Name – Shivranjan Y. Pathak

College - MITWPU

Name of the Project : Customized Virtual File System

Technology Used

- C programming language

User Interface

- Command Line Interface (CLI)

Platform Required

- Linux-based operating system (e.g., Ubuntu, Kali Linux),Windows

---

Hardware Requirements

- Minimum 4GB RAM
- Minimum 2GHz dual-core processor
- 100MB free disk space for development environment and project files

---

Project Description

Overview

The Customized Virtual File System (CVFS) is an implementation of a file management system that emulates the functionalities of a traditional file system. It allows users to create, read, write, open, close, and delete files. Additionally, it provides mechanisms to manipulate and retrieve file metadata and content. The VFS is designed to help users understand the basic operations of a file system in an educational context.

Key Features

1. **File Operations**:
   - **Create**: Allows the creation of new regular files with specified permissions.
   - **Open**: Opens an existing file in a specified mode (read, write, or read/write).
   - **Close**: Closes an open file by its name or file descriptor.
   - **Close All**: Closes all open files.
   - **Read**: Reads data from a file into a buffer.
   - **Write**: Writes data from a buffer into a file.
   - **Delete (rm)**: Deletes a file by its name.
   - **Truncate**: Removes all data from a file without deleting the file itself.
2. **Metadata Operations**:
   - **stat**: Displays metadata information about a file by its name.
   - **fstat**: Displays metadata information about a file by its file descriptor.
   - **ls**: Lists all files along with their inode numbers, sizes, and link counts.
3. **File Positioning**:
   - **lseek**: Changes the read/write offset of an open file based on a specified position (start, current, or end).
4. **User Assistance**:
   - **man**: Provides a manual for various commands describing their usage and functionality.
   - **help**: Displays a list of available commands and their descriptions.

Components

1. **Inode Structure**:
   - Represents each file with attributes like file name, inode number, file size, actual size, file type, buffer for file data, link count, reference count, permissions, and a pointer to the next inode.Every file contains its unique inode.
2. **Superblock Structure**:
   - Maintains the overall file system status, including total and free inodes.
3. **File Table Structure**:
   - Manages information about open files, including read/write offsets, mode, count, and a pointer to the corresponding inode.

     i. **Mode** – It contains the mode in which we open the file.
     ii. **Offset** – It's a point from where we want to Read or Write the Data.
     iii. **Pointer** – Which points to the specific entry from incore inode table,incore inode table is basically a table which contains the inode which are loaded into the memory.

4. **User File Descriptor Table (UFDT)**:
   - Array of pointers which holds the specific entry from the file.
   - First 3 entries from the UFDT are reserved for the standard input,standard output,standard error.
5. **UAREA :**
   - It's a specific area allocated by the O.S to the process which contains the important information about that process for every process there is a specific UAREA.

Initialization

- **CreateDILB**: Dynamically allocates inodes and links them in a linked list.
- **InitialiseSuperBlock**: Initializes the superblock and UFDT array.

Error Handling

- The system provides comprehensive error messages for various failure cases, such as invalid parameters, file not found, insufficient permissions, and more.

Data Structures

1. Inode Structure

The inode (index node) structure is a fundamental data structure in the file system used to store information about a file. It contains metadata but not the actual data of the file. Each file has a unique inode.

- **FileName**: Stores the name of the file.
- **InodeNumber**: A unique identifier for each inode.
- **FileSize**: The size of the file.
- **FileActualSize**: The actual size of the content within the file.
- **FileType**: Specifies the type of file (e.g., regular file, directory).
- **Buffer**: A pointer to the actual data of the file.
- **LinkCount**: The number of hard links pointing to the inode.
- **ReferenceCount**: The number of references or file descriptors associated with the inode.
- **Permission**: The permissions assigned to the file (e.g., read, write).
- **Next**: A pointer to the next inode in a linked list structure.

2. Superblock Structure

The superblock is a critical data structure in the file system that holds information about the entire file system. It acts as a manager that keeps track of overall status and resources.

- **TotalInodes**: Total number of inodes in the file system.
- **FreeInode**: Number of available inodes that can be allocated to new files.

3. File Table Structure

The file table structure is used to maintain information about open files. It acts as an intermediary between file descriptors and inodes, allowing multiple file descriptors to point to the same file.

- **ReadOffset**: The current position in the file where the next read operation will start.
- **WriteOffset**: The current position in the file where the next write operation will start.
- **Count**: The reference count indicating how many file descriptors are using this file table entry.

- **Mode**: The mode in which the file is opened (e.g., read, write, read/write).
- **PtrInode**: A pointer to the inode associated with the file.

4. User File Descriptor Table (UFDT)

The User File Descriptor Table (UFDT) is an array of file descriptors that map to entries in the file table. Each entry in the UFDT points to a file table entry, enabling users to access files using file descriptors.

- **PtrFileTable**: A pointer to the file table entry for a particular file descriptor.

Initialization Functions

CreateDILB (Dynamic Inode List Block)

This function dynamically allocates inodes and links them into a linked list structure. It ensures that a pool of inodes is available for allocation when new files are created.

- **Inode Allocation**: Dynamically allocates memory for inodes and initializes their attributes.
- **Linked List**: Links the allocated inodes into a singly linked list for easy traversal and management.

InitialiseSuperBlock

This function initializes the superblock and the UFDT array. It sets up the initial state of the file system, making it ready for file operations.

- **Superblock Initialization**: Sets the total number of inodes and the number of free inodes.
- **UFDT Initialization**: Initializes the UFDT array to NULL, indicating that no files are currently open.

Additional Data Structures

SuperBlockObject

An instance of the superblock structure that holds the current state of the file system.

- **TotalInodes**: Represents the total number of inodes available.
- **FreeInode**: Indicates the number of free inodes that can be allocated.

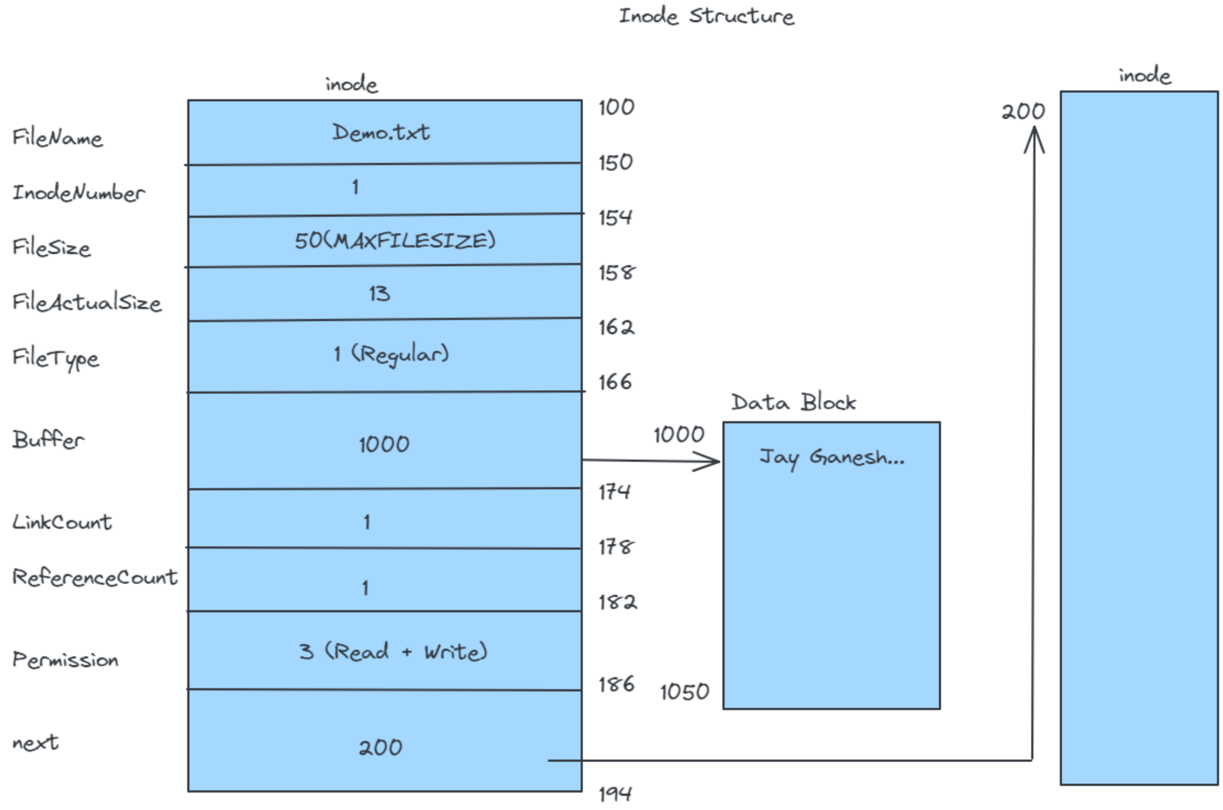
Head of the Inode List

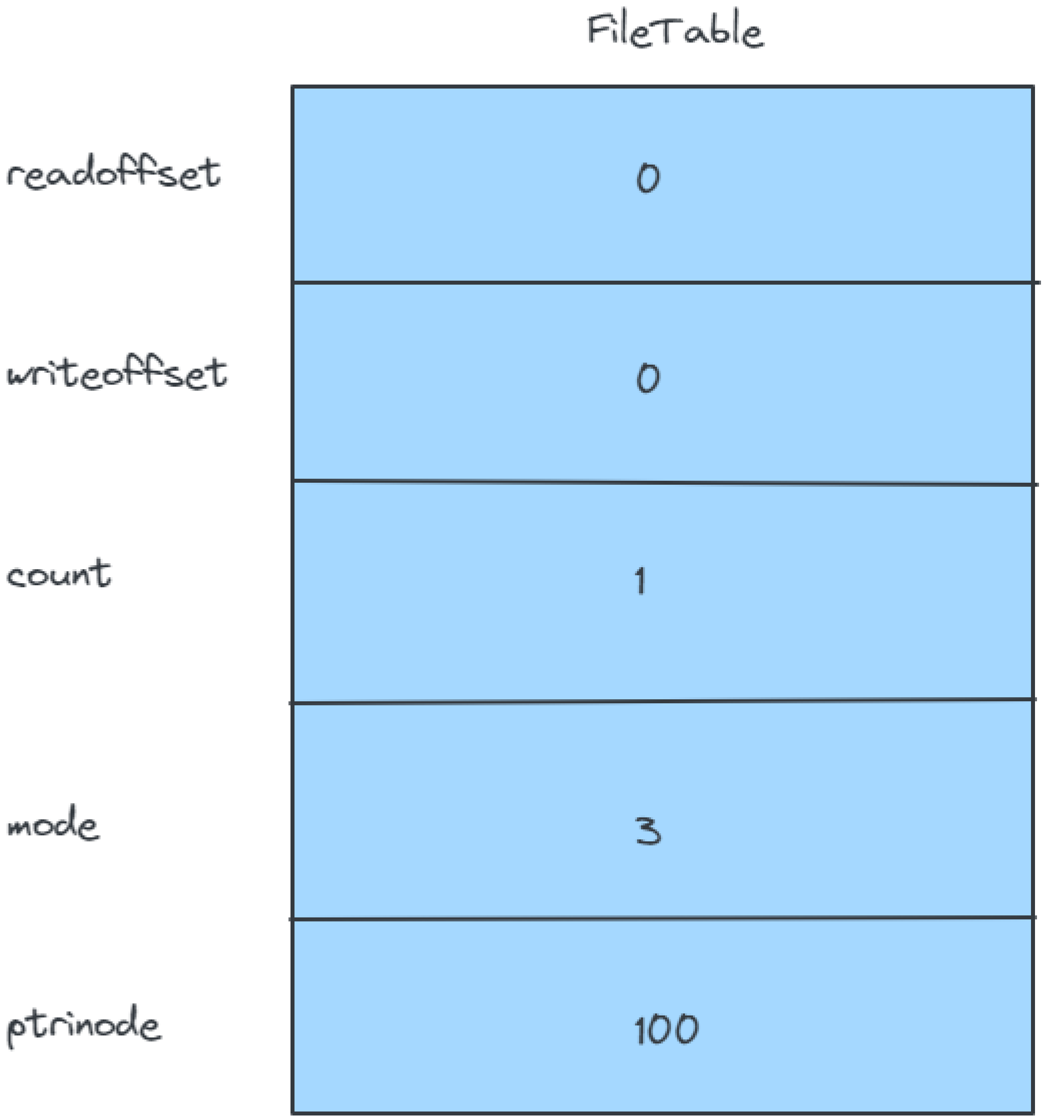
A pointer to the head of the linked list of inodes, allowing traversal and management of inodes

5. Example Usage of Data Structures

- **Creating a File**: Allocates an inode and initializes its attributes.
- **Opening a File**: Searches the inode linked list, updates the file table, and assigns a file descriptor.
- **Reading/Writing**: Uses the file descriptor to access the file table, retrieves the inode, and performs read/write operations on the buffer.
- **Deleting a File**: Decrements the link count, frees the buffer, and updates the inode and superblock.

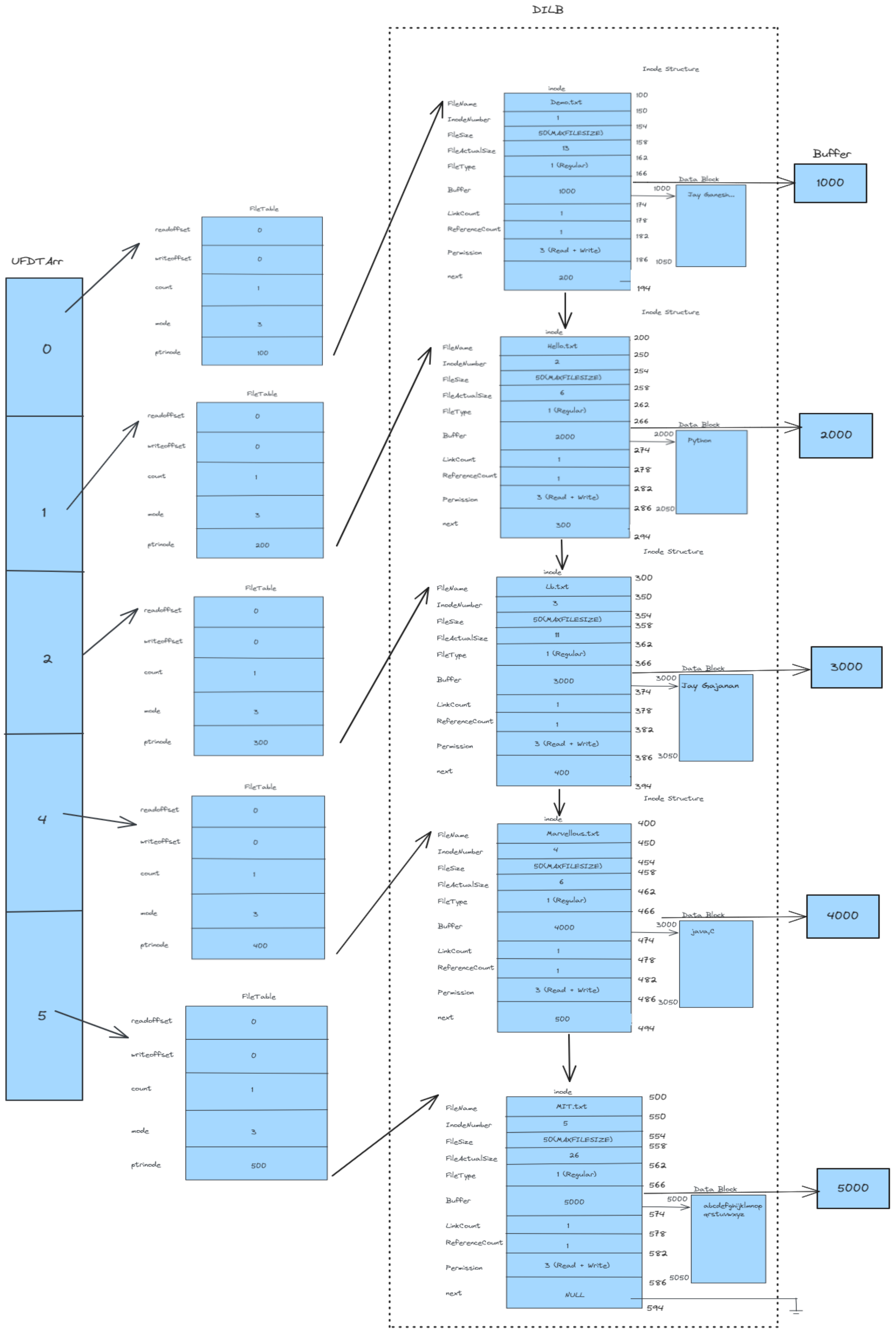
Diagram of data structures used in the project





FileTable

readoffset	0
writeoffset	0
count	1
mode	3
ptrinode	100



FileTable

readoffset	0	0	0	0	0
writeoffset	0	0	0	0	0
count	1	1	1	1	1
mode	3	3	3	3	3
ptrinode	100	200	300	400	500

Inode Structure

FileName	Demo.txt	100	Hello.txt	200	Lb.txt	300	Marvellous.txt	400	MIT.txt	500
InodeNumber	1	150	2	250	3	350	4	450	5	550
FileSize	50(MAXFILESIZE)	154	50(MAXFILESIZE)	254	50(MAXFILESIZE)	354	50(MAXFILESIZE)	454	50(MAXFILESIZE)	554
FileActualSize	13	158	6	258	11	358	6	458	26	558
FileType	1 (Regular)	162	1 (Regular)	262	1 (Regular)	362	1 (Regular)	462	1 (Regular)	562
Buffer	1000	166	2000	266	3000	366	4000	466	5000	566
LinkCount	1	174	1	274	1	374	1	474	1	574
ReferenceCount	1	178	1	278	1	378	1	478	1	578
Permission	3 (Read + Write)	182	3 (Read + Write)	282	3 (Read + Write)	382	3 (Read + Write)	482	3 (Read + Write)	582
next	200	186	300	286	400	386	500	486	NULL	586
		194		294		394		494		594

Flow of the Project

1. Initialization

The VFS is set up by initializing the superblock and creating the Disk Inode List Block (DILB).

a. Initialize Superblock

- **Function Call**: `InitialiseSuperBlock()`
- **Description**: Sets up the superblock with the total number of inodes and free inode count.

b. Create Disk Inode List Block (DILB)

- **Function Call**: `CreateDILB()`
- **Description**: Allocates memory for inodes and initializes them, linking each inode in a singly linked list to form a pool of available inodes.

2. File Operations

File operations include creating, opening, reading, writing, and closing files.

a. Creating a File

- **Function Call**: `CreateFile(char* name, int permission)`
- **Description**: Allocates a free inode, sets its attributes (name, permissions), and links it into the inode list. Updates the superblock's free inode count.

b. Opening a File

- **Function Call**: `OpenFile(char* name, int mode)`
- **Description**: Searches for the file by name in the inode list. If found, allocates a file table entry, initializes it, and returns a file descriptor.

c. Reading from a File

- **Function Call**: `ReadFile(int fd, char *arr, int size)`
- **Description**: Uses the file descriptor to locate the file table entry. Reads data from the file at the current read offset into the buffer and updates the read offset.

d. Writing to a File

- **Function Call**: `WriteFile(int fd, char *arr, int size)`
- **Description**: Uses the file descriptor to locate the file table entry. Writes data from the buffer to the file at the current write offset, updating the write offset and inode size.

e. Closing a File

- **Function Call**: `CloseFileByFD(int fd)`

- **Description**: Uses the file descriptor to locate the file table entry. Decrements the reference count and, if it reaches zero, deallocates the entry and updates the UFDT.

3. Error Handling and Validation

Checks are performed to ensure the integrity and correctness of file operations.

- **Inode Allocation Check**: Ensures a free inode is available before creating a file.
- **File Existence Check**: Validates that the file exists before opening, reading, or writing.
- **Permission Check**: Verifies the operation (read/write) is allowed based on file permissions.
- **File Descriptor Validation**: Ensures the provided file descriptor is valid and in use.

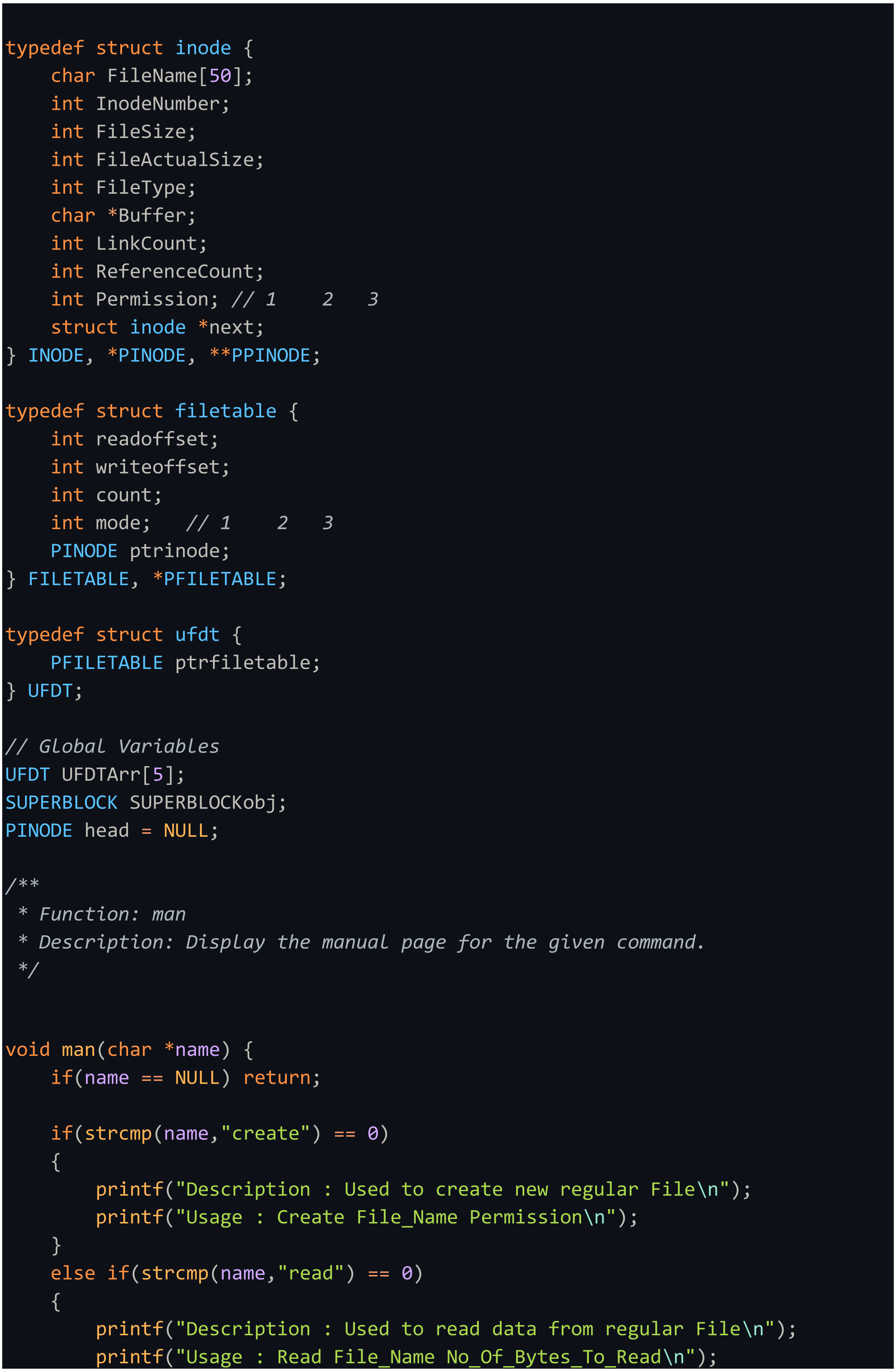
4. Updating Data Structures

File operations update various data structures to reflect the current state of the VFS.

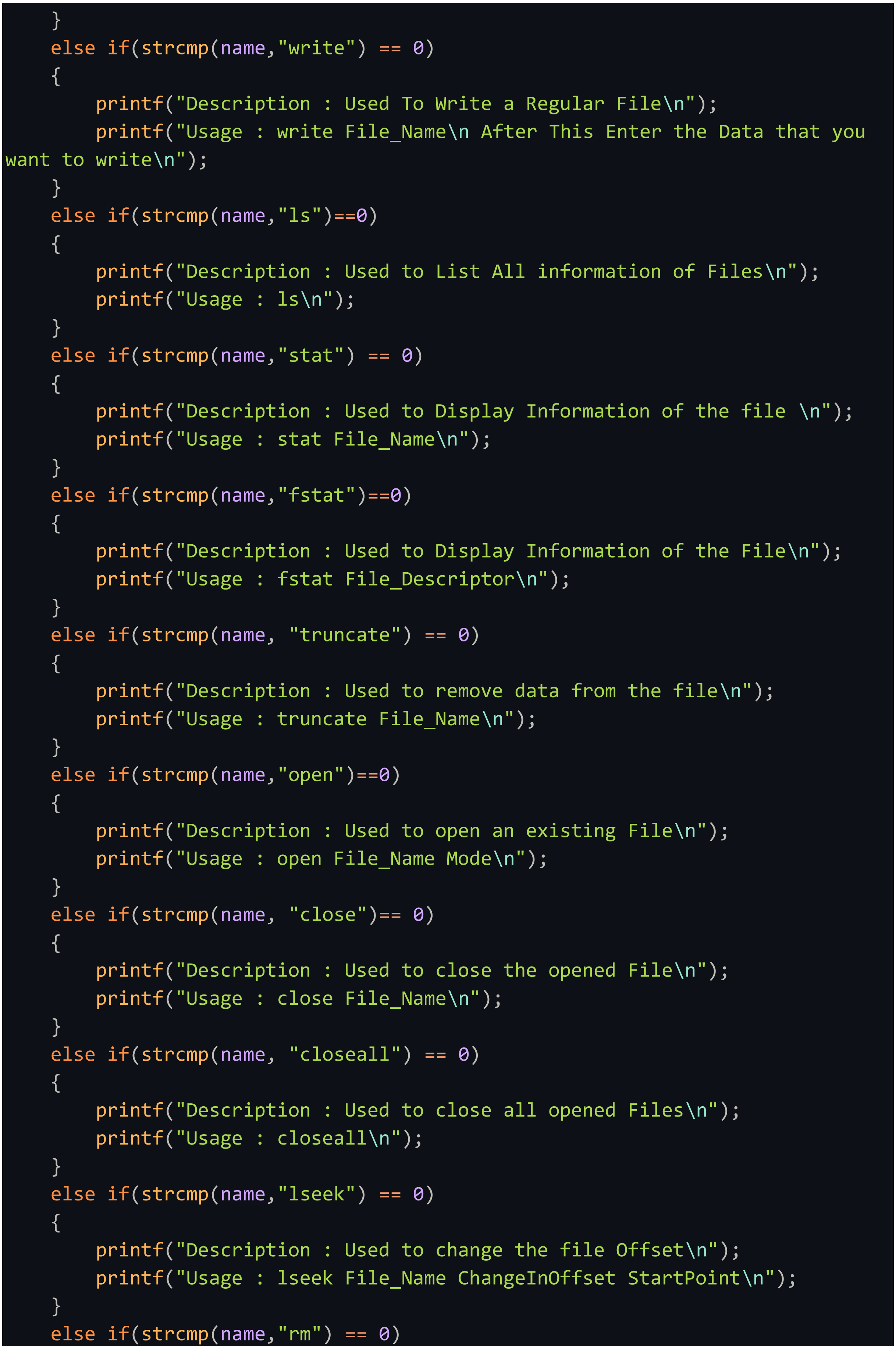
- **Superblock**: Updates the free inode count during file creation and deletion.
- **Inode List**: Updates inode attributes (size, permissions) as files are modified.
- **File Table**: Updates read/write offsets, reference counts, and modes during file operations.
- **UFDT**: Updates entries to reflect open and closed files.

Code of the Project

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>



// Constants and Macros
#define MAXINODE 50
#define READ 1
#define WRITE 2
#define MAXFILESIZE 2048
#define REGULAR 1
#define SPECIAL 2
#define START 0
#define CURRENT 1
#define END 2

// Structure Definitions
typedef struct superblock {
    int TotalInodes;
    int FreeInodes;
} SUPERBLOCK, *PSUPERBLOCK;
```

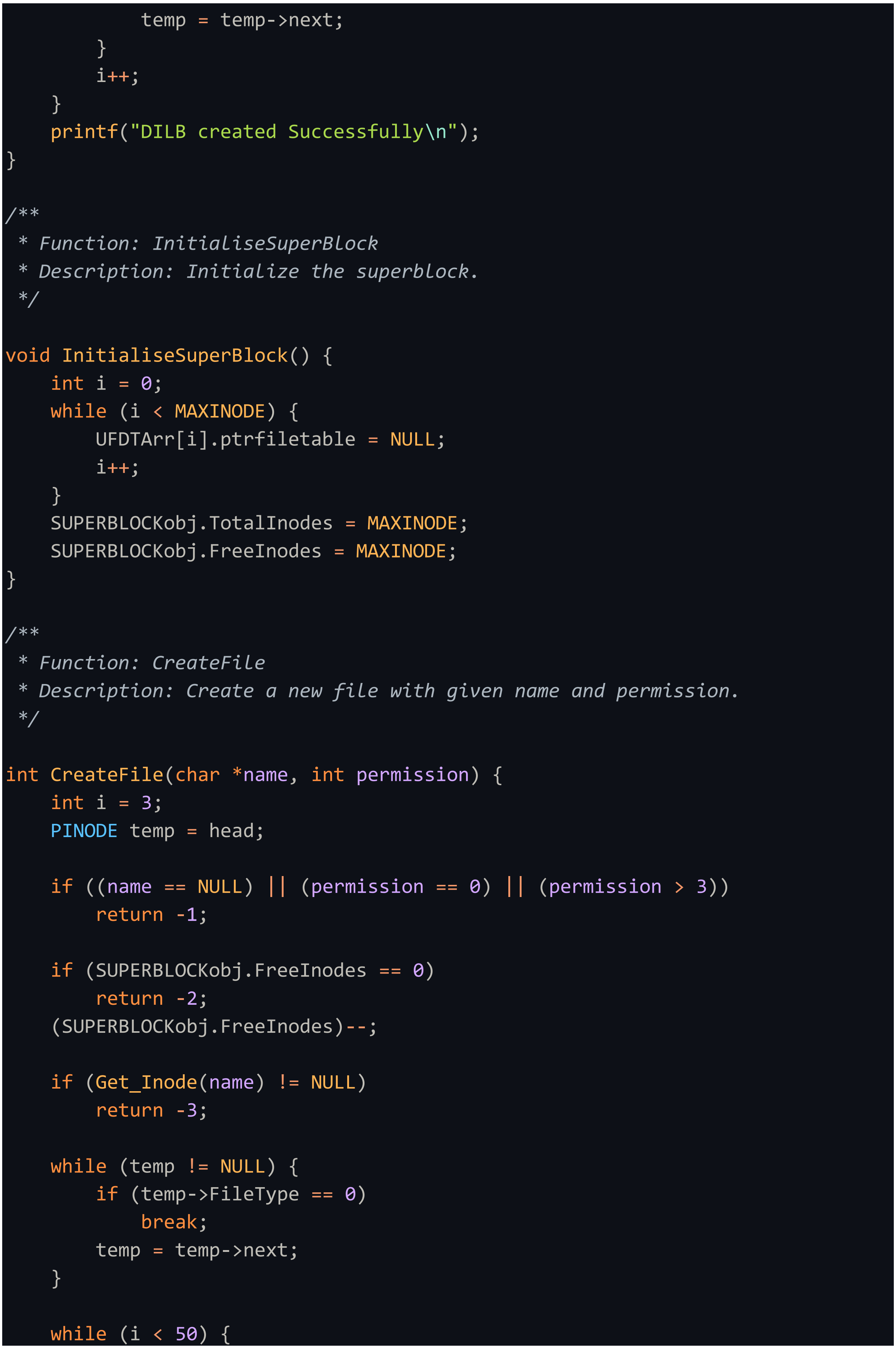


```c
typedef struct inode {
    char FileName[50];
    int InodeNumber;
    int FileSize;
    int FileActualSize;
    int FileType;
    char *Buffer;
    int LinkCount;
    int ReferenceCount;
    int Permission; // 1    2   3
    struct inode *next;
} INODE, *PINODE, **PPINODE;

typedef struct filetable {
    int readoffset;
    int writeoffset;
    int count;
    int mode;   // 1    2   3
    PINODE ptrinode;
} FILETABLE, *PFILETABLE;

typedef struct ufdt {
    PFILETABLE ptrfiletable;
} UFDT;

// Global Variables
UFDT UFDTArr[5];
SUPERBLOCK SUPERBLOCKobj;
PINODE head = NULL;

/**
 * Function: man
 * Description: Display the manual page for the given command.
 */


void man(char *name) {
    if(name == NULL) return;

    if(strcmp(name,"create") == 0)
    {
        printf("Description : Used to create new regular File\n");
        printf("Usage : Create File_Name Permission\n");
    }
    else if(strcmp(name,"read") == 0)
    {
        printf("Description : Used to read data from regular File\n");
        printf("Usage : Read File_Name No_Of_Bytes_To_Read\n");
```



```
    }
    else if(strcmp(name,"write") == 0)
    {
        printf("Description : Used To Write a Regular File\n");
        printf("Usage : write File_Name\n After This Enter the Data that you
want to write\n");
    }
    else if(strcmp(name,"ls")==0)
    {
        printf("Description : Used to List All information of Files\n");
        printf("Usage : ls\n");
    }
    else if(strcmp(name,"stat") == 0)
    {
        printf("Description : Used to Display Information of the file \n");
        printf("Usage : stat File_Name\n");
    }
    else if(strcmp(name,"fstat")==0)
    {
        printf("Description : Used to Display Information of the File\n");
        printf("Usage : fstat File_Descriptor\n");
    }
    else if(strcmp(name, "truncate") == 0)
    {
        printf("Description : Used to remove data from the file\n");
        printf("Usage : truncate File_Name\n");
    }
    else if(strcmp(name,"open")==0)
    {
        printf("Description : Used to open an existing File\n");
        printf("Usage : open File_Name Mode\n");
    }
    else if(strcmp(name, "close")== 0)
    {
        printf("Description : Used to close the opened File\n");
        printf("Usage : close File_Name\n");
    }
    else if(strcmp(name, "closeall") == 0)
    {
        printf("Description : Used to close all opened Files\n");
        printf("Usage : closeall\n");
    }
    else if(strcmp(name,"lseek") == 0)
    {
        printf("Description : Used to change the file Offset\n");
        printf("Usage : lseek File_Name ChangeInOffset StartPoint\n");
    }
    else if(strcmp(name,"rm") == 0)
    {
        printf("Description : Used to remove any FIle\n");
        printf("Usage : rm File_Name\n");
    }
    else
    {
        printf("ERROR : No Manual Entry Available\n");
    }
}

/**
 * Function: DisplayHelp
 * Description: Display help for all commands.
 */


void DisplayHelp() {
    printf("ls : To List out the Files\n");
    printf("clear : To Clear the console\n");
    printf("open : To Open a File\n");
    printf("close : To Close a File\n");
    printf("closeall : To Close all opened Files\n");
    printf("read : To read the contents from file\n");
    printf("write : To Write contents in a file\n");
    printf("exit : To Terminate the File System\n");
    printf("stat : To Display information of the file using name\n");
    printf("fstat : To Display information ofthe file using Descriptor\n");
    printf("truncate : To Remove all data from file\n");
    printf("rm : To delete a file \n");
}

/**
 * Function: GetFDFromName
 * Description: Get the file descriptor from the file name.
 */


int GetFDFromName(char *name) {
    int i = 0;
    while (i < 50) {
        if (UFDTArr[i].ptrfiletable != NULL) {
            if (strcmp((UFDTArr[i].ptrfiletable->ptrinode->FileName), name) ==
0) {
                break;
            }
        }
        i++;
    }
    if (i == 50)
```

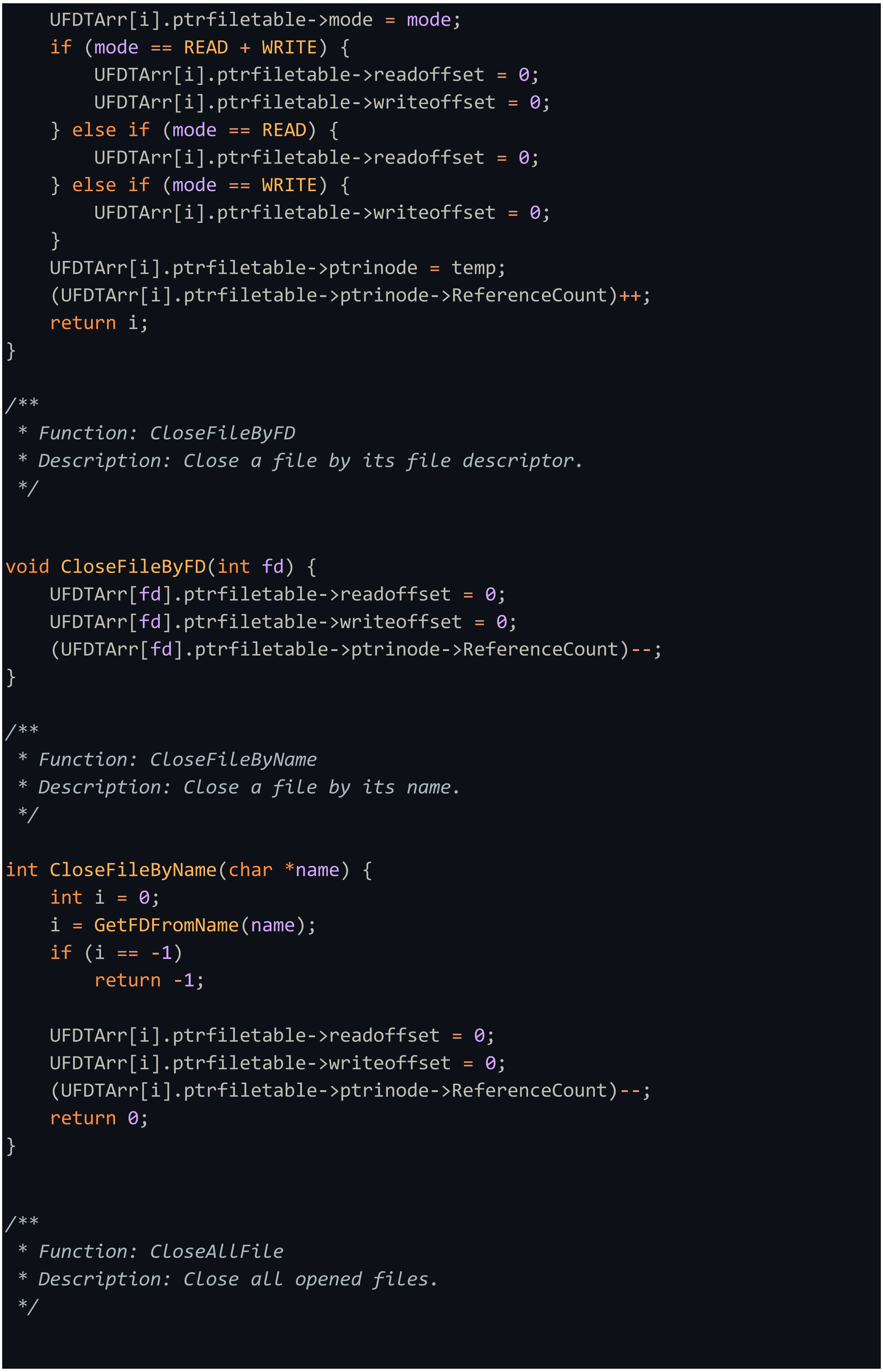
```
        return -1;
    else
        return i;
}

/**
 * Function: Get_Inode
 * Description: Get the inode from the file name.
 */

PINODE Get_Inode(char *name) {
    PINODE temp = head;
    if (name == NULL)
        return NULL;

    while (temp != NULL) {
        if (strcmp(name, temp->FileName) == 0)
            break;
        temp = temp->next;
    }
    return temp;
}

/**
 * Function: CreateDILB
 * Description: Create Disk Inode List Block.
 */

void CreateDILB() {
    int i = 3;
    PINODE newn = NULL;
    PINODE temp = head;

    while (i <= MAXINODE) {
        newn = (PINODE)malloc(sizeof(INODE));
        newn->LinkCount = 0;
        newn->ReferenceCount = 0;
        newn->FileType = 0;
        newn->FileSize = 0;
        newn->Buffer = NULL;
        newn->next = NULL;
        newn->InodeNumber = i;

        if (temp == NULL) {
            head = newn;
            temp = head;
        } else {
            temp->next = newn;
```



```c
            temp = temp->next;
        }
        i++;
    }
    printf("DILB created Successfully\n");
}

/**
 * Function: InitialiseSuperBlock
 * Description: Initialize the superblock.
 */

void InitialiseSuperBlock() {
    int i = 0;
    while (i < MAXINODE) {
        UFDTArr[i].ptrfiletable = NULL;
        i++;
    }
    SUPERBLOCKobj.TotalInodes = MAXINODE;
    SUPERBLOCKobj.FreeInodes = MAXINODE;
}

/**
 * Function: CreateFile
 * Description: Create a new file with given name and permission.
 */

int CreateFile(char *name, int permission) {
    int i = 3;
    PINODE temp = head;

    if ((name == NULL) || (permission == 0) || (permission > 3))
        return -1;

    if (SUPERBLOCKobj.FreeInodes == 0)
        return -2;
    (SUPERBLOCKobj.FreeInodes)--;

    if (Get_Inode(name) != NULL)
        return -3;

    while (temp != NULL) {
        if (temp->FileType == 0)
            break;
        temp = temp->next;
    }

    while (i < 50) {
        if (UFDTArr[i].ptrfiletable == NULL)
            break;
        i++;
    }

    UFDTArr[i].ptrfiletable = (PFILETABLE)malloc(sizeof(FILETABLE));
    UFDTArr[i].ptrfiletable->count = 1;
    UFDTArr[i].ptrfiletable->mode = permission;
    UFDTArr[i].ptrfiletable->readoffset = 0;
    UFDTArr[i].ptrfiletable->writeoffset = 0;
    UFDTArr[i].ptrfiletable->ptrinode = temp;

    strcpy(UFDTArr[i].ptrfiletable->ptrinode->FileName, name);
    UFDTArr[i].ptrfiletable->ptrinode->FileType = REGULAR;
    UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount = 1;
    UFDTArr[i].ptrfiletable->ptrinode->LinkCount = 1;
    UFDTArr[i].ptrfiletable->ptrinode->FileSize = MAXFILESIZE;
    UFDTArr[i].ptrfiletable->ptrinode->FileActualSize = 0;
    UFDTArr[i].ptrfiletable->ptrinode->Permission = permission;
    UFDTArr[i].ptrfiletable->ptrinode->Buffer = (char *)malloc(MAXFILESIZE);

    return i;
}

/**
 * Function: rm_File
 * Description: Remove the file with given name.
 */


int rm_File(char *name) {
    int fd = 0;

    fd = GetFDFromName(name);
    if (fd == -1)
        return -1;

    (UFDTArr[fd].ptrfiletable->ptrinode->LinkCount)--;
    if (UFDTArr[fd].ptrfiletable->ptrinode->LinkCount == 0) {
        UFDTArr[fd].ptrfiletable->ptrinode->FileType = 0;
        free(UFDTArr[fd].ptrfiletable->ptrinode->Buffer); // Free buffer
memory
        free(UFDTArr[fd].ptrfiletable);
    }
    UFDTArr[fd].ptrfiletable = NULL;
    (SUPERBLOCKobj.FreeInodes)++;
    return 0;
}
```

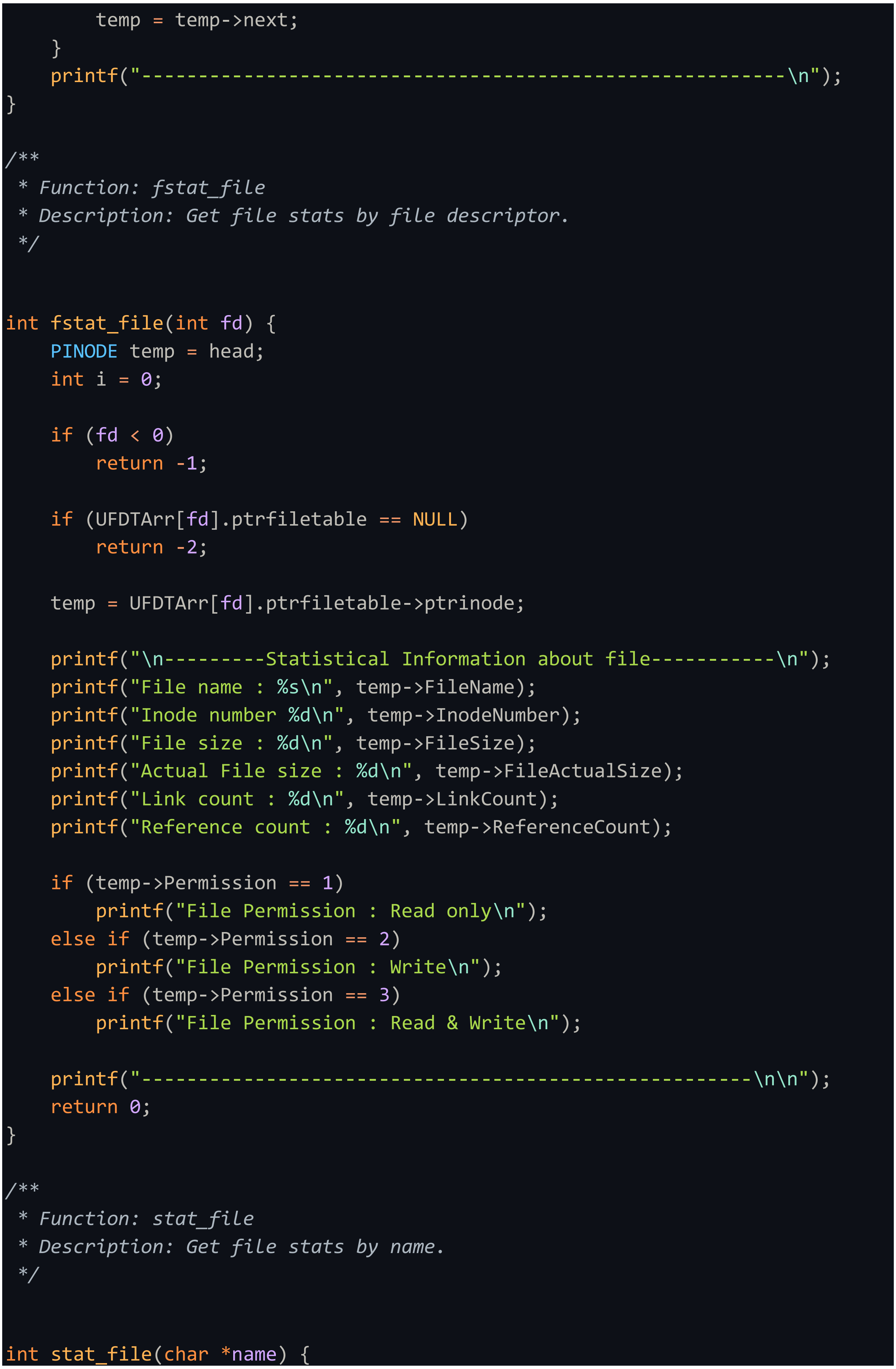
```
/**
 * Function: ReadFile
 * Description: Read data from the file.
 */


int ReadFile(int fd, char *arr, int isize) {
    int read_size = 0;

    if (UFDTArr[fd].ptrfiletable == NULL)
        return -1;

    if (UFDTArr[fd].ptrfiletable->mode != READ && UFDTArr[fd].ptrfiletable-
>mode != READ + WRITE)
        return -2;
    if (UFDTArr[fd].ptrfiletable->ptrinode->Permission != READ &&
UFDTArr[fd].ptrfiletable->ptrinode->Permission != READ + WRITE)
        return -2;
    if (UFDTArr[fd].ptrfiletable->readoffset == UFDTArr[fd].ptrfiletable-
>ptrinode->FileActualSize)
        return -3;
    if (UFDTArr[fd].ptrfiletable->ptrinode->FileType != REGULAR)
        return -4;

    read_size = (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) -
(UFDTArr[fd].ptrfiletable->readoffset);
    if (read_size < isize) {
        strncpy(arr, (UFDTArr[fd].ptrfiletable->ptrinode->Buffer) +
(UFDTArr[fd].ptrfiletable->readoffset), read_size);
        UFDTArr[fd].ptrfiletable->readoffset = UFDTArr[fd].ptrfiletable-
>readoffset + read_size;
    } else {
        strncpy(arr, (UFDTArr[fd].ptrfiletable->ptrinode->Buffer) +
(UFDTArr[fd].ptrfiletable->readoffset), isize);
        (UFDTArr[fd].ptrfiletable->readoffset) = (UFDTArr[fd].ptrfiletable-
>readoffset) + isize;
    }
    return isize;
}


/**
 * Function: WriteFile
 * Description: Write data to the file.
 */

int WriteFile(int fd, char *arr, int isize) {
    if (((UFDTArr[fd].ptrfiletable->mode) != WRITE) &&
((UFDTArr[fd].ptrfiletable->mode) != READ + WRITE))
```

```
        return -1;
    if (((UFDTArr[fd].ptrfiletable->ptrinode->Permission) != WRITE) && ((UFDTArr[fd].ptrfiletable->ptrinode->Permission) != READ + WRITE))
        return -1;
    if ((UFDTArr[fd].ptrfiletable->writeoffset) == MAXFILESIZE)
        return -2;
    if ((UFDTArr[fd].ptrfiletable->ptrinode->FileType) != REGULAR)
        return -3;

    strncpy((UFDTArr[fd].ptrfiletable->ptrinode->Buffer) + (UFDTArr[fd].ptrfiletable->writeoffset), arr, isize);

    (UFDTArr[fd].ptrfiletable->writeoffset) = (UFDTArr[fd].ptrfiletable->writeoffset) + isize;
    (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) = (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + isize;
    return isize;
}


/**
 * Function: OpenFile
 * Description: Open a file with the given name and mode.
 */

int OpenFile(char *name, int mode) {
    int i = 0;
    PINODE temp = NULL;

    if (name == NULL || mode <= 0)
        return -1;

    temp = Get_Inode(name);
    if (temp == NULL)
        return -2;

    if (temp->Permission < mode)
        return -3;

    while (i < 50) {
        if (UFDTArr[i].ptrfiletable == NULL)
            break;
        i++;
    }

    UFDTArr[i].ptrfiletable = (PFILETABLE)malloc(sizeof(FILETABLE));
    if (UFDTArr[i].ptrfiletable == NULL) return -1;

    UFDTArr[i].ptrfiletable->count = 1;
```

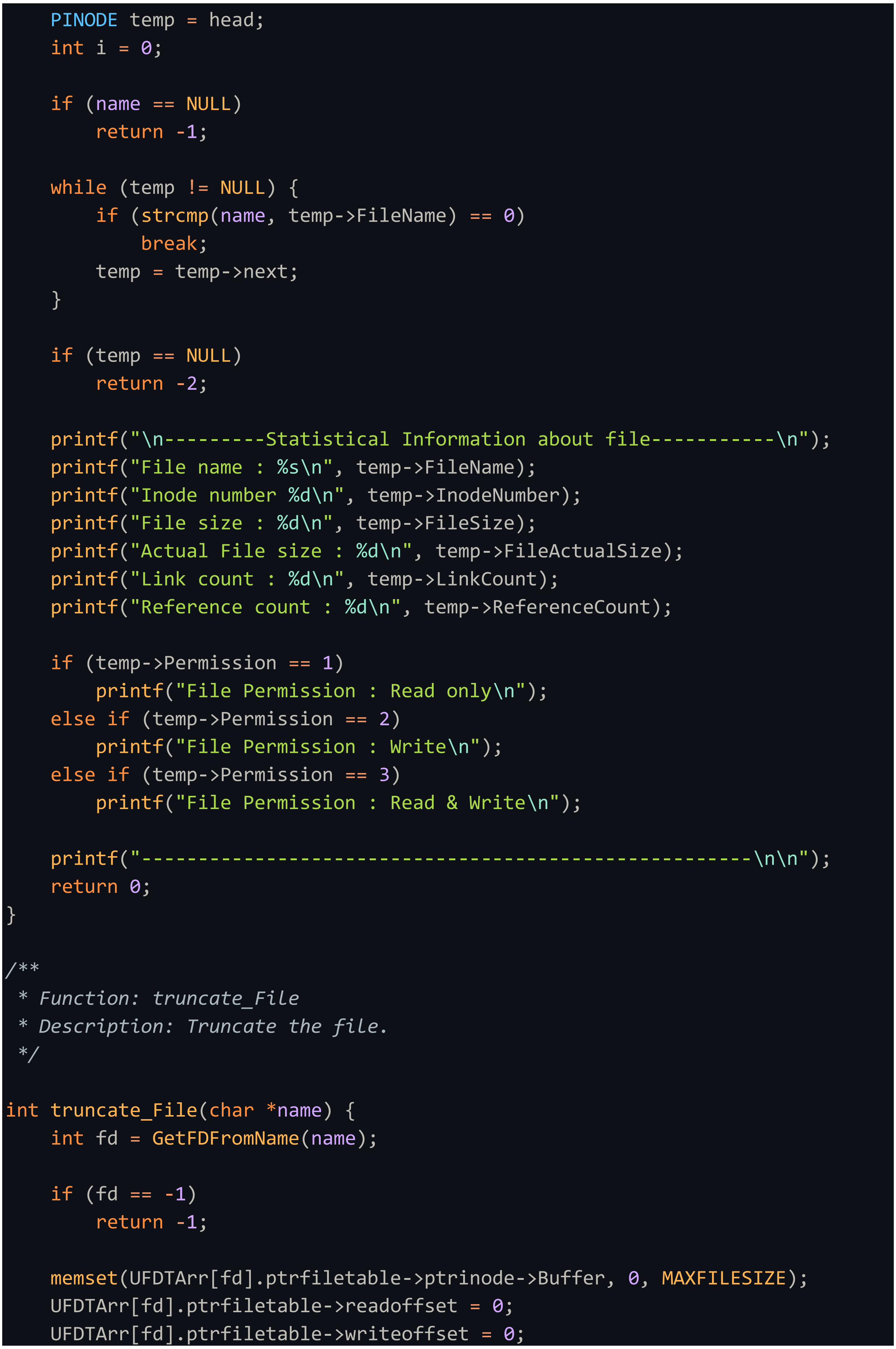


```
    UFDTArr[i].ptrfiletable->mode = mode;
    if (mode == READ + WRITE) {
        UFDTArr[i].ptrfiletable->readoffset = 0;
        UFDTArr[i].ptrfiletable->writeoffset = 0;
    } else if (mode == READ) {
        UFDTArr[i].ptrfiletable->readoffset = 0;
    } else if (mode == WRITE) {
        UFDTArr[i].ptrfiletable->writeoffset = 0;
    }
    UFDTArr[i].ptrfiletable->ptrinode = temp;
    (UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)++;
    return i;
}

/**
 * Function: CloseFileByFD
 * Description: Close a file by its file descriptor.
 */


void CloseFileByFD(int fd) {
    UFDTArr[fd].ptrfiletable->readoffset = 0;
    UFDTArr[fd].ptrfiletable->writeoffset = 0;
    (UFDTArr[fd].ptrfiletable->ptrinode->ReferenceCount)--;
}

/**
 * Function: CloseFileByName
 * Description: Close a file by its name.
 */

int CloseFileByName(char *name) {
    int i = 0;
    i = GetFDFromName(name);
    if (i == -1)
        return -1;

    UFDTArr[i].ptrfiletable->readoffset = 0;
    UFDTArr[i].ptrfiletable->writeoffset = 0;
    (UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)--;
    return 0;
}


/**
 * Function: CloseAllFile
 * Description: Close all opened files.
 */
```

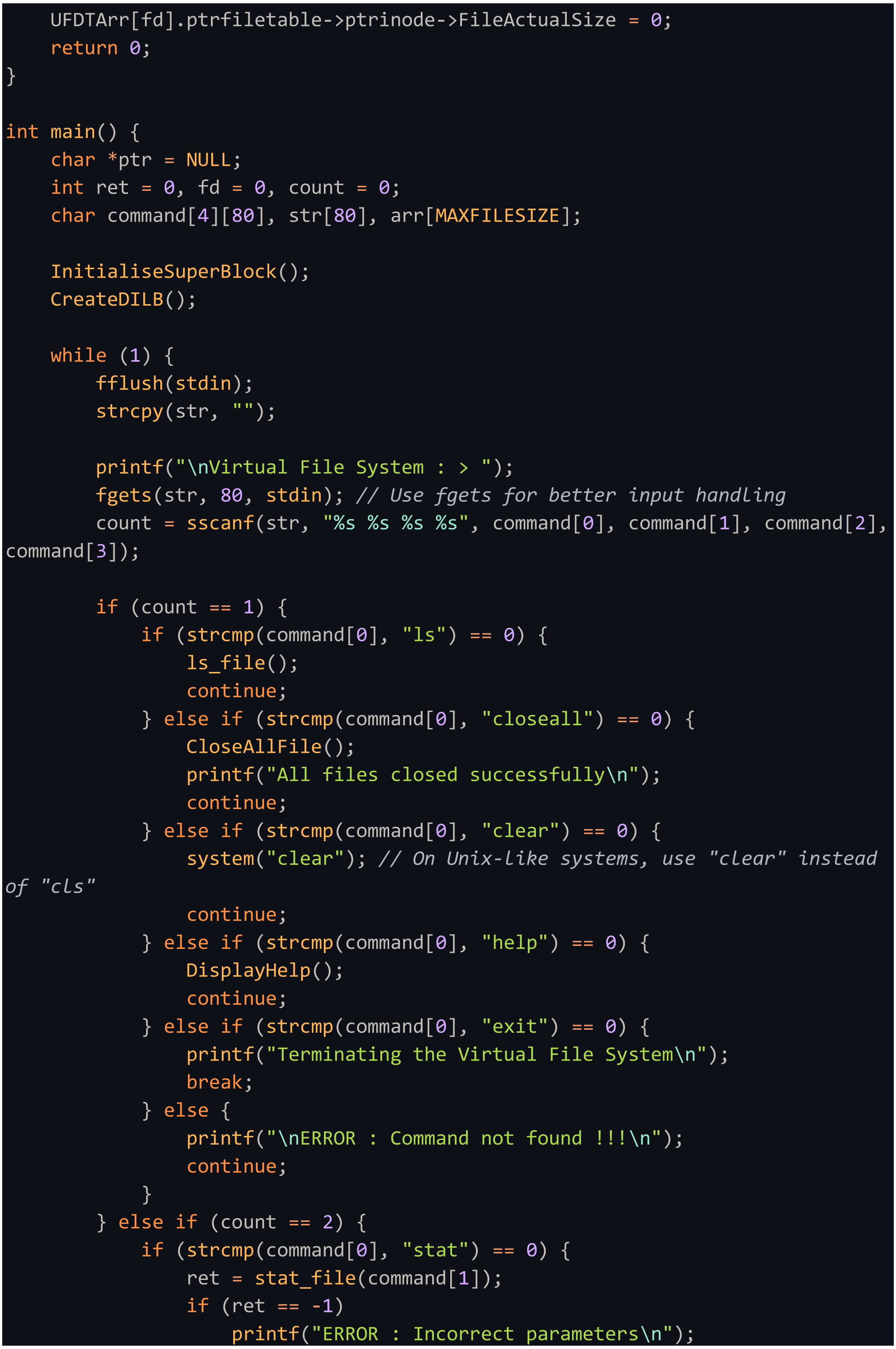
```
void CloseAllFile() {
    int i = 0;
    while (i < 50) {
        if (UFDTArr[i].ptrfiletable != NULL) {
            UFDTArr[i].ptrfiletable->readoffset = 0;
            UFDTArr[i].ptrfiletable->writeoffset = 0;
            (UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)--;
            break; // added break to exit loop after closing
        }
        i++;
    }
}

/**
 * Function: LseekFile
 * Description: Change the file offset.
 */


int LseekFile(int fd, int size, int from) {
    if ((fd < 0) || (from < 0) || (from > 2)) // Adjusted condition
        return -1;
    if (UFDTArr[fd].ptrfiletable == NULL)
        return -1;

    if ((UFDTArr[fd].ptrfiletable->mode == READ) || (UFDTArr[fd].ptrfiletable-
>mode == READ + WRITE)) {
        if (from == CURRENT) {
            if (((UFDTArr[fd].ptrfiletable->readoffset) + size) >
UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)
                return -1;
            if (((UFDTArr[fd].ptrfiletable->readoffset) + size) < 0)
                return -1;
            (UFDTArr[fd].ptrfiletable->readoffset) =
(UFDTArr[fd].ptrfiletable->readoffset) + size;
        } else if (from == START) {
            if (size > (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) ||
size < 0)
                return -1;
            (UFDTArr[fd].ptrfiletable->readoffset) = size;
        } else if (from == END) {
            if ((UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size >
MAXFILESIZE || ((UFDTArr[fd].ptrfiletable->readoffset) + size) < 0)
                return -1;
            (UFDTArr[fd].ptrfiletable->readoffset) =
(UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size;
        }
    } else if (UFDTArr[fd].ptrfiletable->mode == WRITE) {
        if (from == CURRENT) {
            if (((UFDTArr[fd].ptrfiletable->writeoffset) + size) >
MAXFILESIZE)
                return -1;
            if (((UFDTArr[fd].ptrfiletable->writeoffset) + size) < 0)
                return -1;
            if (((UFDTArr[fd].ptrfiletable->writeoffset) + size) >
(UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize))
                (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) =
(UFDTArr[fd].ptrfiletable->writeoffset) + size;
            (UFDTArr[fd].ptrfiletable->writeoffset) =
(UFDTArr[fd].ptrfiletable->writeoffset) + size;
        } else if (from == START) {
            if (size > MAXFILESIZE || size < 0)
                return -1;
            if (size > (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize))
                (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) = size;
            (UFDTArr[fd].ptrfiletable->writeoffset) = size;
        } else if (from == END) {
            if ((UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size >
MAXFILESIZE || ((UFDTArr[fd].ptrfiletable->writeoffset) + size) < 0)
                return -1;
            (UFDTArr[fd].ptrfiletable->writeoffset) =
(UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size;
        }
    }
    return 0;
}

/**
 * Function: ls_file
 * Description: List all the files.
 */

void ls_file() {
    PINODE temp = head;

    if (SUPERBLOCKobj.FreeInodes == MAXINODE) {
        printf("Error : There are no files\n");
        return;
    }

    printf("\nFile Name\tInode number\tFile size\tLink count\n");
    printf("---------------------------------------------------------\n");
    while (temp != NULL) {
        if (temp->FileType != 0) {
            printf("%s\t\t%d\t\t%d\t\t%d\n", temp->FileName, temp-
>InodeNumber, temp->FileSize, temp->LinkCount);
        }
```



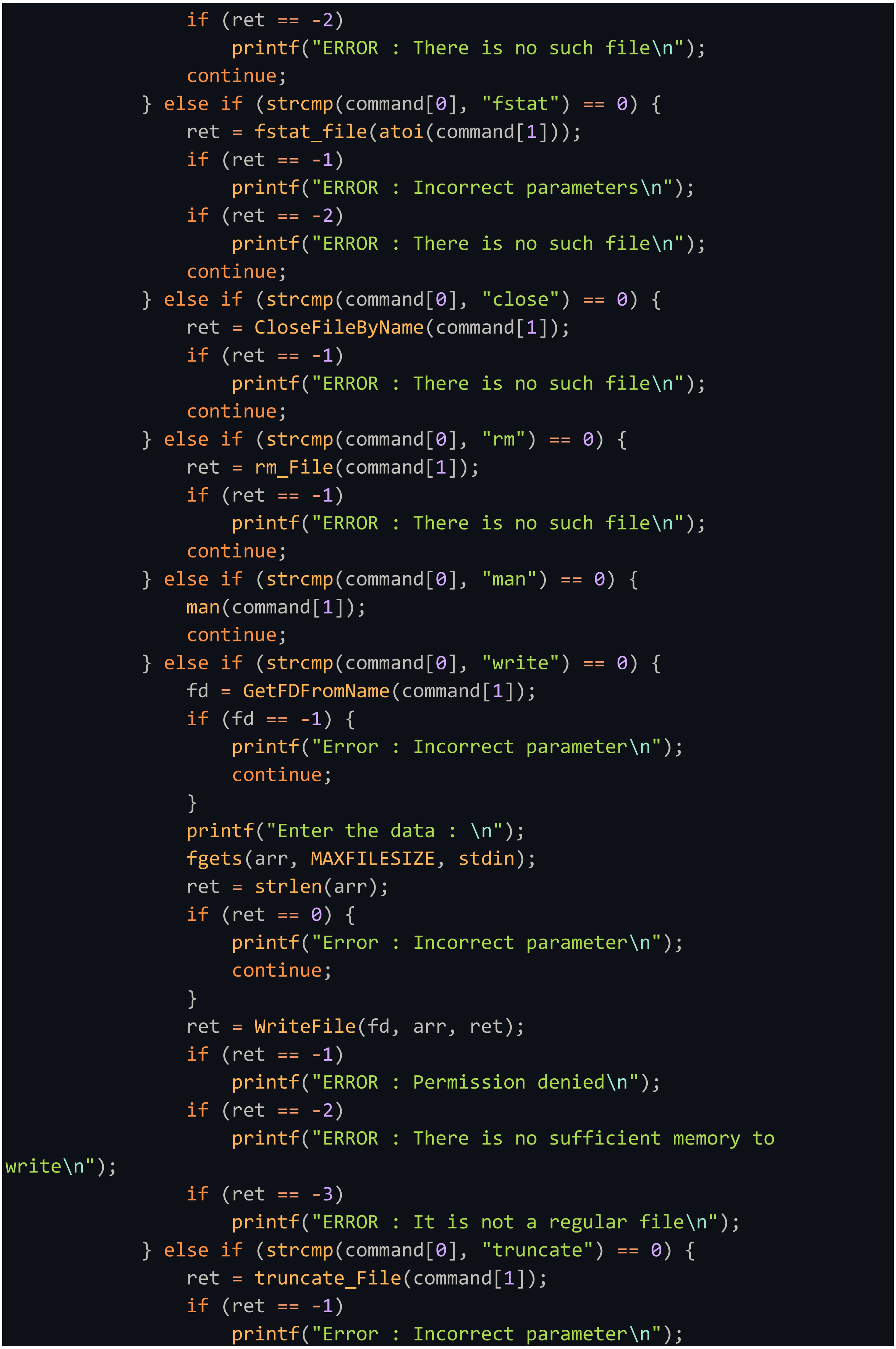
```c
        temp = temp->next;
    }
    printf("---------------------------------------------------------\n");
}

/**
 * Function: fstat_file
 * Description: Get file stats by file descriptor.
 */


int fstat_file(int fd) {
    PINODE temp = head;
    int i = 0;

    if (fd < 0)
        return -1;

    if (UFDTArr[fd].ptrfiletable == NULL)
        return -2;

    temp = UFDTArr[fd].ptrfiletable->ptrinode;

    printf("\n---------Statistical Information about file-----------\n");
    printf("File name : %s\n", temp->FileName);
    printf("Inode number %d\n", temp->InodeNumber);
    printf("File size : %d\n", temp->FileSize);
    printf("Actual File size : %d\n", temp->FileActualSize);
    printf("Link count : %d\n", temp->LinkCount);
    printf("Reference count : %d\n", temp->ReferenceCount);

    if (temp->Permission == 1)
        printf("File Permission : Read only\n");
    else if (temp->Permission == 2)
        printf("File Permission : Write\n");
    else if (temp->Permission == 3)
        printf("File Permission : Read & Write\n");

    printf("------------------------------------------------------\n\n");
    return 0;
}

/**
 * Function: stat_file
 * Description: Get file stats by name.
 */


int stat_file(char *name) {
```



```
    PINODE temp = head;
    int i = 0;

    if (name == NULL)
        return -1;

    while (temp != NULL) {
        if (strcmp(name, temp->FileName) == 0)
            break;
        temp = temp->next;
    }

    if (temp == NULL)
        return -2;

    printf("\n---------Statistical Information about file-----------\n");
    printf("File name : %s\n", temp->FileName);
    printf("Inode number %d\n", temp->InodeNumber);
    printf("File size : %d\n", temp->FileSize);
    printf("Actual File size : %d\n", temp->FileActualSize);
    printf("Link count : %d\n", temp->LinkCount);
    printf("Reference count : %d\n", temp->ReferenceCount);

    if (temp->Permission == 1)
        printf("File Permission : Read only\n");
    else if (temp->Permission == 2)
        printf("File Permission : Write\n");
    else if (temp->Permission == 3)
        printf("File Permission : Read & Write\n");

    printf("------------------------------------------------------\n\n");
    return 0;
}

/**
 * Function: truncate_File
 * Description: Truncate the file.
 */

int truncate_File(char *name) {
    int fd = GetFDFromName(name);

    if (fd == -1)
        return -1;

    memset(UFDTArr[fd].ptrfiletable->ptrinode->Buffer, 0, MAXFILESIZE);
    UFDTArr[fd].ptrfiletable->readoffset = 0;
    UFDTArr[fd].ptrfiletable->writeoffset = 0;
```



```
    UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize = 0;
    return 0;
}

int main() {
    char *ptr = NULL;
    int ret = 0, fd = 0, count = 0;
    char command[4][80], str[80], arr[MAXFILESIZE];

    InitialiseSuperBlock();
    CreateDILB();

    while (1) {
        fflush(stdin);
        strcpy(str, "");

        printf("\nVirtual File System : > ");
        fgets(str, 80, stdin); // Use fgets for better input handling
        count = sscanf(str, "%s %s %s %s", command[0], command[1], command[2],
command[3]);

        if (count == 1) {
            if (strcmp(command[0], "ls") == 0) {
                ls_file();
                continue;
            } else if (strcmp(command[0], "closeall") == 0) {
                CloseAllFile();
                printf("All files closed successfully\n");
                continue;
            } else if (strcmp(command[0], "clear") == 0) {
                system("clear"); // On Unix-like systems, use "clear" instead
of "cls"
                continue;
            } else if (strcmp(command[0], "help") == 0) {
                DisplayHelp();
                continue;
            } else if (strcmp(command[0], "exit") == 0) {
                printf("Terminating the Virtual File System\n");
                break;
            } else {
                printf("\nERROR : Command not found !!!\n");
                continue;
            }
        } else if (count == 2) {
            if (strcmp(command[0], "stat") == 0) {
                ret = stat_file(command[1]);
                if (ret == -1)
                    printf("ERROR : Incorrect parameters\n");
```



```
                if (ret == -2)
                    printf("ERROR : There is no such file\n");
                continue;
            } else if (strcmp(command[0], "fstat") == 0) {
                ret = fstat_file(atoi(command[1]));
                if (ret == -1)
                    printf("ERROR : Incorrect parameters\n");
                if (ret == -2)
                    printf("ERROR : There is no such file\n");
                continue;
            } else if (strcmp(command[0], "close") == 0) {
                ret = CloseFileByName(command[1]);
                if (ret == -1)
                    printf("ERROR : There is no such file\n");
                continue;
            } else if (strcmp(command[0], "rm") == 0) {
                ret = rm_File(command[1]);
                if (ret == -1)
                    printf("ERROR : There is no such file\n");
                continue;
            } else if (strcmp(command[0], "man") == 0) {
                man(command[1]);
                continue;
            } else if (strcmp(command[0], "write") == 0) {
                fd = GetFDFromName(command[1]);
                if (fd == -1) {
                    printf("Error : Incorrect parameter\n");
                    continue;
                }
                printf("Enter the data : \n");
                fgets(arr, MAXFILESIZE, stdin);
                ret = strlen(arr);
                if (ret == 0) {
                    printf("Error : Incorrect parameter\n");
                    continue;
                }
                ret = WriteFile(fd, arr, ret);
                if (ret == -1)
                    printf("ERROR : Permission denied\n");
                if (ret == -2)
                    printf("ERROR : There is no sufficient memory to
write\n");
                if (ret == -3)
                    printf("ERROR : It is not a regular file\n");
            } else if (strcmp(command[0], "truncate") == 0) {
                ret = truncate_File(command[1]);
                if (ret == -1)
                    printf("Error : Incorrect parameter\n");
            } else {
                printf("\nERROR : Command not found !!!\n");
                continue;
            }
        } else if (count == 3) {
            if (strcmp(command[0], "create") == 0) {
                ret = CreateFile(command[1], atoi(command[2]));
                if (ret >= 0)
                    printf("File is successfully created with file descriptor
: %d\n", ret);
                if (ret == -1)
                    printf("ERROR : Incorrect parameters\n");
                if (ret == -2)
                    printf("ERROR : There is no inodes\n");
                if (ret == -3)
                    printf("ERROR : File already exists\n");
                if (ret == -4)
                    printf("ERROR : Memory allocation failure\n");
                continue;
            } else if (strcmp(command[0], "open") == 0) {
                ret = OpenFile(command[1], atoi(command[2]));
                if (ret >= 0)
                    printf("File is successfully opened with file descriptor :
%d\n", ret);
                if (ret == -1)
                    printf("ERROR : Incorrect parameters\n");
                if (ret == -2)
                    printf("ERROR : File not present\n");
                if (ret == -3)
                    printf("ERROR : Permission denied\n");
                continue;
            } else if (strcmp(command[0], "read") == 0) {
                fd = GetFDFromName(command[1]);
                if (fd == -1) {
                    printf("Error : Incorrect parameter\n");
                    continue;
                }
                ptr = (char *)malloc(sizeof(atoi(command[2])) + 1);
                if (ptr == NULL) {
                    printf("Error : Memory allocation failure\n");
                    continue;
                }
                ret = ReadFile(fd, ptr, atoi(command[2]));
                if (ret == -1)
                    printf("ERROR : File not existing\n");
                if (ret == -2)
                    printf("ERROR : Permission denied\n");
                if (ret == -3)
```

```
                    printf("ERROR : Reached at end of file\n");
                if (ret == -4)
                    printf("ERROR : It is not a regular file\n");
                if (ret == 0)
                    printf("ERROR : File empty\n");
                if (ret > 0) {
                    write(2, ptr, ret);
                }
                continue;
            } else {
                printf("\nERROR : Command not found !!!\n");
                continue;
            }
        } else if (count == 4) {
            if (strcmp(command[0], "lseek") == 0) {
                fd = GetFDFromName(command[1]);
                if (fd == -1) {
                    printf("Error : Incorrect parameter\n");
                    continue;
                }
                ret = LseekFile(fd, atoi(command[2]), atoi(command[3]));
                if (ret == -1) {
                    printf("ERROR : Unable to perform lseek\n");
                }
            } else {
                printf("\nERROR : Command not found !!!\n");
                continue;
            }
        } else {
            printf("\nERROR : Command not found !!!\n");
            continue;
        }
    }
    return 0;
}
```

System Calls Explanation

open

c

Copy code

```
/*
 * Name of function: open
 * Input parameters: const char *filename, int flags
 * Return value: int (file descriptor)
 * Description: Opens a file and returns a file descriptor.
 */
```

- Internal working: The open system call checks the file permissions, retrieves the inode, and updates the file table with the new file descriptor.

close

c

Copy code

```
/*
 * Name of function: close
 * Input parameters: int fd
 * Return value: int (0 for success, -1 for failure)
 * Description: Closes an open file descriptor.
 */
```

- Internal working: The close system call updates the file table, releasing the file descriptor and associated resources.

read

c

Copy code

```
/*
 * Name of function: read
 * Input parameters: int fd, void *buf, size_t count
 * Return value: ssize_t (number of bytes read)
 * Description: Reads data from a file descriptor into a buffer.
 */
```

- Internal working: The read system call reads data from the file's data blocks into the buffer, updating the file offset.

write

c

Copy code

```
/*
 * Name of function: write
 * Input parameters: int fd, const void *buf, size_t count
 * Return value: ssize_t (number of bytes written)
 * Description: Writes data from a buffer to a file descriptor.
 */
```

- Internal working: The write system call writes data from the buffer to the file's data blocks, updating the file offset and inode.

lseek

c

Copy code

```
/*
 * Name of function: lseek
 * Input parameters: int fd, off_t offset, int whence
 * Return value: off_t (new file offset)
 * Description: Repositions the file offset for a file descriptor.
 */
```

- Internal working: The lseek system call updates the file offset based on the specified parameters (offset and whence).

stat

c

Copy code

```
/*
 * Name of function: stat
 * Input parameters: const char *pathname, struct stat *statbuf
 * Return value: int (0 for success, -1 for failure)
 * Description: Retrieves file status information.
 */
```

- Internal working: The stat system call retrieves the inode information for the specified file and fills the stat structure.

chmod

c

Copy code

```
/*
 * Name of function: chmod
 * Input parameters: const char *pathname, mode_t mode
 * Return value: int (0 for success, -1 for failure)
 * Description: Changes the permissions of a file.
 */
```

- Internal working: The chmod system call updates the inode's mode field with the new permissions.

unlink

c

Copy code

```
/*
 * Name of function: unlink
 * Input parameters: const char *pathname
 * Return value: int (0 for success, -1 for failure)
 * Description: Removes a file from the file system.
 */
```

- Internal working: The unlink system call removes the directory entry, decreases the inode's link count, and frees the inode if the link count reaches zero.

Command Explanations

ls

- Lists files and directories in the current directory.

ls -l

- Lists files and directories in long format, showing detailed information.

ls -a

- Lists all files, including hidden files (those starting with a dot).

rm

- Removes files or directories.

cat

- Concatenates and displays file content.

cd

- Changes the current directory.

chmod

- Changes file permissions.

cp

- Copies files or directories.

df

- Displays disk space usage.

find

- Searches for files in a directory hierarchy.

grep

- Searches for patterns within files.

ln

- Creates hard and symbolic links.

mkdir

- Creates directories.

pwd

- Prints the current working directory.

touch

- Creates an empty file or updates the timestamp of an existing file.

uname

- Displays system information.

stat

- Displays file or file system status.

man

- Displays the manual page for a command.

mkfs

- Creates a file system on a storage device.

FAQ’s

What is a file system?

A file system is a method and data structure that an operating system uses to control how data is stored and retrieved.

Which file systems are used by Linux and Windows operating systems?

- Linux: ext4, xfs, btrfs
- Windows: NTFS, FAT32, exFAT

What are the parts of the file system?

- Superblock
- Inode Table
- Data Blocks
- Directory Structure
- File Allocation Table

Explain UAREA and its contents.

UAREA, or User Area, contains process-specific information such as open file descriptors, user credentials, and process state.

Explain the use of the File Table and its contents.

The File Table maintains the status of all open files, including file descriptors, pointers to the inode, and file access modes.

Explain the use of In-Core inode Table and its use.

The In-Core inode Table caches frequently accessed inodes to speed up file operations.

What does inode mean?

An inode is a data structure that represents a file in the file system, storing attributes like size, ownership, and pointers to data blocks.

What are the contents of the Superblock?

The Superblock contains metadata about the file system, such as its size, number of inodes, and block size.

What are the types of files?

- Regular files
- Directories
- Special files (character and block devices)
- Pipes
- Symbolic links
- Sockets

What are the contents of the inode?

- File type
- Permissions
- Timestamps (creation, modification, access)
- File size
- Pointers to data blocks

What is the use of a directory file?

A directory file contains entries that map filenames to inode numbers, organizing files in a hierarchical structure.

How does the operating system maintain security for files?

The OS uses permissions and access control lists (ACLs) to restrict access to files based on user identity and roles.

What happens when a user wants to open the file?

The OS checks permissions, retrieves the inode, updates the file table, and returns a file descriptor to the user.

What happens when a user calls lseek system call?

The lseek system call repositions the file offset for a file descriptor, allowing random access within a file.

What is the difference between a library function and a system call?

- Library Function: A function provided by the programming language's standard library, executed in user space.
- System Call: A function provided by the OS kernel, executed in kernel space, and involves a mode switch.

What is the use of this project?

This project demonstrates the implementation of a basic file system, providing an educational tool for understanding file system concepts and operations.

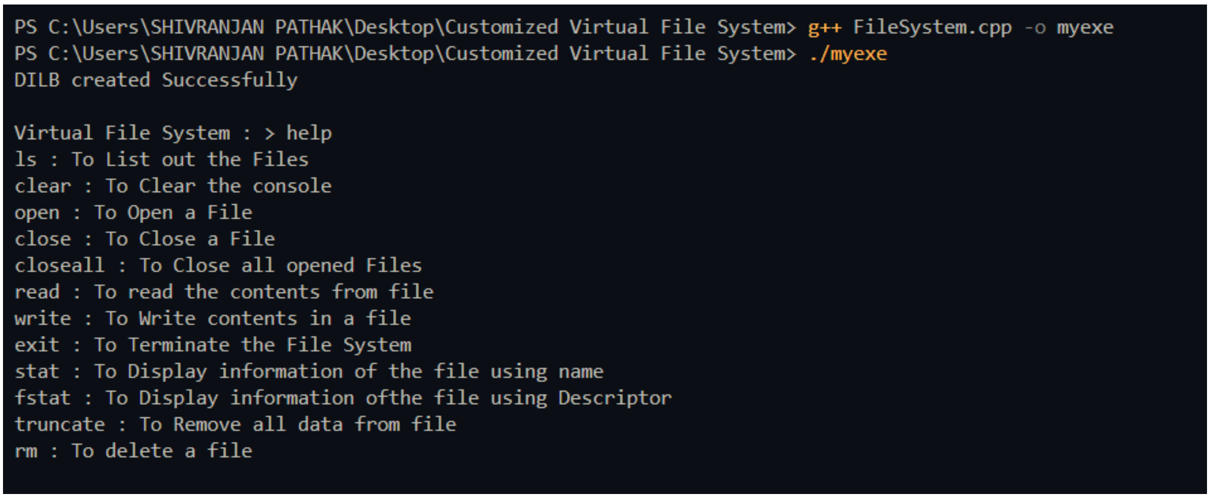
What are the difficulties that you faced in this project?

- Managing inode and block allocation
- Handling concurrent file access
- Ensuring data consistency and integrity

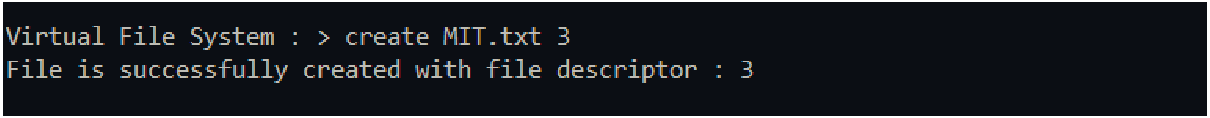
Is there any improvement needed in this project?

- Implementing advanced features like journaling for data recovery
- Enhancing performance through better caching mechanisms
- Adding support for more complex file operations

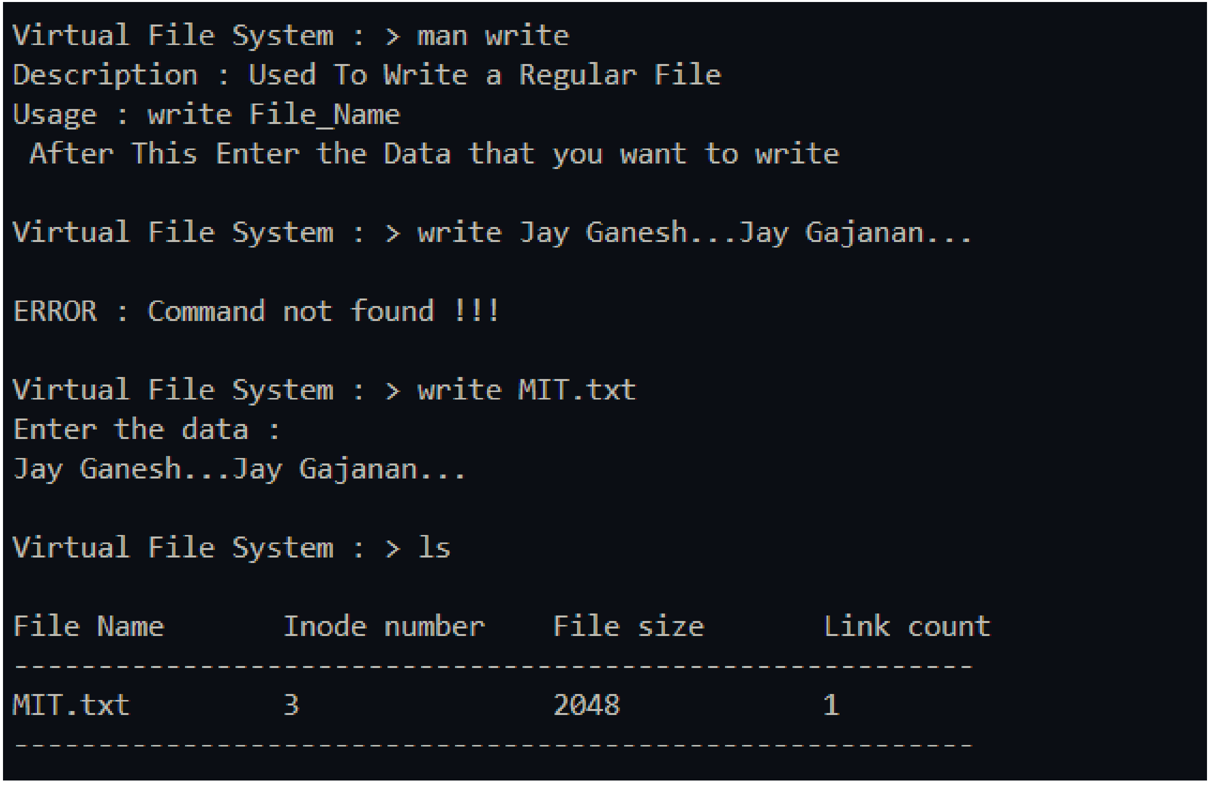
Output Screenshots



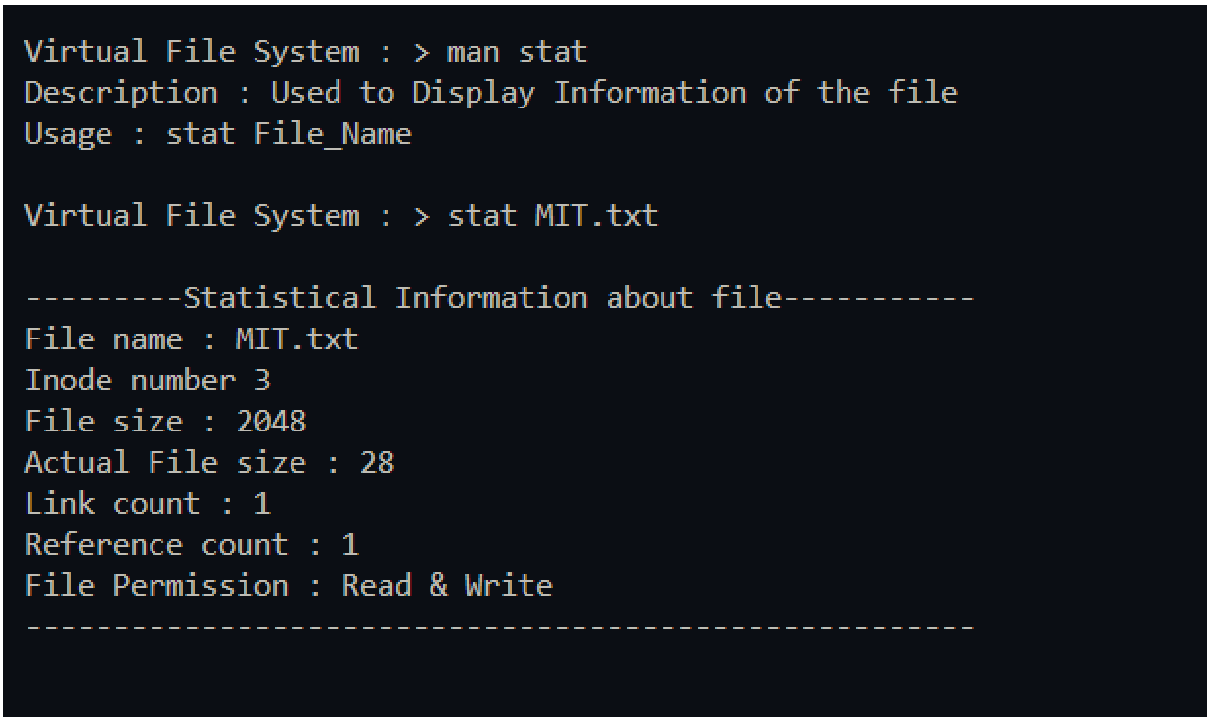
```
PS C:\Users\SHIVRANJAN PATHAK\Desktop\Customized Virtual File System> g++ FileSystem.cpp -o myexe
PS C:\Users\SHIVRANJAN PATHAK\Desktop\Customized Virtual File System> ./myexe
DILB created Successfully

Virtual File System : > help
ls : To List out the Files
clear : To Clear the console
open : To Open a File
close : To Close a File
closeall : To Close all opened Files
read : To read the contents from file
write : To Write contents in a file
exit : To Terminate the File System
stat : To Display information of the file using name
fstat : To Display information ofthe file using Descriptor
truncate : To Remove all data from file
rm : To delete a file
```



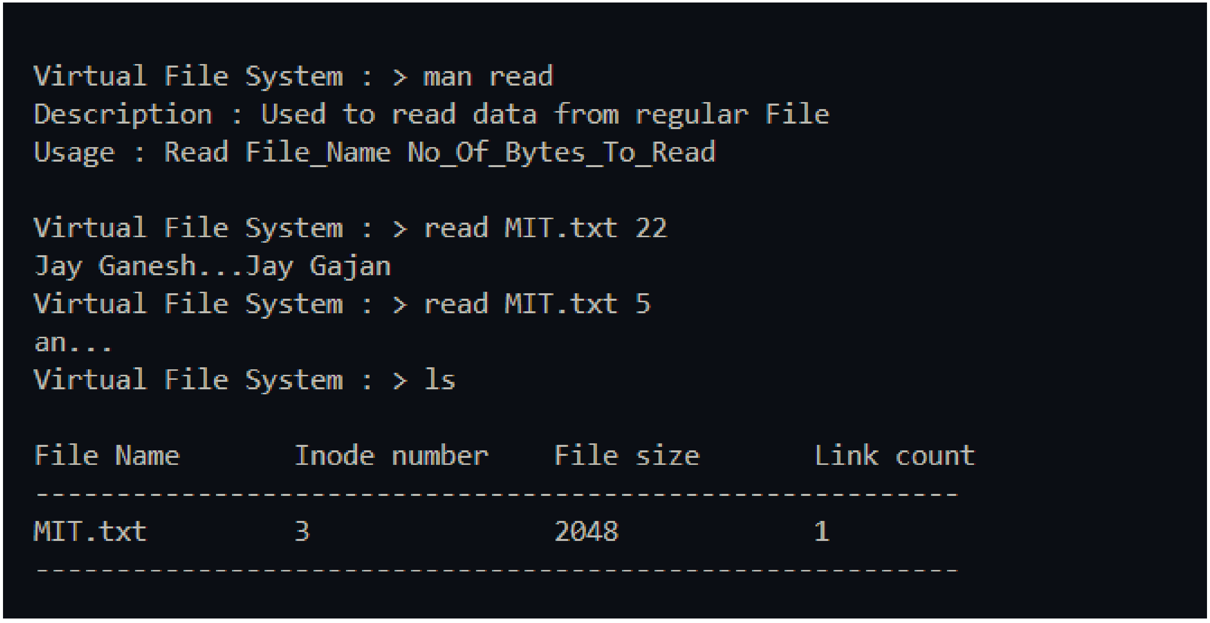
```
Virtual File System : > create MIT.txt 3
File is successfully created with file descriptor : 3
```



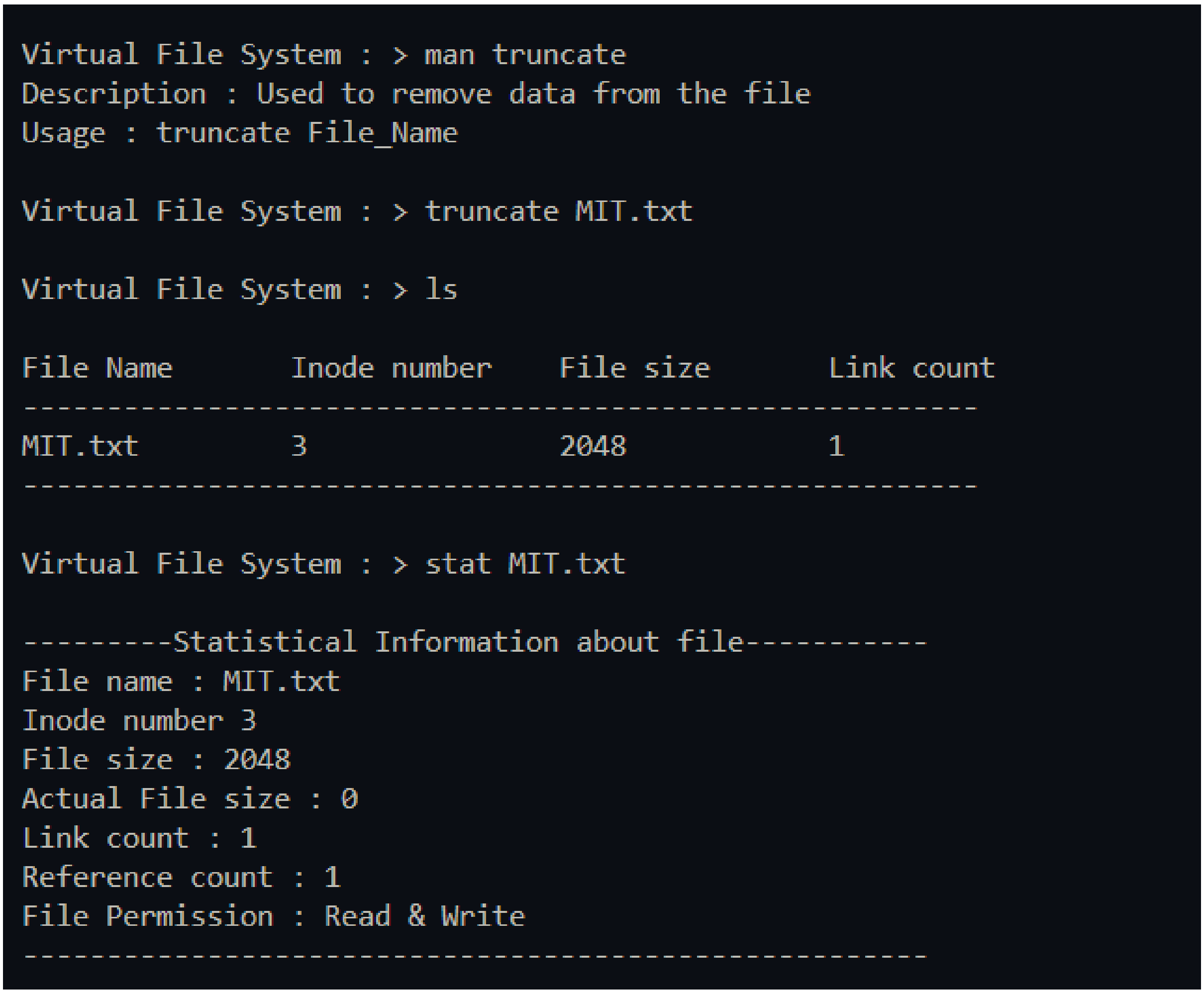
```
Virtual File System : > man write
Description : Used To Write a Regular File
Usage : write File_Name
 After This Enter the Data that you want to write

Virtual File System : > write Jay Ganesh...Jay Gajanan...

ERROR : Command not found !!!

Virtual File System : > write MIT.txt
Enter the data :
Jay Ganesh...Jay Gajanan...

Virtual File System : > ls

File Name       Inode number    File size       Link count
---------------------------------------------------------
MIT.txt         3               2048            1
---------------------------------------------------------
```



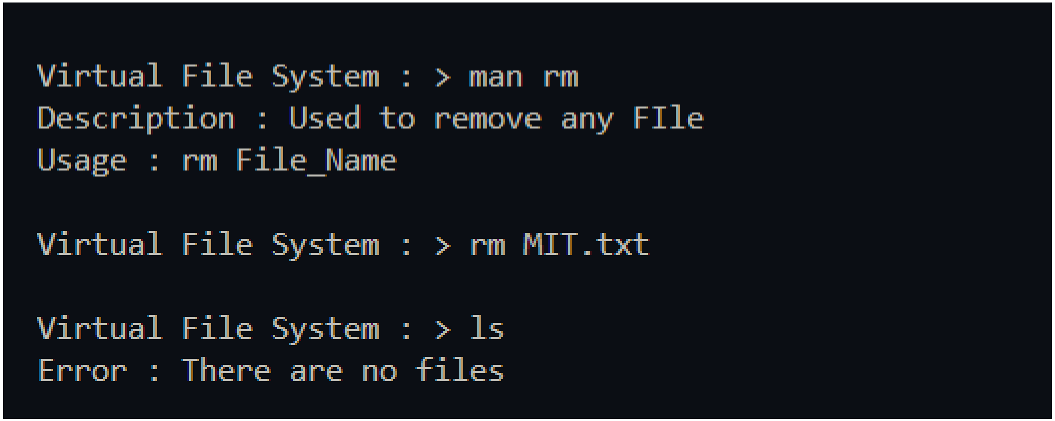
```
Virtual File System : > man stat
Description : Used to Display Information of the file
Usage : stat File_Name

Virtual File System : > stat MIT.txt

---------Statistical Information about file-----------
File name : MIT.txt
Inode number 3
File size : 2048
Actual File size : 28
Link count : 1
Reference count : 1
File Permission : Read & Write
------------------------------------------------------
```



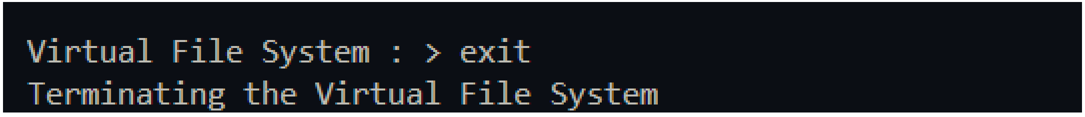
```
Virtual File System : > man read
Description : Used to read data from regular File
Usage : Read File_Name No_Of_Bytes_To_Read

Virtual File System : > read MIT.txt 22
Jay Ganesh...Jay Gajan
Virtual File System : > read MIT.txt 5
an...
Virtual File System : > ls

File Name       Inode number    File size       Link count
---------------------------------------------------------
MIT.txt         3               2048            1
---------------------------------------------------------
```



```
Virtual File System : > man truncate
Description : Used to remove data from the file
Usage : truncate File_Name

Virtual File System : > truncate MIT.txt

Virtual File System : > ls

File Name       Inode number    File size       Link count
---------------------------------------------------------
MIT.txt         3               2048            1
---------------------------------------------------------

Virtual File System : > stat MIT.txt

---------Statistical Information about file-----------
File name : MIT.txt
Inode number 3
File size : 2048
Actual File size : 0
Link count : 1
Reference count : 1
File Permission : Read & Write
------------------------------------------------------
```



```
Virtual File System : > man rm
Description : Used to remove any FIle
Usage : rm File_Name

Virtual File System : > rm MIT.txt

Virtual File System : > ls
Error : There are no files
```



```
Virtual File System : > exit
Terminating the Virtual File System
```