
listSome "small.txt" "10" "5",fail,There are not enough entries in the file,Wed Mar 20 23:19:05 2024
listSome "computer_science_students.txt" "5" "1" "argument4",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 23:19:14 2024
listSome "computer_science_students.txt" "5",fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 23:19:19 2024
listSome,fail,Incorrect number of arguments! Expected: 3,Wed Mar 20 23:19:26 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24185  0.0  0.0   2880  1280 pts/1    S+   23:20   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

gtuStudentGrades

Valid Input:

Terminal input and output**:**

```
Command (type 'exit' to exit): gtuStudentGrades

Available commands:

-----------------------------------------

gtuStudentGrades

Description: Display all of the available commands

Example Usage: gtuStudentGrades

-----------------------------------------

gtuStudentGrades "<file_name>"

Description: Create a new file

Example Usage: gtuStudentGrades "grades.txt"

-----------------------------------------

addStudentGrade "<file_name>" "<student_name_surname>" "<grade>"

Description: Add a new student's grade to the system

Example Usage: addStudentGrade "grades.txt" "John Doe" "AA"

-----------------------------------------
```

Log File

```
gtuStudentGrades,success,Display all of the available commands,Wed Mar 20 23:27:03 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       24203  0.0  0.0   2880  1408 pts/1    S+   23:28   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

exit

Terminal input and output:

```
-----------------------------------------

Command (type 'exit' to exit): exit
root@315e18c77eda:/workspace#
root@315e18c77eda:/workspace#
root@315e18c77eda:/workspace#
root@315e18c77eda:/workspace#
```

Log File

```
exit,success,Exit the program,Wed Mar 20 23:29:39 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24207  0.0  0.0   2880  1408 pts/1    S+   23:30   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Invalid Command

Terminal input and output:

```
Command (type 'exit' to exit): Unknown_Command
Invalid command
```

Log File

```
Unknown_Command,fail,Invalid command,Wed Mar 20 23:31:49 2024
```

Zombie Process Control

```
root@315e18c77eda:/workspace# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     24213  0.0  0.0   2880  1408 pts/1    S+   23:32   0:00 grep --color=auto Z
root@315e18c77eda:/workspace#
```

Usage Of Makefile

make*:* Compiles the code
run: Run the program
run2: Run the program with valgrind
clean: Delete only the program
clean2: Delete all .txt file, .log file and the program.

Additional Information

All tests were conducted using Docker. It was also re-tested on a computer with Ubuntu 22.04 installed. Just in case, I'm including the Docker file inside the document as well.