**Project Overview**

**Purpose**

The purpose of this project is to create a basic virtual file system that allows users to perform file operations such as creating, opening, reading, writing, and deleting files. It provides a simplified abstraction of a file system using data structures to manage files and their metadata.

**Components**

1. **Superblock:** Manages the overall file system state, including the total number of inodes and the number of free inodes.
2. **Inode:** Represents metadata about files, including file name, size, type, permissions, and data buffer.
3. **File Table:** Keeps track of open files, including file descriptors, read/write offsets, and file modes.
4. **User File Descriptor Table (UFDT):** Manages the file table entries for each open file, mapping file descriptors to file table entries.

**Data Structures**

* **SUPERBLOCK:** Contains information about the total number of inodes and the count of free inodes.
* **INODE:** Stores file-related metadata such as file name, size, type, permissions, and a pointer to the file's data buffer.
* **FILETABLE:** Keeps track of file operations such as read/write offsets, mode, and references to the inode.
* **UFDT:** An array of file tables, managing open files and their descriptors.

**Functionalities**

1. **Create File:** Allows users to create a new file with specified permissions.
2. **Open File:** Opens an existing file with a specified mode (read/write).
3. **Close File:** Closes an open file by its name or file descriptor.
4. **Read File:** Reads data from an open file into a buffer.
5. **Write File:** Writes data from a buffer to an open file.
6. **Delete File (rm):** Removes a file from the file system and frees up its inode.
7. **Truncate File:** Clears the content of a file but keeps the file itself.
8. **File Statistics (stat/fstat):** Displays metadata about a file, including its name, size, permissions, and reference count.
9. **List Files (ls):** Lists all files in the file system with their metadata.
10. **Change File Offset (lseek):** Adjusts the read/write offset in an open file.
11. **Close All Files:** Closes all open files in the file system.
12. **Help/Manual (man):** Provides descriptions and usage instructions for various commands.

**Commands and Usage**

* **create File\_name Permission:** Create a new file.
* **open File\_name mode:** Open an existing file.
* **close File\_name:** Close an open file.
* **read File\_name No\_Of\_Bytes:** Read data from a file.
* **write File\_name:** Write data to a file.
* **truncate File\_name:** Clear file content.
* **rm File\_name:** Delete a file.
* **stat File\_name:** Show file statistics.
* **fstat File\_Descriptor:** Show file statistics using file descriptor.
* **ls:** List all files.
* **lseek File\_Name ChangeInOffset StartPoint:** Change file offset.
* **closeall:** Close all open files.
* **help:** Show help for commands.
* **exit:** Terminate the virtual file system.

This project provides a basic framework for understanding file systems and their operations. It can be extended with more features like advanced file permissions, directories, and error handling.

**Each Hard Disk is divided into four parts:**

1. Boot Block (BB)
2. Super Block (SB)
3. Disk Inode List Block (DILB)
4. Data Block (DB)

**File System Structure:**

* File system is way of starting and representing the files in a particular format.
* NTFS, FAT 32, FAT 64, UFS are the types of files systems.
* When we create any file system the previously stored data gets overwritten and new file system get formed.

**Components of a File System:**

* **Every file system is divided into four parts.**

1. Boot Block (BB)
2. Super Block (SB)
3. Disk Inode List Block (DILB)
4. Data Block (DB)

* **BOOT BLOCK** is a block which contains the code which is used to start the file system.
* **INODE** is a structure which contains the complete information about the file.

Every file contains unique **INODE**.

* **Disk Inode List Block (DILB)** contains the linked list of all **INODEs.**
* **DATA BLOCK** is a block which contains actual data of the file.
* When we start our laptop the **boot block** gets active and it will start the **operating system**.
* When we double-click on the file the operating system will open its directory and fetch the **inode** number.
* From that inode number the actual inode gets fetch from the **DILB**.
* From that inode the operating system gets the block number and after getting the block number the operating system opens the data block and read the actual data of the file.

**Internal Implementation of File System:**

* **UAREA (User Area):**
* It’s a specific area allocated by the operating system to the process.
* Which contains important information about that process.
* For every process there is a separate UAREA.
* **UFDT (User File Descriptor Table):**
* It is an area of **pointers** which holds the address of the specific entry from the file table.
* First three entries from the UFDT are reserved for **Standard Input**, **Standard Output** and **Standard Error**.
* From the **index** number **3** we can use the entries.
* **FT (File Table):**
* File table is a table which contains information about the file that we count to open.
* File table contains the below entries like,

1. **Mode:** It contains the mode which we **open** the file.
2. **Offset:** It is a point from where we want **read** or **write** the data.
3. **Pointer:** Which points to the specific entry from **Incore Inode Table**.

* **IIT (Incore Inode Table):**
* It is a table which contains the **inodes** which are loaded into the memory.
* All the files inodes are present in IIT.
* The complete information of the inode is stored inside IIT.

**Data Structures in this project:**

The **Custom Virtual File System (CVFS) project** uses several key data structures to manage files, inodes, and other components effectively. Here’s a breakdown of the data structures used:

**1. Inode Structure**

* **Purpose**: Stores metadata about each file.
* **Fields**:
  + **File Name**: Name of the file.
  + **File Size**: Size of the file in bytes.
  + **File Type**: Type of the file (e.g., regular file, directory).
  + **Permissions**: Access permissions (read, write, execute).
  + **Block Pointers**: Pointers to the data blocks where the file content is stored.
  + **Link Count**: Number of references to the file.
  + **Timestamp**: Creation and modification times.

**2. Super Block Structure**

* **Purpose**: Stores overall file system metadata.
* **Fields**:
  + **Total Inodes**: Total number of inodes in the file system.
  + **Free Inodes**: Number of free inodes available.
  + **Total Data Blocks**: Total number of data blocks.
  + **Free Data Blocks**: Number of free data blocks available.

**3. Disk Inode List Block (DILB)**

* **Purpose**: Stores the list of inodes.
* **Fields**:
  + **Inode List**: A linked list or array of inodes.
  + **Pointer to Next Block**: If the inode list is spread across multiple blocks, a pointer to the next block.

**4. Data Block Structure**

* **Purpose**: Stores the actual data content of files.
* **Fields**:
  + **Data**: Raw data of the file.
  + **Block Number**: Unique identifier for each block.

**5. User Area (UAREA) Structure**

* **Purpose**: Stores process-specific information.
* **Fields**:
  + **Process ID**: Unique identifier for the process.
  + **File Descriptors**: Array of file descriptors opened by the process.
  + **Process State**: Current state of the process (e.g., running, waiting).

**6. User File Descriptor Table (UFDT)**

* **Purpose**: Maps file descriptors to file table entries.
* **Fields**:
  + **File Descriptor Array**: An array of pointers to file table entries.
  + **Reserved Entries**: First three entries reserved for standard input, output, and error.

**7. File Table (FT) Structure**

* **Purpose**: Maintains information about open files.
* **Fields**:
  + **Mode**: Mode in which the file is opened (read, write, etc.).
  + **Offset**: Current read/write position in the file.
  + **Inode Pointer**: Pointer to the inode of the open file.

**8. Incore Inode Table (IIT)**

* **Purpose**: Holds inodes currently loaded in memory.
* **Fields**:
  + **Incore Inode Array**: Array of inodes that are currently in use.
  + **Reference Count**: Number of file table entries referring to the inode.

These data structures work together to simulate a file system, enabling the management of files, directories, and associated metadata efficiently.

**The flow of Project:**

**1. Initialization**

1. **InitialiseSuperBlock():**
   * Initializes the UFDTArr array to NULL and sets SUPERBLOCKobj.ToatalInodes to MAXINODE and SUPERBLOCKobj.FreeInode to MAXINODE.
2. **CreateDILB():**
   * Creates a linked list of inodes (DILB - Disk Inode Linked Block) with MAXINODE elements. Each inode is initialized with default values and linked together.

**2. Command Loop (Shell)**

The program enters an infinite loop where it reads commands from the user and executes them:

1. **Read Input:**
   * Reads a command from the user using **fgets()**.
2. **Parse Command:**
   * Parses the command into different components **(command[0], command[1], etc.)** using **sscanf().**

**3. Handle Commands**

**Commands are handled based on their structure:**

**Commands with 1 Argument**

1. **ls:**
   * **Function:** Lists all files with their inode number, size, and link count.
   * **Function Call:** ls\_file()
2. **closeall:**
   * **Function:** Closes all open files.
   * **Function Call:** CloseAllFile()
3. **cls:**
   * **Function:** Clears the console (specific to Windows system("cls")).
4. **help:**
   * **Function:** Displays a help message listing all available commands.
   * **Function Call:** DisplayHelp()
5. **exit:**
   * **Function:** Terminates the program.
   * **Breaks:** Exits the loop.

**Commands with 2 Arguments**

1. **stat <file\_name>:**
   * **Function:** Displays information about a file by name.
   * **Function Call:** stat\_file(command[1])
2. **fstat <file\_descriptor>:**
   * **Function:** Displays information about a file by file descriptor.
   * **Function Call:** fstat\_file(atoi(command[1]))
3. **close <file\_name>:**
   * **Function:** Closes a file by its name.
   * **Function Call:** CloseFileByName(command[1])
4. **rm <file\_name>:**
   * **Function:** Removes a file.
   * **Function Call:** rm\_File(command[1])
5. **man <command>:**
   * **Function:** Displays the manual entry for a specific command.
   * **Function Call:** man(command[1])
6. **write <file\_name>:**
   * **Function:** Writes data to a file. User is prompted to enter the data after executing this command.
   * **Function Call:** WriteFile(fd, arr, ret)
7. **truncate <file\_name>:**
   * **Function:** Clears all data from a file.
   * **Function Call:** truncate\_File(command[1])

**Commands with 3 Arguments**

1. **create <file\_name> <permission>:**
   * **Function:** Creates a new file with the specified permissions.
   * **Function Call:** CreateFile(command[1], atoi(command[2]))
2. **open <file\_name> <mode>:**
   * **Function:** Opens a file with the specified mode (READ, WRITE, or READ+WRITE).
   * **Function Call:** OpenFile(command[1], atoi(command[2]))
3. **read <file\_name> <number\_of\_bytes>:**
   * **Function:** Reads data from a file into a buffer.
   * **Function Call:** ReadFile(fd, arr, isize)

**4. Detailed Command Functions**

1. **CreateFile(char \*name, int permission):**
   * Creates a new file with the given name and permissions if space is available and the file does not already exist.
2. **OpenFile(char \*name, int mode):**
   * Opens a file by name and sets the file's access mode.
3. **ReadFile(int fd, char \*arr, int isize):**
   * Reads data from a file specified by file descriptor fd into the buffer arr.
4. **WriteFile(int fd, char \*arr, int isize):**
   * Writes data from the buffer arr to the file specified by file descriptor fd.
5. **CloseFileByFD(int fd):**
   * Closes a file specified by file descriptor fd.
6. **CloseFileByName(char \*name):**
   * Closes a file specified by its name.
7. **CloseAllFile():**
   * Closes all open files.
8. **LseekFile(int fd, int size, int from):**
   * Changes the file offset for reading or writing based on from (START, CURRENT, or END).
9. **ls\_file():**
   * Lists all files with their metadata.
10. **fstat\_file(int fd):**
    * Displays file statistics for a file specified by its file descriptor.
11. **stat\_file(char \*name):**
    * Displays file statistics for a file specified by its name.
12. **truncate\_File(char \*name):**
    * Clears all data from a file specified by its name.

**Summary**

The program simulates a simple virtual file system using inodes and file tables. It allows basic file operations such as creating, opening, reading, writing, and deleting files through a command-line interface. It also provides functionality to manage and view file metadata and supports multiple file descriptors for simultaneous file operations.

**Actual Code and Explanation of the project.**

**#include<stdio.h>                   // Standard input/output library**

**#include<stdlib.h>                  // Standard library for memory allocation, process control, etc.**

**#include<string.h>                  // Library for string handling functions.**

**#include<unistd.h>                  // Unix standard library for operating system API**

**#include<iostream>                  // Input-output stream library for C++**

1. **#include<stdio.h>**
   * **Purpose:** This is the standard input/output library in C. It provides functionalities for file handling, input/output operations such as **printf, scanf, fgets,** etc.
   * **Example Functions:**
     + **printf():** Outputs formatted text to the console.
     + **scanf():** Reads formatted input from the console.
     + **fgets():** Reads a string from a file or standard input.
2. **#include<stdlib.h>**
   * **Purpose:** This is the standard library in C. It includes functions for dynamic memory allocation, process control, conversions, and other utility functions.
   * **Example Functions:**
     + **malloc():** Allocates memory dynamically.
     + **free():** Deallocates previously allocated memory.
     + **exit():** Terminates the program.
3. **#include<string.h>**
   * **Purpose:** This library provides functions to manipulate C-style strings and memory blocks. It includes functions for copying, concatenating, comparing, and manipulating strings.
   * **Example Functions:**
     + **strcpy():** Copies one string to another.
     + **strcat():** Concatenates two strings.
     + **strcmp():** Compares two strings.
     + **memset():** Fills a block of memory with a specific value.
4. **#include<unistd.h>**
   * **Purpose:** This library is part of the POSIX standard and provides access to the operating system's API, such as file operations, process control, and more. It's mainly used in Unix-like operating systems (**Linux, macOS, etc.**).
   * **Example Functions:**
     + **write():** Writes data to a file descriptor.
     + **read():** Reads data from a file descriptor.
     + **fork():** Creates a new process by duplicating the calling process.
     + **exec():** Replaces the current process image with a new process image.
5. **#include<iostream>**
   * **Purpose:** This is a C++ library for input/output stream handling. It is used for operations such as reading from the standard input and writing to the standard output.
   * **Example Objects:**
     + **std::cout:** Standard output stream, used to print data to the console.
     + **std::cin:** Standard input stream, used to get input from the user.
     + **std::endl:** Used to insert a newline character and flush the output buffer.

**Note:** Including both C and C++ headers like stdio.h and iostream together is uncommon unless you are writing code that mixes C and C++ features. C++ includes its own standard libraries (iostream, string, etc.) that provide similar functionalities with additional features. Mixing them can lead to confusion or unexpected behavior if not managed carefully.

**// Constants defining various values used in the filesystem simulation**

**#define MAXINODE 5                  // Maximum number of inodes**

**#define READ 1                      // Permission flag for read access**

**#define WRITE 2                     // Permission flag for write access**

**#define MAXFILESIZE 50              // Maximum file size in the simulated filesystem**

**#define REGULAR 1                   // File type flag for regular files**

**#define SPECIAL 2                   // File type flag for special files (not implemented in this code)**

**#define START 0                     // Seek start position flag (used in file pointer movement)**

**#define CURRENT 1                   // Seek current position flag**

**#define END 2                       // Seek end position flag**

1. **#define MAXINODE 5**
   * **Purpose:** This constant defines the maximum number of inodes that the simulated file system can handle. An inode is a data structure that stores information about a file or directory, such as its size, permissions, and location on disk.
   * **Use Case:** The file system will allow a maximum of 5 files to be created or managed simultaneously.
2. **#define READ 1**
   * **Purpose:** This constant is used as a flag to represent read permissions for files. A value of 1 indicates that the file can be read.
   * **Use Case:** It is used to check or set the read permissions when performing operations on a file.
3. **#define WRITE 2**
   * **Purpose:** This constant is used as a flag to represent write permissions for files. A value of 2 indicates that the file can be written to.
   * **Use Case:** It is used to check or set the write permissions when performing operations on a file.
4. **#define MAXFILESIZE 50**
   * **Purpose:** This constant defines the maximum size a file can have in the simulated file system. In this case, the maximum file size is 50 units (likely bytes).
   * **Use Case:** It limits the size of any file that can be created or modified within the file system.
5. **#define REGULAR 1**
   * **Purpose:** This constant is a flag that represents a regular file type. A regular file typically contains user data or text.
   * **Use Case:** It is used to differentiate between regular files and other types (such as special files, though special files are not implemented in this code).
6. **#define SPECIAL 2**
   * **Purpose:** This constant is a flag that represents a special file type. Special files could include device files, pipes, etc. However, this feature is not implemented in the current code.
   * **Use Case:** This constant is reserved for future use or expansion to handle special file types.
7. **#define START 0**
   * **Purpose:** This constant is used to define the starting position for file pointer operations, such as seeking to the start of the file.
   * **Use Case:** When performing file operations like lseek, this flag can be used to move the file pointer to the beginning of the file.
8. **#define CURRENT 1**
   * **Purpose:** This constant is used to define the current position of the file pointer for seek operations.
   * **Use Case:** It allows the file pointer to move relative to its current position.
9. **#define END 2**
   * **Purpose:** This constant is used to define the end position of the file for file pointer operations.
   * **Use Case:** It allows the file pointer to move relative to the end of the file, such as appending data to the file.

**// Structure representing a superblock, which stores file metadata**

**typedef struct superblock**

**{**

**int ToatalInodes;               // Total number of inodes in the filesystem**

**int FreeInode;                  // Number of free inodes available**

**}SUPERBLOCK, \*PSUPERBLOCK;          // 'SUPERBLOCK' for creating the structure and 'PSUPERBLOCK' for creating pointer to the structure.**

**Explanation**

1. **typedef struct superblock**
   * **Purpose:** This line begins the definition of a structure named superblock. The typedef keyword allows us to define a new type name (SUPERBLOCK) for this structure, making it easier to use in the code.
   * **Use Case:** Instead of writing struct superblock, we can simply use SUPERBLOCK when declaring variables of this type.
2. **int ToatalInodes;**
   * **Purpose:** This field stores the total number of inodes available in the simulated file system.
   * **Use Case:** The value here defines the upper limit for how many files or directories the file system can manage. Every file or directory has an associated inode.
3. **int FreeInode;**
   * **Purpose:** This field keeps track of how many inodes are still available for allocation.
   * **Use Case:** As files are created or deleted, this value is updated to reflect the number of inodes that can still be allocated. It is crucial for managing file creation and ensuring that the file system does not exceed its inode capacity.
4. **SUPERBLOCK, \*PSUPERBLOCK;**
   * **Purpose:** SUPERBLOCK is defined as an alias for the struct superblock, allowing easier and more readable code. PSUPERBLOCK is a pointer type that points to a SUPERBLOCK structure.
   * **Use Case:**
     + SUPERBLOCK can be used to declare variables of the structure type, e.g., SUPERBLOCK sb;.
     + PSUPERBLOCK can be used to declare pointers to such structures, e.g., PSUPERBLOCK psb;.

**Summary**

The **superblock** structure is essential in managing inodes within the file system. It tracks both the total number of inodes and the number of free inodes available, allowing the file system to manage file creation and deletion effectively. The use of typedef makes the code cleaner and easier to manage.

**This block of code defines an inode structure, which is a fundamental part of many file systems, responsible for storing metadata about files. The structure includes fields that describe various aspects of a file, such as its name, size, type, and permissions. Here's a detailed explanation:**

**Structure Definition**

**typedef struct inode**

**{**

**char FileName[50]; // Name of the file**

**int InodeNumber; // Unique inode number for each file**

**int FileSize; // Size of the file as per filesystem**

**int FileActualSize; // Actual size of the file data**

**int FileType; // Type of the file (regular or special)**

**char \*Buffer; // Pointer to the file data (actual content)**

**int LinkCount; // Number of links (references) to this inode**

**int ReferenceCount; // Reference count indicating how many processes are using this file**

**int permission; // Permission for the file (1 = Read, 2 = Write, 3 = Read\Write)**

**struct inode \*next; // Pointer to the next inode in the linked list**

**} INODE, \*PINODE, \*\*PPINODE; // 'INODE' for structure, 'PINODE' for pointer, 'PPINODE' for pointer to pointer**

**Explanation**

1. **char FileName[50];**
   * **Purpose:** Stores the name of the file.
   * **Use Case:** When a file is created, its name is stored here, allowing the file system to identify it.
2. **int InodeNumber;**
   * **Purpose:** A unique identifier (number) for the inode.
   * **Use Case:** This number uniquely identifies the file within the file system, making it possible to differentiate between files.
3. **int FileSize;**
   * **Purpose:** The size of the file as recognized by the file system.
   * **Use Case:** This value may be different from FileActualSize and is used by the file system for management tasks like allocation.
4. **int FileActualSize;**
   * **Purpose:** The actual size of the file data stored in the Buffer.
   * **Use Case:** This value reflects the true amount of data held within the file, essential for reading and writing operations.
5. **int FileType;**
   * **Purpose:** Indicates the type of file (e.g., regular file or special file).
   * **Use Case:** Helps the system understand how to handle the file, with typical values being REGULAR or SPECIAL.
6. **char \*Buffer;**
   * **Purpose:** A pointer to the actual content of the file stored in memory.
   * **Use Case:** This is where the file's data is stored, and it is accessed when reading from or writing to the file.
7. **int LinkCount;**
   * **Purpose:** The number of links (references) to this inode.
   * **Use Case:** When a file is hard-linked, this count increases. It helps manage file deletion; a file is only truly deleted when its LinkCount drops to zero.
8. **int ReferenceCount;**
   * **Purpose:** Tracks how many processes currently have this file open.
   * **Use Case:** Ensures that resources are properly managed, particularly during operations like closing a file or deleting an inode.
9. **int permission;**
   * **Purpose:** Specifies the file's permissions (e.g., read-only, write-only, read/write).
   * **Use Case:** Determines what operations are allowed on the file, controlling access at the file system level.
10. **struct inode \*next;**
    * **Purpose:** A pointer to the next inode in a linked list.
    * **Use Case:** This allows inodes to be linked together in a list, enabling traversal through all inodes in the file system.
11. **INODE, \*PINODE, \*\*PPINODE;**
    * **Purpose:** INODE is the type for the structure itself, PINODE is a pointer to an INODE, and PPINODE is a pointer to a pointer to an INODE.
    * **Use Case:**
      + INODE can be used to declare variables of this type, e.g., INODE inode;.
      + PINODE is useful for pointers to INODE, e.g., PINODE pinode = &inode;.
      + PPINODE is useful when a function needs to modify the pointer to an INODE, e.g., PPINODE ppinode;.

**Summary**

The **inode** structure encapsulates all the metadata needed to manage a file within the file system. It holds critical information like the file's name, size, type, permissions, and pointers to the actual data and other inodes. This design allows for efficient file management, ensuring that each file can be uniquely identified, accessed, and manipulated.

**This block of code defines a filetable structure, which is used to keep track of the status of an open file within the virtual file system. It includes various fields that represent the current state of a file when it is being accessed, such as the current read and write offsets, the mode in which the file was opened, and a pointer to the corresponding inode.**

**typedef struct filetable**

**{**

**int readoffset; // Current read offset for the file**

**int writeoffset; // Current write offset for the file**

**int count; // Number of processes using this file table entry**

**int mode; // Mode in which the file is opened (read\write\read-write)**

**PINODE ptrinode; // Pointer to the inode associated with this file**

**} FILETABLE, \*PFILETABLE; // 'FILETABLE' for structure, 'PFILETABLE' for pointer to the structure**

**Explanation**

1. **int readoffset;**
   * **Purpose:** Tracks the current position in the file where the next read operation will start.
   * **Use Case:** When data is read from the file, this offset is updated to reflect the new position in the file, ensuring that subsequent reads continue from where the last one left off.
2. **int writeoffset;**
   * **Purpose:** Tracks the current position in the file where the next write operation will occur.
   * **Use Case:** Similar to readoffset, this offset is updated as data is written to the file, allowing for appending or overwriting content at the correct location.
3. **int count;**
   * **Purpose:** Maintains the number of processes that are currently using this file table entry.
   * **Use Case:** This count is important for managing file access and ensuring that the file table entry is only closed or deallocated when no processes are using it.
4. **int mode;**
   * **Purpose:** Indicates the mode in which the file was opened (e.g., read-only, write-only, or read/write).
   * **Use Case:** The mode determines what operations are allowed on the file while it is open. For example, if the file is opened in read-only mode, write operations should be disallowed.
5. **PINODE ptrinode;**
   * **Purpose:** A pointer to the inode associated with this file.
   * **Use Case:** This pointer links the file table entry to the inode, allowing the file system to access the metadata and actual content of the file during operations like reading and writing.

**Summary**

The **filetable** structure is essential for managing open files in the virtual file system. It keeps track of where in the file the next read or write will happen, how many processes are using the file, and the mode in which the file was opened. By linking to the inode, the filetable structure allows the file system to efficiently manage and operate on open files.

**This block of code defines a structure named UFDT, which stands for User File Descriptor Table. This structure is used to manage file descriptors in a virtual file system, associating each file descriptor with a corresponding file table entry.**

**typedef struct ufdt**

**{**

**PFILETABLE ptrfiletable; // Pointer to the file table entry associated with this UFDT entry**

**} UFDT; // 'UFDT' for structure**

**Explanation**

1. **PFILETABLE ptrfiletable;**
   * **Purpose:** This field is a pointer to a filetable structure, which keeps track of the status of an open file, such as the current read and write offsets, file access mode, and other related information.
   * **Use Case:** Each UFDT entry points to a specific file table entry, allowing the virtual file system to manage multiple open files simultaneously. This connection is crucial for performing operations like reading, writing, and seeking within the correct file.

**Summary**

The UFDT structure represents an entry in the User File Descriptor Table, with each entry pointing to a filetable structure that holds the details of an open file. The UFDT is a key component in the virtual file system's management of file descriptors, allowing the system to keep track of which files are open and how they are being accessed.

**This block of code defines and initializes key global variables used in the virtual file system to manage files, inodes, and the overall file system state.**

**// Global variables**

**UFDT UFDTArr[50];                   // Array of UFDT entries, each representing an open file**

**SUPERBLOCK SUPERBLOCKobj;           // Instance of the superblock structure**

**PINODE head = NULL;                 // Head pointer for the linked list of inodes**

**Global Variables Explanation**

1. **UFDT UFDTArr[50];**
   * **Purpose:** This array holds 50 entries of the UFDT structure, with each entry representing an open file in the virtual file system.
   * **Functionality:** Each element of the array is associated with a file table entry through the ptrfiletable pointer. This allows the system to manage up to 50 open files simultaneously.
2. **SUPERBLOCK SUPERBLOCKobj;**
   * **Purpose:** This is an instance of the SUPERBLOCK structure, which keeps track of the overall state of the file system, including the total number of inodes and the number of free inodes available.
   * **Functionality:** The superblock is a critical part of the file system, as it provides essential metadata that governs the allocation and management of inodes.
3. **PINODE head = NULL;**
   * **Purpose:** This is a pointer to the head of a linked list of inodes. Each inode in the linked list represents a file within the file system.
   * **Functionality:** The linked list of inodes is used to keep track of all the files in the virtual file system. The head pointer serves as the starting point for traversing this list.

**Summary**

**These global variables form the backbone of the virtual file system:**

* **UFDTArr[]** manages the open files and their associated file table entries.
* **SUPERBLOCKobj** tracks the overall state and availability of inodes.
* **head** is the entry point to the linked list of inodes, representing all files in the system.

Together, they enable the basic operations and management of the file system.

**// Display manual information based on the command name**

**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 into regular file\n");**

**printf("Usage : write File\_name\n After this enter the data that we want to write\n");**

**}**

**else if(strcmp(name,"ls") == 0)**

**{**

**printf("Description : Used to list all information of flies\n");**

**printf("Usage : ls\n");**

**}**

**else if(strcmp(name,"stat") == 0)**

**{**

**printf("Description : Used to display information fo file\n");**

**printf("Usage : stat File\_name\n");**

**}**

**else if(strcmp(name,"fstat") == 0)**

**{**

**printf("Description : Used to display information fo file\n");**

**printf("Usage : stat File\_Descriptor\n");**

**}**

**else if(strcmp(name,"truncate") == 0)**

**{**

**printf("Description : Uset to remove data from file\n");**

**printf("Usage : truncate File\_name\n");**

**}**

**else if(strcmp(name,"open") == 0)**

**{**

**printf("Description : Used to open existing file\n");**

**printf("Usage : open File\_name mode\n");**

**}**

**else if(strcmp(name,"close") == 0)**

**{**

**printf("Description : Used to close opened file\n");**

**printf("Usage : close File\_name\n");**

**}**

**else if(strcmp(name,"closeall") == 0)**

**{**

**printf("Description : Used to closeall opened file\n");**

**printf("Usage : closeall\n");**

**}**

**else if(strcmp(name,"lseek") == 0)**

**{**

**printf("Description : Used to change file offse\n");**

**printf("Usage : lseek File\_Name ChangeInOffset StartPoint\n");**

**}**

**else if(strcmp(name,"rm") == 0)**

**{**

**printf("Description : Used to delete the file\n");**

**printf("Usage : rm File\_Name\n");**

**}**

**else**

**{**

**// If the command is not found, display an error message**

**printf("ERROR : No manual entry available.\n");**

**}**

**}**

**// Display a list of available commands and their purposes**

**void DisplayHelp()**

**{**

**printf("ls : To list out all files\n");**

**printf("clear : To clear console\n");**

**printf("create : To create new file\n");**

**printf("open : To open the file\n");**

**printf("close : To close the file\n");**

**printf("closeall : To close all opened files\n");**

**printf("read : To Read the contents from file\n");**

**printf("write : To Write contents into file\n");**

**printf("exit : To Terminate file system\n");**

**printf("stat : To Display information of file using name\n");**

**printf("fstat : To Display information of file using file descriptor\n");**

**printf("lseek : To change the file offset");**

**printf("truncate : To Remove all data from file\n");**

**printf("rm : To Delete the file\n");**

**}**

**Explanation: GetFDFromName function**

**// Function to get the file descriptor (FD) corresponding to a given file name**

**int GetFDFromName(char \*name)**

**{**

**int i = 0; // Initialize a counter to iterate through the UFDT array**

**// Loop through the UFDT array to find a file table entry matching the given name**

**while(i < 50) // The UFDT array has a maximum size of 50**

**{**

**// Check if the current UFDT entry's file table pointer is not NULL**

**if(UFDTArr[i].ptrfiletable != NULL)**

**{**

**// Compare the file name in the inode with the provided name**

**// If the names match, break out of the loop as the file is found**

**if(strcmp((UFDTArr[i].ptrfiletable->ptrinode->FileName), name) == 0)**

**{**

**break;**

**}**

**}**

**// Move to the next UFDT entry**

**i++;**

**}**

**// After the loop, check if we reached the end of the UFDT array without finding the file**

**if(i == 50) // If 'i' equals 50, it means the file was not found**

**{**

**return -1; // Return -1 to indicate the file was not found**

**}**

**else // If the loop was exited early due to finding the file**

**{**

**return i; // Return the index 'i' which corresponds to the file descriptor**

**}**

**}**

**Explanation:**

1. **Purpose**:
   * The GetFDFromName function is designed to locate the file descriptor (FD) for a file given its name. It searches through the User File Descriptor Table (UFDT) to find a matching file name and returns the index (FD) where the file is located.
2. **Working**:
   * **Initialization**:
     + A counter variable i is initialized to 0 to iterate through the UFDT array.
   * **Loop Through UFDT**:
     + The loop runs through all 50 possible entries in the UFDTArr array.
     + For each entry, it checks if the ptrfiletable is not NULL, indicating that the slot in the UFDT array is in use.
     + If the slot is in use, the function compares the file name stored in the inode (using ptrfiletable->ptrinode->FileName) with the name provided as an argument.
   * **Check for Match**:
     + If a match is found, the loop breaks, and the file descriptor (index i) is returned.
   * **File Not Found**:
     + If the loop completes without finding a match, the function returns -1 to indicate that the file does not exist in the UFDT array.
3. **Return Values**:
   * **File Descriptor (FD)**: The function returns the index of the file in the UFDT array, which acts as the file descriptor.
   * **-1**: If the file is not found in the array, the function returns -1.

This function is essential for translating a file's name into its corresponding file descriptor, enabling operations on the file within the system.

**Explanation: Get\_Inode function**

**// Function to get the inode structure pointer for a given file name**

**PINODE Get\_Inode(char \*name)**

**{**

**PINODE temp = head; // Start with the head of the inode list**

**int i = 0; // Initialize a counter (though not used in this function)**

**// Check if the provided file name is NULL**

**if(name == NULL)**

**return NULL; // If the name is NULL, return NULL since there's nothing to search for**

**// Traverse the inode linked list**

**while(temp != NULL) // Continue until the end of the list**

**{**

**// Compare the current inode's file name with the provided name**

**if(strcmp(name, temp->FileName) == 0)**

**break; // If a match is found, exit the loop**

**temp = temp->next; // Move to the next inode in the list**

**}**

**// Return the inode pointer**

**// If the file was found, this returns the pointer to the matching inode**

**// If not, it returns NULL (since temp would be NULL if the loop ended naturally)**

**return temp;**

**}**

**Explanation:**

1. **Purpose**:
   * The Get\_Inode function is designed to find and return the inode structure pointer (PINODE) associated with a given file name. The inode contains important metadata about the file, such as its size, type, and permissions.
2. **Working**:
   * **Initialization**:
     + A temporary pointer temp is set to point to the head of the inode linked list. This pointer will be used to traverse the list.
     + An integer i is initialized but is not actually used in this function, so it can be considered unnecessary here.
   * **Check for Null Name**:
     + The function first checks if the provided name is NULL. If it is, the function returns NULL immediately because a NULL name would mean there’s no valid file name to search for.
   * **Traverse the Inode List**:
     + The function enters a while loop that traverses the linked list of inodes, starting from the head.
     + For each inode in the list, the function compares the file name stored in that inode (temp->FileName) with the provided name using strcmp.
     + If a match is found, the loop breaks, meaning the function has found the inode associated with the file name.
   * **Return the Inode**:
     + If the loop finds a matching file name, the function returns the pointer temp, which points to the inode structure.
     + If the loop finishes without finding a match, temp will be NULL (since it will have traversed to the end of the list), and the function returns NULL.
3. **Return Values**:
   * **Inode Pointer (PINODE)**: If the file is found, the function returns a pointer to the corresponding inode structure.
   * **NULL**: If the file is not found or if the input name is NULL, the function returns NULL.

**Summary:**

* This function is crucial for locating the inode associated with a given file name within a file system. The inode contains the necessary information for file operations, so finding it is often the first step in any file manipulation process.

**Explanation:** **CreateDILB function**

**// Function to create the Disk Inode List Block (DILB)**

**// This initializes a linked list of inodes with a maximum of MAXINODE inodes**

**void CreateDILB()**

**{**

**int i = 1; // Initialize inode counter starting from 1**

**PINODE newn = NULL; // Pointer for new inode to be created**

**PINODE temp = head; // Temporary pointer to traverse the inode list**

**// Loop to create inodes until the maximum number of inodes (MAXINODE) is reached**

**while(i <= MAXINODE)**

**{**

**// Allocate memory for a new inode**

**newn = (PINODE)malloc(sizeof(INODE));**

**// Initialize the fields of the new inode**

**newn->LinkCount = 0; // No links to the inode initially**

**newn->ReferenceCount = 0; // No references to the inode initially**

**newn->FileType = 0; // FileType is set to 0 (could mean uninitialized or regular file)**

**newn->FileSize = 0; // Initial file size is 0**

**newn->Buffer = NULL; // Buffer is initially NULL, no data allocated yet**

**newn->next = NULL; // Next inode in the list is NULL (will be set later)**

**newn->InodeNumber = i; // Assign an inode number starting from 1**

**// Check if this is the first inode being created**

**if(temp == NULL)**

**{**

**head = newn; // Set the head of the list to this new inode**

**temp = head; // Update temp to point to the head**

**}**

**else**

**{**

**temp->next = newn; // Link the previous inode to the new inode**

**temp = temp->next; // Move temp to point to the newly created inode**

**}**

**i++; // Increment the inode counter**

**}**

**printf("DILB created successfully\n"); // Print a success message**

**}**

**Explanation:**

1. **Purpose**:
   * The CreateDILB function creates a linked list of inodes, often referred to as a Disk Inode List Block (DILB). Each inode in this list represents a file or a directory in a file system, holding metadata about the file.
2. **Initialization**:
   * **Inode Counter**: The function starts with i = 1 to assign sequential inode numbers.
   * **Pointers**:
     + newn is a pointer to the newly created inode.
     + temp is used to traverse the inode linked list. It initially points to the head of the list.
3. **Loop for Creating Inodes**:
   * The while loop runs until i exceeds MAXINODE, ensuring that the maximum number of inodes are created.
   * **Memory Allocation**:
     + Each iteration allocates memory for a new inode using malloc.
   * **Inode Initialization**:
     + Various fields of the inode are initialized:
       - LinkCount and ReferenceCount are set to 0, indicating no links or references initially.
       - FileType is set to 0, which might indicate an uninitialized state or a regular file type.
       - FileSize is 0, as the file is empty initially.
       - Buffer is set to NULL because no data is associated with the inode yet.
       - next is set to NULL, indicating that this is currently the last inode in the list.
       - InodeNumber is set to i, giving each inode a unique identifier.
4. **Linking Inodes**:
   * The function checks if temp is NULL to determine if this is the first inode being created:
     + If temp is NULL, this means the list is empty, so the new inode becomes the head of the list.
     + If the list already has inodes, the new inode is linked to the end of the list, and temp is updated to point to the newly created inode.
5. **Completion**:
   * After the loop completes, all inodes are created and linked in a list, and a success message is printed.

**Summary:**

* The **CreateDILB** function sets up a basic file system structure by creating a specified number of inodes, each of which can later be used to store file information. This function is fundamental for initializing the file system's metadata management.

**Explanation: InitialiseSuperBlock function**

**// Function to initialize the Superblock and the UFDT array**

**void InitialiseSuperBlock()**

**{**

**int i = 0; // Initialize the index for the UFDT array**

**// Loop through all entries in the UFDT array**

**while(i < MAXINODE)**

**{**

**UFDTArr[i].ptrfiletable = NULL; // Set each file table pointer to NULL, indicating no open files**

**i++; // Move to the next index**

**}**

**// Initialize the SUPERBLOCK object**

**SUPERBLOCKobj.ToatalInodes = MAXINODE; // Set the total number of inodes to MAXINODE**

**SUPERBLOCKobj.FreeInode = MAXINODE; // Set the number of free inodes to MAXINODE (initially all inodes are free)**

**}**

**Explanation:**

1. **Purpose**:
   * The InitialiseSuperBlock function initializes the file descriptor table (UFDT) and the superblock. This is crucial for setting up the file system's metadata and management structures.
2. **UFDT Initialization**:
   * **Index Initialization**: i is used to iterate over the UFDT array.
   * **Loop**: The while loop runs from i = 0 to i < MAXINODE, iterating through each entry in the UFDTArr array.
     + **Setting ptrfiletable to NULL**: Each entry in the UFDTArr array is initialized by setting the ptrfiletable pointer to NULL. This indicates that no files are currently open.
3. **Superblock Initialization**:
   * **Setting ToatalInodes**: This field is set to MAXINODE, representing the total number of inodes available in the file system.
   * **Setting FreeInode**: This field is also set to MAXINODE, indicating that initially all inodes are free and available for use.

**Summary:**

* The **InitialiseSuperBlock** function prepares the file system's data structures by setting up the UFDT to track file descriptors and initializing the superblock to manage inode information. This setup is essential for the file system to correctly manage file operations and metadata.

**Explanation: CreateFile function**

**// Function to create a new file with the given name and permissions**

**int CreateFile(char \*name, int permission)**

**{**

**int i = 3; // Initialize index for UFDT array; start from index 3**

**PINODE temp = head; // Pointer to traverse the inode linked list**

**// Validate input parameters**

**if((name == NULL) || (permission == 0) || (permission > 3))**

**return -1; // Return error code -1 for invalid input**

**// Check if there are free inodes available**

**if(SUPERBLOCKobj.FreeInode == 0)**

**return -2; // Return error code -2 if no free inodes are available**

**// Decrease the count of free inodes**

**(SUPERBLOCKobj.FreeInode)--;**

**// Check if the file with the same name already exists**

**if(Get\_Inode(name) != NULL)**

**return -3; // Return error code -3 if file already exists**

**// Find an unused inode**

**while(temp != NULL)**

**{**

**if(temp->FileType == 0)**

**break; // Found an unused inode**

**temp = temp->next; // Move to the next inode**

**}**

**// Find an unused slot in the UFDT array**

**while(i < 50)**

**{**

**if(UFDTArr[i].ptrfiletable == NULL)**

**break; // Found an unused UFDT slot**

**i++;**

**}**

**// Allocate memory for the file table entry**

**UFDTArr[i].ptrfiletable = (PFILETABLE)malloc(sizeof(FILETABLE));**

**// Initialize file table entry**

**UFDTArr[i].ptrfiletable->count = 1; // Set reference count to 1**

**UFDTArr[i].ptrfiletable->mode = permission; // Set file permissions**

**UFDTArr[i].ptrfiletable->readoffset = 0; // Initialize read offset**

**UFDTArr[i].ptrfiletable->writeoffset = 0; // Initialize write offset**

**UFDTArr[i].ptrfiletable->ptrinode = temp; // Assign the inode to the file table entry**

**// Initialize the inode**

**strcpy(UFDTArr[i].ptrfiletable->ptrinode->FileName, name); // Set file name**

**UFDTArr[i].ptrfiletable->ptrinode->FileType = REGULAR; // Set file type to regular**

**UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount = 1; // Set reference count**

**UFDTArr[i].ptrfiletable->ptrinode->LinkCount = 1; // Set link count**

**UFDTArr[i].ptrfiletable->ptrinode->FileSize = MAXFILESIZE; // Set file size**

**UFDTArr[i].ptrfiletable->ptrinode->FileActualSize = 0; // Set actual file size to 0**

**UFDTArr[i].ptrfiletable->ptrinode->permission = permission; // Set file permissions**

**UFDTArr[i].ptrfiletable->ptrinode->Buffer = (char \*)malloc(MAXFILESIZE); // Allocate memory for file data**

**return i; // Return the index of the file table entry**

**}**

**Explanation:**

1. **Purpose**:
   * The CreateFile function is designed to create a new file in the file system. It allocates resources, initializes the necessary structures, and returns the file descriptor.
2. **Parameter Validation**:
   * **Input Check**: The function checks if the name is NULL, if permission is 0 or greater than 3. If any of these conditions are met, it returns -1 indicating invalid input.
   * **Free Inodes Check**: It verifies if there are any free inodes available. If not, it returns -2.
3. **Inode Availability**:
   * **Existing File Check**: The function uses Get\_Inode to check if a file with the same name already exists. If found, it returns -3.
   * **Finding an Unused Inode**: It traverses the inode linked list to find an unused inode (where FileType is 0).
4. **File Descriptor Table Slot**:
   * **Finding an Unused UFDT Slot**: The function searches for an unused slot in the UFDTArr array starting from index 3.
5. **Memory Allocation and Initialization**:
   * **Allocate File Table Entry**: It allocates memory for a FILETABLE structure and initializes its fields.
   * **Initialize Inode**: The function sets the file name, type, reference count, link count, size, permissions, and allocates memory for the file's data buffer.
6. **Return Value**:
   * **File Descriptor**: The function returns the index of the UFDTArr entry where the new file's table entry is stored.

**Summary:**

* The **CreateFile** function handles the creation of a new file by validating input, finding resources, allocating necessary memory, and initializing data structures. It returns an index representing the file descriptor for further operations.

**Output of CreateFile Function on Command Prompt**

The output of the **CreateFile** function on the command prompt depends on the input parameters provided and the state of the virtual file system (CVFS). Below are some possible scenarios:

1. **Valid File Creation:**
   * **Command: create myfile 3**
   * **Output:**

**File created successfully with file descriptor 3**

* + **Description:** This output indicates that a new file named **myfile** was successfully created with permission 3. The file descriptor 3 is returned as the index in the **UFDTArr array** where the file's table entry is stored.

1. **Invalid Parameters (Null Name or Invalid Permission):**
   * **Command: create NULL 2**
   * **Output:**

**ERROR: Invalid input parameters**

* + **Description:** This error message indicates that the function received either a NULL name or an invalid permission value (e.g., 0 or greater than 3). As a result, the function returned -1, signaling an input validation error.

1. **No Free Inodes:**
   * **Command:** **create newfile 1**
   * **Output:**

**ERROR: No free inodes available**

* + **Description:** This error message means that the file system has run out of free inodes, and therefore no new file can be created. The function returns -2 to indicate this issue.

1. **File Already Exists:**
   * **Command:** **create existingfile 2**
   * **Output:**

**ERROR: File already exists**

* + **Description:** If a file with the name existingfile already exists in the file system, the function will return -3, and this error message will be displayed.

1. **File Descriptor Table Full:**
   * **Command:** **create newfile 1**
   * **Output:**

**ERROR: No available file descriptors**

* + **Description:** If the UFDTArr array is full (no empty slots from index 3 onward), the system cannot store the file descriptor for the new file. Although the current implementation doesn’t explicitly handle this case, it’s a possible scenario that could be addressed with additional checks.

**Summary**

The **CreateFile** function provides different outputs based on the state of the file system and the validity of input parameters. Successful file creation returns a file descriptor, while various errors can result from invalid inputs, lack of resources, or existing files. Each scenario produces a specific message on the command prompt to inform the user of the operation's result.

**D:\Marvellous\_Infosystems\Logic\_Building\_with\_Industrial\_Project\_Batch\Logic\_Building\_Project\CVFS Project>g++ CVFS.cpp -o myexe**

**D:\Marvellous\_Infosystems\Logic\_Building\_with\_Industrial\_Project\_Batch\Logic\_Building\_Project\CVFS Project>myexe.exe**

**DILB created successfully**

**Marvellous VFS : > man create**

**Description : Used to create new regular file**

**Usage : create File\_name Permission**

**Marvellous VFS : > create myfile 3**

**File is successfully created with flie descriptor : 3**

**Explanation: rm\_File function**

**// Function to remove a file from the file system**

**int rm\_File(char \*name)**

**{**

**int fd = 0; // File descriptor to store the result of GetFDFromName**

**// Get the file descriptor associated with the file name**

**fd = GetFDFromName(name);**

**// Check if file descriptor is invalid**

**if(fd == -1)**

**return -1; // Return error code -1 if file was not found**

**// Decrease the link count of the inode associated with the file**

**(UFDTArr[fd].ptrfiletable->ptrinode->LinkCount)--;**

**// Check if the link count of the inode has reached zero**

**if(UFDTArr[fd].ptrfiletable->ptrinode->LinkCount == 0)**

**{**

**// If link count is zero, mark the inode as unused**

**UFDTArr[fd].ptrfiletable->ptrinode->FileType = 0;**

**// Free the memory allocated for the file's buffer**

**free(UFDTArr[fd].ptrfiletable->ptrinode->Buffer);**

**// Free the memory allocated for the file table entry**

**free(UFDTArr[fd].ptrfiletable);**

**}**

**// Set the UFDT slot to NULL, indicating it's no longer in use**

**UFDTArr[fd].ptrfiletable = NULL;**

**// Increase the count of free inodes**

**(SUPERBLOCKobj.FreeInode)++;**

**return 0; // Return 0 to indicate success**

**}**

**Marvellous VFS : > man rm**

**Description : Used to delete the file**

**Usage : rm File\_Name**

**Marvellous VFS : > rm myfile**

**Explanation:**

1. **Purpose**:
   * The rm\_File function is used to remove a file from the file system by freeing its resources and updating the metadata.
2. **Retrieve File Descriptor**:
   * **GetFDFromName**: The function retrieves the file descriptor associated with the file name. If the file is not found, it returns -1 to indicate an error.
3. **Update Inode Information**:
   * **Link Count**: The function decrements the LinkCount of the inode. This tracks the number of references to the file.
4. **Check Link Count**:
   * **Link Count Zero**: If the LinkCount reaches zero, it means no references to the file exist. Therefore, the function:
     + Marks the inode as unused by setting FileType to 0.
     + Frees the allocated memory for the file's data buffer.
     + Frees the memory allocated for the file table entry.
5. **Update UFDT**:
   * **Clear UFDT Entry**: The function sets the corresponding UFDTArr slot to NULL, indicating that the file descriptor is no longer in use.
6. **Update Superblock**:
   * **Increase Free Inodes**: The function increments the count of free inodes in the superblock, making the inode available for future use.
7. **Return Value**:
   * **Success**: The function returns 0 to indicate that the file removal was successful.

**Summary:**

* The **rm\_File** function handles the removal of a file by updating its inode, freeing memory, and clearing file descriptor entries. It ensures that the file system remains consistent and that resources are properly managed.

**Output of rm\_File Function on Command Prompt**

The output of the **rm\_File** function on the command prompt will depend on whether the file to be removed exists and other conditions in the file system. Below are some possible scenarios:

1. **Successful File Removal:**
   * **Command: rm myfile**
   * **Output:**

**File 'myfile' removed successfully**

* + **Description:** This output indicates that the file named myfile was successfully removed from the file system. The function found the file descriptor associated with myfile, decremented the link count, and since the link count reached zero, it freed the associated memory, cleared the file descriptor table entry, and updated the superblock.

1. **File Not Found:**
   * **Command: rm nonexistentfile**
   * **Output:**

**ERROR: File 'nonexistentfile' not found**

* + **Description:** This error message indicates that the file named **nonexistentfile** does not exist in the file system. The function attempted to retrieve the file descriptor using **GetFDFromName,** but it returned -1, leading to this error message.

1. **File Descriptor Invalid (Already Removed or Not Opened):**
   * **Command: rm closedfile**
   * **Output:**

**ERROR: Invalid file descriptor**

* + **Description:** If the file descriptor is invalid (e.g., the file has already been removed or wasn't properly opened), the function will return an error message indicating that the file descriptor is invalid. This can happen if the file descriptor was previously set to NULL.

**Summary**

The **rm\_File** function provides feedback on the command prompt based on whether the file was successfully removed or if any errors occurred (e.g., the file not being found or an invalid file descriptor). Each scenario is accompanied by a clear message to inform the user of the operation's result. The output on the command prompt effectively communicates the success or failure of the file removal operation.

**Explanation: ReadFile function:**

**// Function to read data from a file**

**int ReadFile(int fd, char \*arr, int isize)**

**{**

**int read\_size = 0; // Variable to store the actual number of bytes to read**

**// Check if the file descriptor is valid**

**if(UFDTArr[fd].ptrfiletable == NULL)**

**{**

**return -1; // Return -1 if file descriptor is not valid (file not open)**

**}**

**// Check if the file is opened in read mode or read/write mode**

**if(UFDTArr[fd].ptrfiletable->mode != READ && UFDTArr[fd].ptrfiletable->mode != READ+WRITE)**

**{**

**return -2; // Return -2 if file is not opened in a mode that allows reading**

**}**

**// Check if the permissions of the file allow reading**

**if(UFDTArr[fd].ptrfiletable->ptrinode->permission != READ && UFDTArr[fd].ptrfiletable->ptrinode->permission != READ+WRITE)**

**{**

**return -2; // Return -2 if file does not have read permission**

**}**

**// Check if the read offset is at the end of the file**

**if(UFDTArr[fd].ptrfiletable->readoffset == UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)**

**{**

**return -3; // Return -3 if there is no data left to read (end of file)**

**}**

**// Check if the file type is regular (not a special file)**

**if(UFDTArr[fd].ptrfiletable->ptrinode->FileType != REGULAR)**

**{**

**return -4; // Return -4 if the file is not a regular file**

**}**

**// Calculate the number of bytes available to read**

**read\_size = (UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) - (UFDTArr[fd].ptrfiletable->readoffset);**

**// Check if the requested read size exceeds the available data size**

**if(read\_size < isize)**

**{**

**// If the requested size is greater than available size, read only the available size**

**strncpy(arr, (UFDTArr[fd].ptrfiletable->ptrinode->Buffer) + (UFDTArr[fd].ptrfiletable->readoffset), read\_size);**

**// Update the read offset**

**UFDTArr[fd].ptrfiletable->readoffset = UFDTArr[fd].ptrfiletable->readoffset + read\_size;**

**}**

**else**

**{**

**// If the requested size is within the available size, read the requested size**

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

**// Update the read offset**

**(UFDTArr[fd].ptrfiletable->readoffset) = (UFDTArr[fd].ptrfiletable->readoffset) + isize;**

**}**

**// Return the number of bytes read**

**return isize;**

**}**

**Explanation:**

1. **Purpose**:
   * The ReadFile function reads data from a file into a buffer and updates the file descriptor's read offset.
2. **Validation Checks**:
   * **File Descriptor Check**: Ensures the file descriptor is valid.
   * **Mode Check**: Verifies if the file is opened in a mode that allows reading.
   * **Permission Check**: Ensures the file has the appropriate permissions for reading.
   * **End of File Check**: Checks if the read offset has reached the end of the file.
   * **File Type Check**: Ensures the file is a regular file (not special).
3. **Read Size Calculation**:
   * **Available Data**: Calculates the amount of data available to read from the current offset to the end of the file.
   * **Data Copy**: Uses strncpy to copy data from the file buffer to the user-provided buffer (arr).
4. **Offset Update**:
   * **Adjust Read Offset**: Updates the read offset based on the number of bytes read.
5. **Return Value**:
   * **Bytes Read**: Returns the number of bytes that were read, which is isize if the requested size was successfully read, or the number of bytes available if the request exceeded available data.

**Summary:**

* The **ReadFile** function reads data from an open file, handling various edge cases such as permissions, file type, and end-of-file conditions. It updates the file descriptor's read offset and ensures that only the available data is read, managing buffer and file system integrity.

The **output** of the **ReadFile** function on the command prompt depends on the specific situation during the execution of the command. Below are the potential scenarios and corresponding outputs:

**Scenario 1: Successful Read**

**Command:** **read myFileName 10**

* **Condition:** The file myFileName is open, has more than or equal to 10 bytes available for reading, and all checks (mode, permission, etc.) pass.
* **Output:**

**10 bytes read from file**

This indicates that the function successfully read 10 bytes from the file myFileName.

**Scenario 2: File Not Open or Invalid File Descriptor**

**Command:** **read myFileName 10**

* **Condition:** The file myFileName is not open, or the file descriptor is invalid (e.g., ptrfiletable is NULL).
* **Output:**

**ERROR: File not open or invalid file descriptor**

This occurs when the function cannot find an open file associated with the provided file descriptor.

**Scenario 3: File Not Opened in a Readable Mode**

**Command:** **read myFileName 10**

* **Condition:** The file myFileName is open, but it is not opened in a mode that allows reading (e.g., it might be open only for writing).
* **Output:**

**ERROR: File not opened in a readable mode**

The function returns -2 because the file is not in a mode that allows reading.

**Scenario 4: File Lacks Read Permission**

**Command:** **read myFileName 10**

* **Condition:** The file myFileName does not have read permissions.
* **Output:**

**ERROR: File lacks read permissions**

This error is returned when the file's permissions do not include reading.

**Scenario 5: End of File Reached**

**Command:** **read myFileName 10**

* **Condition:** The read offset is at the end of the file (i.e., there's no more data left to read).
* **Output:**

**ERROR: End of file reached**

The function returns -3, indicating that there's no data left to read in the file.

**Scenario 6: File is Not a Regular File**

**Command:** **read myFileName 10**

* **Condition:** The file myFileName is not a regular file (e.g., it could be a directory or a special file).
* **Output:**

**ERROR: Not a regular file**

This output appears when the function determines that the file type is not regular.

**Scenario 7: Requested Size Exceeds Available Data**

**Command:** **read myFileName 100**

* **Condition:** The file myFileName is open, readable, and has less than 100 bytes left to read (e.g., only 50 bytes are left).
* **Output:**

**50 bytes read from file**

In this case, only the remaining bytes (50 in this example) are read, and the function returns the number of bytes actually read.

**Summary:**

The **ReadFile** function handles various cases based on file descriptor validity, permissions, and other conditions. The output on the command prompt provides feedback on whether the read operation was successful or why it failed.

**Marvellous VFS : > man read**

**Description : Used to read data from regular file**

**Usage : read File\_name No\_Of\_Bytes\_To\_Read**

**Marvellous VFS : > read file.txt 10**

**India is m**

**Explanation: WriteFile function**

**// Function to write data into a file**

**int WriteFile(int fd, char \*arr, int isize)**

**{**

**// Check if the file is opened in write or read/write mode**

**if(((UFDTArr[fd].ptrfiletable->mode) != WRITE) && ((UFDTArr[fd].ptrfiletable->mode) != READ+WRITE))**

**{**

**return -1; // Return -1 if file is not opened in a mode that allows writing**

**}**

**// Check if the file has the necessary write permissions**

**if(((UFDTArr[fd].ptrfiletable->ptrinode->permission) != WRITE) && ((UFDTArr[fd].ptrfiletable->ptrinode->permission) != READ+WRITE))**

**{**

**return -1; // Return -1 if file does not have write permissions**

**}**

**// Check if the write offset is at the maximum file size**

**if((UFDTArr[fd].ptrfiletable->writeoffset) == MAXFILESIZE)**

**{**

**return -2; // Return -2 if there is no more space left to write (file is full)**

**}**

**// Check if the file type is regular (not a special file)**

**if((UFDTArr[fd].ptrfiletable->ptrinode->FileType) != REGULAR)**

**{**

**return -3; // Return -3 if the file is not a regular file**

**}**

**// Copy data from the provided buffer (arr) to the file's buffer at the write offset**

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

**// Update the write offset by adding the number of bytes written**

**(UFDTArr[fd].ptrfiletable->writeoffset) = (UFDTArr[fd].ptrfiletable->writeoffset) + isize;**

**// Update the actual size of the file**

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

**// Return the number of bytes written**

**return isize;**

**}**

**Explanation:**

1. **Purpose**:
   * The WriteFile function writes data from a buffer into a file and updates the file descriptor's write offset and file size.
2. **Validation Checks**:
   * **Mode Check**: Ensures the file is opened in a mode that allows writing (WRITE or READ+WRITE).
   * **Permission Check**: Verifies the file has the appropriate write permissions.
   * **File Size Check**: Checks if the file is already full and cannot accommodate more data.
   * **File Type Check**: Ensures the file is a regular file (not a special file).
3. **Data Copy**:
   * **Write Operation**: Uses strncpy to copy data from the user-provided buffer (arr) to the file's buffer at the current write offset.
4. **Offset and Size Updates**:
   * **Write Offset**: Updates the write offset to reflect the position after the data is written.
   * **File Size**: Increases the file's actual size by the number of bytes written.
5. **Return Value**:
   * **Bytes Written**: Returns the number of bytes written, which is isize.

**Summary:**

* The **WriteFile** function is responsible for writing data into an open file, managing permissions, file size, and offsets to ensure the data is correctly written and the file system's integrity is maintained.

The **WriteFile** function writes data from a buffer into a file, handling various conditions and updating the file's metadata. The output of this function on the command prompt can vary based on the operation's success or failure. Below are possible scenarios and their corresponding outputs:

**Scenario 1: Successful Write**

**Command:** write myFileName bufferData 10

* **Condition:** The file myFileName is open in write or read/write mode, has write permissions, and is a regular file. There is enough space to write the data.
* **Output:**

**10 bytes written to file**

This indicates that the function successfully wrote 10 bytes from **bufferData** to the file **myFileName**.

**Scenario 2: File Not Opened in a Writable Mode**

**Command:** write myFileName bufferData 10

* **Condition:** The file myFileName is not open in a mode that allows writing (e.g., it is open in read-only mode).
* **Output:**

**ERROR: File not opened in a writable mode**

The function returns -1 because the file is not opened in a mode that allows writing.

**Scenario 3: File Lacks Write Permission**

**Command:** write myFileName bufferData 10

* **Condition:** The file myFileName does not have write permissions.
* **Output:**

**ERROR: File lacks write permissions**

This error occurs when the function determines that the file does not have the appropriate permissions for writing.

**Scenario 4: No Space Left to Write (File Full)**

**Command:** write myFileName bufferData 10

* **Condition:** The file myFileName is open in a writable mode, but the write offset has reached the maximum file size (MAXFILESIZE).
* **Output:**

**ERROR: No space left to write (file is full)**

The function returns -2, indicating that there is no more space left to write data into the file.

**Scenario 5: File is Not a Regular File**

**Command:** write myFileName bufferData 10

* **Condition:** The file myFileName is not a regular file (e.g., it could be a directory or a special file).
* **Output:**

**ERROR: Not a regular file**

This output appears when the function determines that the file type is not regular.

**Scenario 6: Partial Write Due to Limited Space**

**Command:** write myFileName bufferData 50

* **Condition:** The file myFileName has limited space available, and only part of the requested 50 bytes can be written.
* **Output:**

**X bytes written to file**

In this case, the function writes as much data as possible within the available space and returns the actual number of bytes written, denoted here as X.

**Summary:**

The **WriteFile** function handles various conditions such as file mode, permissions, available space, and file type. The output on the command prompt provides feedback on whether the write operation was successful or why it failed. The function returns the number of bytes successfully written or an error code to indicate specific issues.

**Marvellous VFS : > man write**

**Description : Used to write into regular file**

**Usage : write File\_name**

**After this enter the data that we want to write**

**Marvellous VFS : > write myfile.txt**

**Enter the data :**

**India is my country**

**Explanation: OpenFile function**

**// Function to open a file and return its file descriptor**

**int OpenFile(char \*name, int mode)**

**{**

**int i = 0;**

**PINODE temp = NULL;**

**// Validate input parameters**

**if(name == NULL || mode <= 0)**

**{**

**return -1; // Return -1 if invalid input parameters**

**}**

**// Get the inode (file information) using the file name**

**temp = Get\_Inode(name);**

**// Check if the file does not exist**

**if(temp == NULL)**

**{**

**return -2; // Return -2 if file does not exist**

**}**

**// Check if the mode requested exceeds the file's permissions**

**if(temp->permission < mode)**

**{**

**return -3; // Return -3 if the file does not have the requested permissions**

**}**

**// Find an empty slot in the UFDT (User File Descriptor Table) array**

**while(i < 50)**

**{**

**if(UFDTArr[i].ptrfiletable == NULL)**

**{**

**break; // Found an empty slot**

**}**

**i++;**

**}**

**// Check if no empty slot was found**

**if(i == 50)**

**{**

**return -4; // Return -4 if the UFDT is full (no available slots)**

**}**

**// Allocate memory for the file table entry**

**UFDTArr[i].ptrfiletable = (PFILETABLE)malloc(sizeof(FILETABLE));**

**// Check if memory allocation failed**

**if(UFDTArr[i].ptrfiletable == NULL)**

**{**

**return -1; // Return -1 if memory allocation failed**

**}**

**// Initialize the file table entry**

**UFDTArr[i].ptrfiletable->count = 1; // File descriptor count**

**UFDTArr[i].ptrfiletable->mode = mode; // Access mode**

**// Initialize file offsets based on the 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;**

**}**

**// Set the inode pointer and increment the reference count**

**UFDTArr[i].ptrfiletable->ptrinode = temp;**

**(UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)++;**

**// Return the file descriptor (index in UFDT array)**

**return i;**

**}**

**Explanation:**

1. **Purpose**:
   * The OpenFile function opens a file, allocates a file descriptor for it, and initializes the file table entry with relevant information.
2. **Validation Checks**:
   * **Input Parameters**: Ensures the file name is provided and the mode is valid.
   * **File Existence**: Checks if the file exists by retrieving its inode.
   * **Permissions**: Verifies the requested access mode is permitted by the file's permissions.
3. **Finding an Available Slot**:
   * Searches the UFDTArr (User File Descriptor Table Array) for an empty slot to allocate a new file descriptor.
4. **Memory Allocation**:
   * Allocates memory for a new FILETABLE structure to manage the file's information.
5. **Initialization**:
   * Sets the file descriptor count and mode.
   * Initializes read and write offsets based on the access mode.
   * Updates the inode reference count to track how many file descriptors are referring to the file.
6. **Return Value**:
   * Returns the index of the allocated file descriptor if successful.
   * Returns specific error codes if any conditions are not met.

**Summary:**

* The **OpenFile** function manages the process of opening a file by checking for validity, allocating a file descriptor, initializing file access parameters, and ensuring proper permission handling. It returns the file descriptor if successful or an appropriate error code if something goes wrong.

**Expected Output on the Command Prompt**

The output of the **OpenFile()** function on the command prompt will depend on whether the file exists, the validity of the access mode, and whether there is an available slot in the UFDTArr.

**Example 1: File Does Not Exist**

**Marvellous VFS : > open non\_existent\_file 1**

**Error : File does not exist**

* **Description:**
  + If the file name provided (e.g., "non\_existent\_file") does not exist in the file system, the function returns -2, and the program prints an error message indicating that the file does not exist.

**Example 2: Invalid Mode**

**Marvellous VFS : > open existing\_file -1**

**Error : Invalid mode**

* **Description:**
  + If the mode provided is invalid (e.g., -1), the function returns -1, and the program prints an error message indicating that the mode is invalid.

**Example 3: No Available File Descriptor**

**Marvellous VFS : > open file1 1**

**Error : No free file descriptor available**

* **Description:**
  + If there are no available slots in the UFDTArr, the function returns -4, and the program prints an error message indicating that no free file descriptor is available.

**Example 4: File Opened Successfully**

**Marvellous VFS : > open file1 1**

**File opened successfully with file descriptor: 3**

* **Description:**
  + If the file exists, the mode is valid, and an available slot is found, the function successfully opens the file and returns the file descriptor (e.g., 3). The program prints a message indicating the file was opened successfully with the given file descriptor.

This output shows whether the file was opened successfully and informs the user of the status of the specified file and access mode.

**Explanation: CloseFileByFD function**

**// Function to close a file by its file descriptor**

**void CloseFileByFD(int fd)**

**{**

**// Reset the read offset to the beginning of the file**

**UFDTArr[fd].ptrfiletable->readoffset = 0;**

**// Reset the write offset to the beginning of the file**

**UFDTArr[fd].ptrfiletable->writeoffset = 0;**

**// Decrement the reference count of the inode**

**// This indicates that one less file descriptor is referring to this file**

**(UFDTArr[fd].ptrfiletable->ptrinode->ReferenceCount)--;**

**}**

**Explanation:**

1. **Purpose**:
   * The CloseFileByFD function is used to close a file that was previously opened using its file descriptor (fd).
2. **Resetting Offsets**:
   * **Read Offset**: The readoffset is reset to 0, which means that if the file is opened again for reading, it will start from the beginning.
   * **Write Offset**: Similarly, the writeoffset is reset to 0, so that future write operations will start from the beginning of the file.
3. **Reference Count Decrement**:
   * **Reference Count**: The ReferenceCount of the file's inode is decremented by 1. This count tracks how many active file descriptors are referring to this particular file. When the count reaches 0, it indicates that no file descriptors are currently referring to the file, potentially allowing the file to be deleted or deallocated if necessary.

**Summary:**

* The **CloseFileByFD** function performs the essential task of closing a file by resetting its read and write offsets and updating the inode's reference count. This function helps in managing the file's state within the file system, ensuring that resources are properly tracked and managed.

**Expected Output on the Command Prompt**

**Marvellous VFS : > open myfile 3**

**File is opened successfully with file descriptor: 5**

**Marvellous VFS : > close 5**

1. **Opening the File:**
   * The file "myfile" is opened successfully with the file descriptor 5. The output for this would be:

**File is opened successfully with file descriptor: 5**

1. **Closing the File:**
   * When the command close 5 is executed, the CloseFileByFD(5) function is called. The function will reset the readoffset and writeoffset of the file associated with file descriptor 5, and decrement the ReferenceCount of the inode.
   * Since the CloseFileByFD function itself does not produce direct output to the command prompt (it’s a void function), there won’t be any textual output after this command. The absence of any errors or further messages would imply that the file was closed successfully.

**Description of Output:**

* **Open File Output:**
  + "File is opened successfully with file descriptor: 5" indicates that the file "myfile" has been opened, and the system allocated file descriptor 5 for it.
* **Close File Output:**
  + No output is directly provided by the CloseFileByFD function. The lack of an error message after executing close 5 suggests the operation completed successfully, meaning the file descriptor 5 is no longer active, and the associated file has been properly closed in the system.

This sequence ensures that the file is properly managed within the virtual file system, maintaining stability and resource management.

**Explanation: CloseFileByName function**

**// Function to close a file by its name**

**int CloseFileByName(char \*name)**

**{**

**int i = 0;**

**// Get the file descriptor corresponding to the file name**

**i = GetFDFromName(name);**

**// If the file descriptor is invalid (file not found), return -1**

**if(i == -1)**

**return -1;**

**// Reset the read offset to the beginning of the file**

**UFDTArr[i].ptrfiletable->readoffset = 0;**

**// Reset the write offset to the beginning of the file**

**UFDTArr[i].ptrfiletable->writeoffset = 0;**

**// Decrement the reference count of the inode**

**(UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)--;**

**// Return 0 indicating successful closure of the file**

**return 0;**

**}**

**Explanation:**

1. **Purpose**:
   * The CloseFileByName function is used to close a file by its name. It performs similar actions to CloseFileByFD, but instead of using a file descriptor (fd), it uses the file's name to identify which file to close.
2. **Getting the File Descriptor**:
   * The function first calls GetFDFromName(name) to get the file descriptor associated with the provided file name. If the file is not found, GetFDFromName returns -1, and the function immediately returns -1 to indicate failure.
3. **Resetting Offsets**:
   * If a valid file descriptor is found, the function resets the file's readoffset and writeoffset to 0. This ensures that the next time the file is opened for reading or writing, operations will start from the beginning of the file.
4. **Reference Count Decrement**:
   * The function decrements the ReferenceCount of the file's inode. This count tracks how many active file descriptors are referring to this file. Reducing the count helps manage resources, as the file may be closed or removed from memory when the count reaches 0.
5. **Return Value**:
   * The function returns 0 to indicate that the file was successfully closed by its name.

**Summary:**

* The **CloseFileByName** function provides a convenient way to close a file by using its name. It ensures that the file's state is properly managed by resetting offsets and updating the inode's reference count, making the system more robust and resource-efficient.

**Expected Output on the Command Prompt**

**Marvellous VFS : > create myfile 3**

**File is successfully created with file descriptor: 3**

**Marvellous VFS : > open myfile 3**

**File is opened successfully with file descriptor: 5**

**Marvellous VFS : > close myfile**

**File closed successfully by name: myfile**

1. **Create a File:**
   * When the file "myfile" is created with the permission 3, it is assigned a file descriptor (e.g., 3). The output would be:

**File is successfully created with file descriptor: 3**

1. **Open the File:**
   * The file "myfile" is opened with permission 3, and it is assigned a different file descriptor (e.g., 5). The output for this command would be:

**File is opened successfully with file descriptor: 5**

1. **Close the File by Name:**
   * When the command close myfile is executed, the CloseFileByName("myfile") function is called. The function finds the file descriptor associated with "myfile" using GetFDFromName, then resets the read and write offsets to 0, decrements the inode's reference count, and returns 0 to indicate success.
   * If the function is successful, it will typically show a message similar to:

**File closed successfully by name: myfile**

**Description of Output:**

* **Create File Output:**
  + "File is successfully created with file descriptor: 3" indicates that the file "myfile" was created successfully with file descriptor 3.
* **Open File Output:**
  + "File is opened successfully with file descriptor: 5" shows that the file "myfile" was opened, and the system assigned it the file descriptor 5.
* **Close File by Name Output:**
  + "File closed successfully by name: myfile" confirms that the file "myfile" was successfully closed using its name.

**Explanation: CloseAllFile function**

**// Function to close all open files**

**void CloseAllFile()**

**{**

**int i = 0;**

**// Iterate through the entire UFDT array to check for open files**

**while(i < MAXFILESIZE)**

**{**

**// Check if there is an active file table entry at this index**

**if(UFDTArr[i].ptrfiletable != NULL)**

**{**

**// Reset the read offset to the beginning of the file**

**UFDTArr[i].ptrfiletable->readoffset = 0;**

**// Reset the write offset to the beginning of the file**

**UFDTArr[i].ptrfiletable->writeoffset = 0;**

**// Decrement the reference count of the inode**

**(UFDTArr[i].ptrfiletable->ptrinode->ReferenceCount)--;**

**// Continue closing other files**

**// The `break` should not be here; it should be removed to allow all files to be closed**

**}**

**i++;**

**}**

**}**

**Explanation:**

1. **Purpose**:
   * The CloseAllFile function is intended to close all open files in the system. This involves iterating over the UFDT (User File Descriptor Table) array, checking for active file entries, and resetting their states.
2. **Iterating Through UFDT**:
   * The function uses a while loop to iterate through each index of the UFDTArr. It checks if there is a non-NULL ptrfiletable, which indicates an open file.
3. **Resetting Offsets**:
   * For each active file entry, the function resets both the readoffset and writeoffset to 0. This ensures that any subsequent operations on these files start from the beginning.
4. **Decrementing Reference Count**:
   * The function decrements the ReferenceCount of the file's inode. If this count reaches zero, the system might later decide to release the resources associated with the file.
5. **Error in Code**:
   * There’s a mistake in the code: the break statement should not be there. The break will cause the loop to terminate after closing the first open file it encounters. This means only one file will be closed, which contradicts the function's purpose of closing all files.
   * **Correction**: The break statement should be removed so that the loop continues until all open files are closed.

**Expected Output on the Command Prompt**

If you compile and run the program, and execute the **CloseAllFile** function with the erroneous break statement in place, the following will happen:

**Example Scenario:**

**Marvellous VFS : > create file1 3**

**File is successfully created with file descriptor: 3**

**Marvellous VFS : > create file2 3**

**File is successfully created with file descriptor: 4**

**Marvellous VFS : > create file3 3**

**File is successfully created with file descriptor: 5**

**Marvellous VFS : > closeall**

**Only the first open file closed: file1**

1. **Create Multiple Files:**
   * Files "file1", "file2", and "file3" are created and assigned file descriptors 3, 4, and 5, respectively.
2. **Execute CloseAllFile:**
   * When the closeall command is issued, only "file1" is closed due to the break statement. The output on the command prompt will indicate that the first file was closed, but the remaining files will stay open.
   * Possible output:

Only the first open file closed: file1

**Description of Output**

* **Create File Output:**
  + "File is successfully created with file descriptor: 3" confirms that "file1" was created and assigned descriptor 3.
  + Similar messages for "file2" and "file3" with descriptors 4 and 5, respectively.
* **CloseAllFile Output:**
  + If the CloseAllFile function is run with the erroneous break in place, only "file1" will be closed. The output would reflect that "file1" was closed, but it might also indicate that the operation did not complete as expected since the remaining files are still open.

**Explanation: LseekFile function**

**// Function to adjust the file offset (seek) for reading or writing**

**int LseekFile(int fd, int size, int from)**

**{**

**// Validate file descriptor and 'from' parameter**

**if((fd < 0) || (from > 2))**

**{**

**return -1; // Invalid input**

**}**

**// Check if the file table entry exists for the given file descriptor**

**if(UFDTArr[fd].ptrfiletable == NULL)**

**{**

**return -1; // File is not open**

**}**

**// Handle read and read/write modes**

**if((UFDTArr[fd].ptrfiletable->mode == READ) || (UFDTArr[fd].ptrfiletable->mode == READ + WRITE))**

**{**

**// Seek relative to the current position**

**if(from == CURRENT)**

**{**

**// Validate the new offset**

**if(((UFDTArr[fd].ptrfiletable->readoffset) + size) > UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)**

**{**

**return -1; // Seeking beyond file size**

**}**

**if(((UFDTArr[fd].ptrfiletable->readoffset) + size) < 0)**

**{**

**return -1; // Seeking before the start of the file**

**}**

**// Update the read offset**

**UFDTArr[fd].ptrfiletable->readoffset += size;**

**}**

**// Seek relative to the start of the file**

**else if(from == START)**

**{**

**// Validate the new offset**

**if(size > UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)**

**{**

**return -1; // Seeking beyond file size**

**}**

**if(size < 0)**

**{**

**return -1; // Negative offset**

**}**

**// Set the read offset**

**UFDTArr[fd].ptrfiletable->readoffset = size;**

**}**

**// Seek relative to the end of the file**

**else if(from == END)**

**{**

**// Validate the new offset**

**if((UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size > MAXFILESIZE)**

**{**

**return -1; // Seeking beyond maximum file size**

**}**

**if(((UFDTArr[fd].ptrfiletable->readoffset) + size) < 0)**

**{**

**return -1; // Negative offset**

**}**

**// Set the read offset relative to the end of the file**

**UFDTArr[fd].ptrfiletable->readoffset = UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize + size;**

**}**

**}**

**// Handle write mode**

**else if(UFDTArr[fd].ptrfiletable->mode == WRITE)**

**{**

**// Seek relative to the current position**

**if(from == CURRENT)**

**{**

**// Validate the new offset**

**if((UFDTArr[fd].ptrfiletable->writeoffset + size) > MAXFILESIZE)**

**{**

**return -1; // Seeking beyond maximum file size**

**}**

**if((UFDTArr[fd].ptrfiletable->writeoffset + size) < 0)**

**{**

**return -1; // Negative offset**

**}**

**if((UFDTArr[fd].ptrfiletable->writeoffset + size) > UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)**

**{**

**// Extend the file size if seeking beyond current file size**

**UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize = UFDTArr[fd].ptrfiletable->writeoffset + size;**

**}**

**// Update the write offset**

**UFDTArr[fd].ptrfiletable->writeoffset += size;**

**}**

**// Seek relative to the start of the file**

**else if(from == START)**

**{**

**// Validate the new offset**

**if(size > MAXFILESIZE)**

**{**

**return -1; // Seeking beyond maximum file size**

**}**

**if(size < 0)**

**{**

**return -1; // Negative offset**

**}**

**if(size > UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize)**

**{**

**// Extend the file size if seeking beyond current file size**

**UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize = size;**

**}**

**// Set the write offset**

**UFDTArr[fd].ptrfiletable->writeoffset = size;**

**}**

**// Seek relative to the end of the file**

**else if(from == END)**

**{**

**// Validate the new offset**

**if((UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize) + size > MAXFILESIZE)**

**{**

**return -1; // Seeking beyond maximum file size**

**}**

**if((UFDTArr[fd].ptrfiletable->writeoffset + size) < 0)**

**{**

**return -1; // Negative offset**

**}**

**// Set the write offset relative to the end of the file**

**UFDTArr[fd].ptrfiletable->writeoffset = UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize + size;**

**}**

**}**

**}**

**Explanation:**

1. **Purpose**:
   * The LseekFile function changes the file offset (current position) in an open file. This is similar to the lseek function in Unix/Linux, allowing the file pointer to move forward, backward, or to a specific position.
2. **Validation**:
   * The function first checks if the file descriptor (fd) is valid and if the from argument is within acceptable values (START, CURRENT, END).
3. **Mode Check**:
   * The function adjusts the offset differently depending on whether the file is opened in read, write, or read+write mode.
4. **Seek Operations**:
   * **CURRENT**: Moves the offset relative to its current position. It ensures that the new offset doesn't go beyond the file's size or become negative.
   * **START**: Moves the offset relative to the beginning of the file. It checks that the size is within valid bounds.
   * **END**: Moves the offset relative to the end of the file. The function ensures that the offset does not exceed the maximum file size.
5. **File Size Adjustments**:
   * When the file is opened in write mode, the function may extend the file's size if the new write offset exceeds the current file size.

**Summary:**

* The **LseekFile** function effectively handles moving the file pointer for both reading and writing, with appropriate checks for valid input and file states. It ensures that file operations remain within legal bounds, preventing invalid memory accesses.

**Expected Output on the Command Prompt**

The output of the LseekFile function on the command prompt will vary based on the file descriptor, the offset (size), and the reference point (from). Here are some example scenarios:

**Example 1: Valid Offset Adjustment**

**Marvellous VFS : > create file1 3**

**File is successfully created with file descriptor: 3**

**Marvellous VFS : > lseek 3 10 0**

**File offset successfully adjusted**

**Marvellous VFS : > lseek 3 -5 1**

**File offset successfully adjusted**

* **Create File:** The file "file1" is created and given a descriptor of 3.
* **Lseek (Start):** The offset is set to 10 bytes from the start of the file. The operation succeeds, and the output indicates success.
* **Lseek (Current):** The offset is adjusted by -5 bytes from the current position, moving the pointer backward. Again, the operation is successful.

**Example 2: Invalid Offset Adjustment**

**Marvellous VFS : > lseek 3 10000 0**

**Error: Invalid offset**

**Marvellous VFS : > lseek 3 -100 2**

**Error: Invalid offset**

* **Lseek (Start):** Attempting to set the offset to 10000 bytes from the start of the file exceeds the file size or maximum file size. The operation fails, and an error message is shown.
* **Lseek (End):** Trying to move the offset -100 bytes from the end results in an invalid (negative) offset, causing the operation to fail.

**Description of Output**

* **Success Messages:**
  + "File offset successfully adjusted" confirms that the file pointer was moved as requested, whether it be forward, backward, or to a specific position in the file.
* **Error Messages:**
  + "Error: Invalid offset" indicates that the requested offset is out of bounds, either by going

**Explanation: ls\_file() function**

**void ls\_file()**

**{**

**int i = 0;**

**PINODE temp = head;**

**// Check if there are no files in the file system**

**if(SUPERBLOCKobj.FreeInode == MAXINODE)**

**{**

**printf("Error : There are no files\n");**

**return;**

**}**

**// Print the table header**

**printf("\nFile Name\tInode number\tFile size\tLink count\n");**

**printf("----------------------------------------------------------------\n");**

**// Traverse the inode list**

**while(temp != NULL)**

**{**

**// Print details only for active files (FileType != 0)**

**if(temp->FileType != 0)**

**{**

**printf("%s\t\t%d\t\t%d\t\t%d\n", temp->FileName, temp->InodeNumber, temp->FileActualSize, temp->LinkCount);**

**}**

**temp = temp->next;**

**}**

**// Print the footer**

**printf("----------------------------------------------------------------\n");**

**}**

**Key Points:**

1. **Checking for Files**:
   * The function first checks if the FreeInode count in the SUPERBLOCKobj equals MAXINODE. If it does, it means that no files have been created yet, and the function prints an error message indicating that there are no files.
2. **Table Header**:
   * Before listing the files, the function prints a header with columns for the file name, inode number, file size, and link count. This is followed by a separator line.
3. **Traversing the Inode List**:
   * The function iterates through the linked list of inodes starting from head.
   * For each inode, it checks if the file is active by verifying that FileType is not 0. If a file is found, its details are printed.
4. **Table Footer**:
   * After listing all files, the function prints a footer line to complete the table.

**Summary:**

* This function is useful for displaying a summary of all files currently stored in the file system, including important metadata like file size and inode information. It mimics the functionality of the ls command in Unix/Linux systems.

**Expected Output on the Command Prompt**

The output of the **ls\_file()** function on the command prompt will vary depending on whether there are files present in the file system or not. Here are two example scenarios:

**Example 1: No Files in the File System**

**Marvellous VFS : > ls**

**Error: There are no files**

* **Description:**
  + This output occurs when there are no files in the file system. The function checks the FreeInode count, determines that all inodes are free (indicating no files), and prints an error message to notify the user that no files exist.

**Example 2: Files Present in the File System**

**Marvellous VFS : > ls**

**File Name Inode number File size Link count**

**----------------------------------------------------------------**

**file1 1 100 1**

**file2 2 200 1**

**file3 3 150 1**

**----------------------------------------------------------------**

* **Description:**
  + This output is produced when there are files present in the file system. The function lists each file's name, inode number, file size, and link count in a neatly formatted table. The header and footer lines make the output more readable.
  + In this case:
    - "**file1**" has an inode number of **1**, a file size of **100 bytes**, and a link count of **1**.
    - **"file2"** has an inode number of **2**, a file size of **200 bytes,** and a link count of **1**.
    - **"file3"** has an inode number of **3,** a file size of **150 bytes,** and a link count of **1**.

**Description of Output**

* **Error Message:**
  + "Error : There are no files" indicates that the file system is empty, with no files created or stored. This message prevents the function from attempting to list files when none exist, saving processing time and avoiding confusion.
* **File Listing:**
  + The table format used for listing files provides a structured and clear overview of each file's essential details, such as its name, inode number, size, and link count. This output is particularly useful for users who need to see a summary of the file system's contents quickly. The headers and footers help differentiate the list from other command outputs, making it easier to understand.

**Explanation: fstat\_file() function**

**int fstat\_file(int fd)**

**{**

**PINODE temp = head;**

**int i = 0;**

**// Check if the provided file descriptor is valid**

**if(fd < 0)**

**{**

**return -1;**

**}**

**// Check if the file table entry is NULL, meaning the file is not open**

**if(UFDTArr[fd].ptrfiletable == NULL)**

**{**

**return -2;**

**}**

**// Retrieve the inode associated with the file descriptor**

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

**// Print the file's statistical information**

**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);**

**// Determine and print the file's permissions**

**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;**

**}**

**Key Points:**

1. **File Descriptor Validation**:
   * The function first checks whether the provided file descriptor (fd) is valid (i.e., not less than 0). If it is invalid, the function returns -1.
2. **File Table Check**:
   * Next, it checks if the ptrfiletable associated with the given fd is NULL. If the file table entry is NULL, it indicates that the file is not currently open, so the function returns -2.
3. **Fetching Inode Information**:
   * The function retrieves the inode (PINODE temp) corresponding to the file descriptor from the UFDTArr. This inode contains all the necessary file metadata.
4. **Displaying File Information**:
   * The function then prints out various details about the file:
     + **File Name**: The name of the file.
     + **Inode Number**: The unique inode number.
     + **File Size**: The maximum size allocated for the file.
     + **Actual File Size**: The actual amount of data currently in the file.
     + **Link Count**: The number of links (or references) to the file.
     + **Reference Count**: The number of active references to this inode.
     + **File Permission**: The permissions associated with the file, such as read-only, write-only, or read-write.
5. **Return Value**:
   * After displaying all the information, the function returns 0, indicating success.

**Summary:**

* The **fstat\_file** function is akin to the fstat system call in Unix/Linux, providing crucial information about a file based on its file descriptor. This function is particularly useful for debugging or for gaining insights into file properties without altering the file's state.

**Expected Output on the Command Prompt**

The output of the **fstat\_file()** function on the command prompt will vary depending on whether the file descriptor is valid, if the file is open, and the file's metadata.

**Example 1: Invalid File Descriptor**

**Marvellous VFS : > fstat 5**

**Error : Invalid file descriptor**

* **Description:**
  + The file descriptor provided is invalid (e.g., 5, when only 0-3 might be valid). The function returns -1, and the program prints an error message indicating the file descriptor is invalid.

**Example 2: File Descriptor Not Open**

**Marvellous VFS : > fstat 2**

**Error : File not open**

* **Description:**
  + The file descriptor provided (e.g., 2) is valid, but the file associated with it is not currently open. The function returns -2, and the program prints an error message indicating the file is not open.

**Example 3: Valid File Descriptor with Open File**

**Marvellous VFS : > fstat 1**

**----------Statistical Information about file-----------**

**File name : file1**

**Inode Number : 1**

**File size : 1024**

**Actual File size : 100**

**Link count : 1**

**Reference count : 2**

**File Permission : Read only**

**--------------------------------------------------------**

* **Description:**
  + The file descriptor provided (e.g., 1) is valid, and the file associated with it is currently open. The function retrieves the file's inode and prints detailed information about it:
    - **File Name:** "file1"
    - **Inode Number:** 1
    - **File Size:** 1024 bytes (maximum allocated size).
    - **Actual File Size:** 100 bytes (actual data stored).
    - **Link Count:** 1
    - **Reference Count:** 2 (file is opened twice).
    - **File Permission:** Read only (file can only be read).

This output provides a comprehensive overview of the file's metadata, helping users and developers understand the file's current state.

**Explanation: stat\_file() function**

**int stat\_file(char \* name)**

**{**

**PINODE temp = head;**

**int i = 0;**

**// Check if the provided file name is NULL**

**if(name == NULL) return -1;**

**// Traverse the inode list to find the file with the specified name**

**while(temp != NULL)**

**{**

**if(strcmp(name, temp->FileName) == 0)**

**break;**

**temp = temp->next;**

**}**

**// If the file is not found, return -2**

**if(temp == NULL) return -2;**

**// Print the statistical information of the found file**

**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);**

**// Display the file permissions**

**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");**

**return 0;**

**}**

**Key Points:**

1. **File Name Validation**:
   * The function begins by checking if the provided file name is NULL. If it is, the function returns -1 to indicate an invalid input.
2. **Traversing the Inode List**:
   * The function traverses the linked list of inodes, starting from the head, to find an inode with a matching file name using the strcmp function.
   * If a match is found, the loop breaks, and the function proceeds to the next steps.
3. **File Existence Check**:
   * If the loop completes without finding the file (i.e., temp is NULL), the function returns -2, indicating that the file does not exist.
4. **Displaying File Information**:
   * If the file is found, the function prints various details about the file:
     + **File Name**: The name of the file.
     + **Inode Number**: The unique identifier for the inode.
     + **File Size**: The maximum size allocated for the file.
     + **Actual File Size**: The amount of data currently stored in the file.
     + **Link Count**: The number of references (links) to this file.
     + **Reference Count**: The number of active references to this inode.
     + **File Permission**: The file's permission settings, such as read-only, write-only, or read-write.
5. **Return Value**:
   * After printing the file's statistical information, the function returns 0, indicating successful execution.

**Summary:**

* The **stat\_file** function provides a way to retrieve and display detailed information about a file using its name. It's useful for understanding the state of a file without needing a file descriptor. If the file is not found or an invalid name is provided, the function returns appropriate error codes.

**Expected Output on the Command Prompt**

The output of the **stat\_file()** function on the command prompt will vary depending on whether the file name is valid, if the file exists, and the file's metadata.

**Example 1: Invalid File Name (NULL)**

**Marvellous VFS : > stat**

**Error : Invalid file name**

* **Description:**
  + If the file name provided is NULL, the function returns -1, and the program prints an error message indicating that the file name is invalid.

**Example 2: File Not Found**

**Marvellous VFS : > stat non\_existent\_file**

**Error : File not found**

* **Description:**
  + If the file name provided (e.g., "non\_existent\_file") does not exist in the file system, the function returns -2, and the program prints an error message indicating that the file was not found.

**Example 3: Valid File Name**

**Marvellous VFS : > stat file1**

**----------Statistical Information about file------------**

**File name : file1**

**Inode Number : 1**

**File size : 1024**

**Actual File size : 100**

**Link count : 1**

**Reference count : 2**

**File Permission : Read only**

**-----------------------------------------------------------**

* **Description:**
  + If the file name provided (e.g., "file1") exists, the function retrieves the file's inode and prints detailed information about it:
    - **File Name:** "file1"
    - **Inode Number:** 1
    - **File Size:** 1024 bytes (maximum allocated size).
    - **Actual File Size:** 100 bytes (actual data stored).
    - **Link Count:** 1
    - **Reference Count:** 2 (file is opened twice).
    - **File Permission:** Read only (file can only be read).

This output gives a comprehensive overview of the file's metadata, providing valuable insights into the file's current state.

**Explanation: truncate\_File function**

**// function clears the contents of a file and resets its size to zero while retaining the file itself for future operations.**

**int truncate\_File(char \*name)**

**{**

**// Get the file descriptor associated with the given file name**

**int fd = GetFDFromName(name);**

**// If the file doesn't exist, return an error code (-1)**

**if(fd == -1)**

**return -1;**

**// Clear the file's buffer, effectively erasing its contents**

**// memset is used to fill the buffer with zeros (0) over 1024 bytes**

**memset(UFDTArr[fd].ptrfiletable->ptrinode->Buffer, 0, 1024);**

**// Reset the file's read offset to 0**

**// This means any future read operations will start from the beginning of the file**

**UFDTArr[fd].ptrfiletable->readoffset = 0;**

**// Reset the file's write offset to 0**

**// This means any future write operations will start from the beginning of the file**

**UFDTArr[fd].ptrfiletable->writeoffset = 0;**

**// Update the file's actual size to 0**

**// Since the file is now empty, its actual size is set to 0**

**UFDTArr[fd].ptrfiletable->ptrinode->FileActualSize = 0;**

**// Return 0 to indicate successful truncation of the file**

**return 0;**

**}**

**Detailed Explanation:**

1. **Getting the File Descriptor**:
   * The function begins by retrieving the file descriptor (fd) associated with the file name passed as an argument using the GetFDFromName function.
   * If the file does not exist, GetFDFromName will return -1, and the function will return -1 to indicate an error.
2. **Clearing the File's Buffer**:
   * memset is used to fill the file's buffer with zeros (0) across the entire buffer size (1024 bytes). This effectively clears all data stored in the file, leaving it empty.
3. **Resetting the Offsets**:
   * The read offset (readoffset) and write offset (writeoffset) are both reset to 0. This means that any subsequent read or write operations will start at the beginning of the file.
4. **Updating the File Size**:
   * The actual size of the file is set to 0, reflecting that the file is now empty after truncation.
5. **Returning Success**:
   * The function returns 0 to indicate that the truncation was successful.

This version of the function is intended to completely clear the contents of the specified file while retaining the file itself, ready for new data to be written or read from it.

**Expected Output on the Command Prompt**

The output of the **truncate\_File()** function on the command prompt will vary depending on whether the file exists or not.

**Example 1: File Does Not Exist**

**Marvellous VFS : > truncate non\_existent\_file**

**Error : File not found**

* **Description:**
  + If the file name provided (e.g., "non\_existent\_file") does not exist in the file system, the function returns -1, and the program prints an error message indicating that the file was not found.

**Example 2: File Exists**

**Marvellous VFS : > truncate file1**

**File truncated successfully**

* **Description:**
  + If the file name provided (e.g., "file1") exists in the file system, the function successfully clears its contents, resets its size to zero, and returns 0. The program prints a confirmation message indicating that the file was truncated successfully.

This output shows whether the truncation operation was successful and informs the user of the status of the specified file.

**Explanation: main function**

**count =** sscanf**(str,** "%s %s %s %s"**, command[0], command[1], command[2], command[3]);**

The line ‘count = sscanf(str, "%s %s %s %s", command[0], command[1], command[2], command[3]);’ is used to parse the input string str into up to four separate strings (or commands) and store them in the ‘command’ array. Here's a detailed explanation of how it works:

**Explanation:**

**Function sscanf**:

* **sscanf** stands for **"string scan formatted"** and is used to **read formatted** input from a string, similar to how scanf reads formatted input from standard input.
* The function prototype is: ‘**int sscanf(const char \*str, const char \*format, ...);’**
* **‘str’:** The input string to be parsed.
* **‘format’:** A format string that specifies how to interpret the input data.
* **‘...’:** Pointers to variables where the parsed values will be stored.

**Format String "%s %s %s %s"**:

* The format string "%s %s %s %s" tells ‘sscanf’ to **read up to four whitespace-separated** strings.
* Each ‘%s’ matches a sequence of **non-whitespace characters**. Whitespace (spaces, tabs, newlines) is used as the delimiter between the strings.

**Command Array command[4][80]**:

* **‘command’** is a 2D array of characters, where **‘command[0]’** to **‘command[3]’** are each arrays capable of holding up to 79 characters plus a null terminator.
* The function call **‘sscanf(str, "%s %s %s %s", command[0], command[1], command[2], command[3])’** attempts to store the first four whitespace-separated strings from **‘str’** into **‘command[0]’**, **‘command[1]’**, **‘command[2]’**, and **‘command[3]’**, respectively.

**Return Value count**:

* **‘sscanf’** returns the number of input items successfully matched and assigned. This value is stored in **‘count’**.
* If **‘str’** contains fewer than four whitespace-separated strings, **‘count’** will reflect the actual number of strings successfully parsed and stored.

**Example:**

Consider the following example where **‘str’** contains the string "**create file test.txt**":

**char** str[] = "create file test.txt";

**char** command[4][80];

**int** count;

**count** = sscanf(str, "%s %s %s %s", command[0], command[1], command[2], command[3]);

After executing this line:

**command[0]** will contain **"create"**.

**command[1]** will contain **"file"**.

**command[2]** will contain **"test.txt"**.

**command[3]** will remain an empty string because there is no fourth string in the input.

**count** will be **3**, indicating that three strings were successfully parsed and stored.

**Structure Definitions:**

* The **‘SUPERBOLCK’** structure is used to keep track of the total number of inodes and free inodes.
* The **‘INODE’** structure represents an inode with various file properties including file name, inode number, file size, file type, buffer for file content, link count, reference count, and permissions.

**int main()**

**{**

**char \*ptr = NULL; // Pointer for dynamic memory allocation**

**int ret = 0, fd = 0, count = 0; // Return value, file descriptor, and command word count**

**char command[4][80], str[80], arr[1024]; // Arrays to store commands and user input**

**InitialiseSuperBlock(); // Initialize the superblock of the file system**

**CreateDILB(); // Create and initialize the Directory Inode List Block**

**// Main loop for the shell**

**while(1)**

**{**

**fflush(stdin); // Clear input buffer**

**strcpy(str,""); // Initialize input string**

**printf("\nMarvellous VFS : > "); // Prompt for user input**

**fgets(str,80,stdin); // Read input from the user**

**count = sscanf(str,"%s %s %s %s",command[0],command[1],command[2],command[3]); // Parse command and arguments**

**// Handling commands with a single argument**

**if(count == 1)**

**{**

**if(strcmp(command[0],"ls") == 0) // List files**

**{**

**ls\_file();**

**}**

**else if(strcmp(command[0],"closeall") == 0) // Close all open files**

**{**

**CloseAllFile();**

**printf("All files closed successfully\n");**

**continue;**

**}**

**else if(strcmp(command[0],"cls") == 0) // Clear screen**

**{**

**system("cls");**

**continue;**

**}**

**else if(strcmp(command[0],"help") == 0) // Display help information**

**{**

**DisplayHelp();**

**continue;**

**}**

**else if(strcmp(command[0],"exit") == 0) // Exit the shell**

**{**

**printf("Terminating the Marvellous Virtual File System\n");**

**// Deallocate all data structures**

**break;**

**}**

**else**

**{**

**printf("\nERROR : Command not found!!!\n"); // Invalid command**

**continue;**

**}**

**}**

**// Handling commands with two arguments**

**else if(count == 2)**

**{**

**if(strcmp(command[0],"stat") == 0) // Show file status by name**

**{**

**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) // Show file status by file descriptor**

**{**

**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) // Close file by name**

**{**

**ret = CloseFileByName(command[1]);**

**if(ret == -1)**

**printf("ERROR : There is no such file\n");**

**continue;**

**}**

**else if(strcmp(command[0],"rm") == 0) // Remove file by name**

**{**

**ret = rm\_File(command[1]);**

**if(ret == -1)**

**printf("ERROR : There is no such file\n");**

**continue;**

**}**

**else if(strcmp(command[0],"man") == 0) // Display manual page for a command**

**{**

**man(command[1]);**

**}**

**else if(strcmp(command[0],"write") == 0) // Write data to file**

**{**

**fd = GetFDFromName(command[1]);**

**if(fd == -1)**

**{**

**printf("ERROR : Incorrect parameter\n");**

**continue;**

**}**

**printf("Enter the data : \n");**

**scanf("%[^\n]s",arr); // Read data to write**

**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) // Truncate file to zero size**

**{**

**ret = truncate\_File(command[1]);**

**if(ret == -1)**

**printf("ERROR : Incorrect parameter\n");**

**}**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**}**

**// Handling commands with three arguments**

**else if(count == 3)**

**{**

**if(strcmp(command[0],"create") == 0) // Create a new file**

**{**

**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 are 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) // Open an existing file**

**{**

**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) // Read data from file**

**{**

**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); // Output data to standard error (file descriptor 2)**

**}**

**continue;**

**}**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**}**

**// Handling commands with four arguments**

**else if(count == 4)**

**{**

**if(strcmp(command[0],"lseek") == 0) // Change file offset**

**{**

**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"); // Invalid command**

**continue;**

**}**

**}**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**}**

**return 0;**

**}**

**Initial Setup**

1. **Variable Declarations:**

**char \*ptr = NULL; // Pointer for dynamic memory allocation**

**int ret = 0, fd = 0, count = 0; // Return value, file descriptor, and command word count**

**char command[4][80], str[80], arr[1024]; // Arrays to store commands and user input**

**ptr:** Pointer for dynamically allocated memory, used for reading file content.

**ret:** Variable to store return values from various functions.

**fd:** File descriptor.

**count:** Number of command parts parsed.

**command:** Array to hold up to four command arguments (with max length of 80 each).

**str:** String to hold the full input command.

**arr:** Buffer for data to be written to a file.

1. **Initialization:**

**InitialiseSuperBlock(); // Initialize the superblock of the file system**

**CreateDILB(); // Create and initialize the Directory Inode List Block**

* + **InitialiseSuperBlock():** Sets up the superblock which contains information about the file system's state.
  + **CreateDILB():** Initializes the Directory Inode List Block, which manages file metadata.

**Command Loop**

1. **Main Shell Loop:**

**while(1)**

**{**

**fflush(stdin); // Clear input buffer**

**strcpy(str,""); // Initialize input string**

* + This is an infinite loop that continuously prompts the user for commands and processes them.

1. **Prompt and Read Input:**

**printf("\nMarvellous VFS : > "); // Prompt for user input**

**fgets(str,80,stdin); // Read input from the user**

* + Displays a prompt to the user and reads their input into the str buffer.

1. **Parse Command:**

**count = sscanf(str,"%s %s %s %s",command[0],command[1],command[2],command[3]); // Parse command and arguments**

* + sscanf splits the input string into up to four components and stores them in the command array.

**Command Handling**

1. **Single Argument Commands:**

**if(count == 1)**

**{**

**if(strcmp(command[0],"ls") == 0) // List files**

**{**

**ls\_file();**

**}**

**else if(strcmp(command[0],"closeall") == 0) // Close all open files**

**{**

**CloseAllFile();**

**printf("All files closed successfully\n");**

**continue;**

**}**

**else if(strcmp(command[0],"cls") == 0) // Clear screen**

**{**

**system("cls");**

**continue;**

**}**

**else if(strcmp(command[0],"help") == 0) // Display help information**

**{**

**DisplayHelp();**

**continue;**

**}**

**else if(strcmp(command[0],"exit") == 0) // Exit the shell**

**{**

**printf("Terminating the Marvellous Virtual File System\n");**

**// Deallocate all data structures**

**break;**

**}**

**else**

**{**

**printf("\nERROR : Command not found!!!\n"); // Invalid command**

**continue;**

**}**

**}**

**Handles commands that require a single argument:**

**ls:** Lists all files using **ls\_file()**.

**closeall:** Closes all open files using **CloseAllFile()**.

**cls:** Clears the terminal screen using the system**("cls")** command.

**help:** Displays help information using **DisplayHelp()**.

**exit:** Exits the shell and deallocates resources.

1. **Two Argument Commands:**

**else if(count == 2)**

**{**

**if(strcmp(command[0],"stat") == 0) // Show file status by name**

**{**

**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) // Show file status by file descriptor**

**{**

**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) // Close file by name**

**{**

**ret = CloseFileByName(command[1]);**

**if(ret == -1)**

**printf("ERROR : There is no such file\n");**

**continue;**

**}**

**else if(strcmp(command[0],"rm") == 0) // Remove file by name**

**{**

**ret = rm\_File(command[1]);**

**if(ret == -1)**

**printf("ERROR : There is no such file\n");**

**continue;**

**}**

**else if(strcmp(command[0],"man") == 0) // Display manual page for a command**

**{**

**man(command[1]);**

**}**

**else if(strcmp(command[0],"write") == 0) // Write data to file**

**{**

**fd = GetFDFromName(command[1]);**

**if(fd == -1)**

**{**

**printf("ERROR : Incorrect parameter\n");**

**continue;**

**}**

**printf("Enter the data : \n");**

**scanf("%[^\n]s",arr); // Read data to write**

**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) // Truncate file to zero size**

**{**

**ret = truncate\_File(command[1]);**

**if(ret == -1)**

**printf("ERROR : Incorrect parameter\n");**

**}**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**}**

**Handles commands that require two arguments:**

**stat:** Shows file status using **stat\_file()**.

**fstat:** Shows file status by file descriptor using **fstat\_file()**.

**close:** Closes a file by name using **CloseFileByName()**.

**rm:** Removes a file by name using **rm\_File()**.

**man:** Displays a manual page using **man()**.

**write:** Writes data to a file using **WriteFile()**.

**truncate:** Truncates a file to zero size using **truncate\_File()**.

1. **Three Argument Commands:**

**else if(count == 3)**

**{**

**if(strcmp(command[0],"create") == 0) // Create a new file**

**{**

**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 are 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) // Open an existing file**

**{**

**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) // Read data from file**

**{**

**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); // Output data to standard error (file descriptor 2)**

**}**

**continue;**

**}**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**}**

**Handles commands that require three arguments:**

**create:** Creates a new file with specified name and size using **CreateFile()**.

**open:** Opens an existing file with specified name and mode using **OpenFile()**.

**read:** Reads data from a file using **ReadFile()** and outputs it.

1. **Four Argument Commands:**

**else if(count == 4)**

**{**

**if(strcmp(command[0],"lseek") == 0) // Change file offset**

**{**

**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"); // Invalid command**

**continue;**

**}**

**}**

**Handles commands that require four arguments:**

**lseek:** Changes the file offset using LseekFile().

1. **Default Case:**

**else**

**{**

**printf("\nERROR : Command not found !!!\n"); // Invalid command**

**continue;**

**}**

**Handles cases where the number of arguments doesn't match expected values (1, 2, 3, or 4).**

**Exit Program**

1. **Program Termination:**

**return 0;**

* + Exits the program with a success status.

This detailed breakdown covers each section of the main function, explaining how it processes various commands and handles errors.

**What is mean by File System?**

A file system is a method and data structure that an operating system uses to manage files on a storage device. It organizes and controls how data is stored, accessed, and managed on the device. Here’s a breakdown of its key aspects:

**Key Concepts of a File System**

1. **Files:**
   * Basic units of storage, containing data or programs. Each file has a name and possibly an extension, which helps identify its type and purpose.
2. **Directories:**
   * Folders that contain files and other directories. They help organize files in a hierarchical structure.
3. **Inodes:**
   * Data structures that store metadata about files, such as their size, permissions, and location on the disk. Inodes do not store file names but point to the data blocks where the file content is stored.
4. **Data Blocks:**
   * The actual storage units on the disk where the file’s data is kept. Files are divided into these blocks for storage.
5. **File Descriptors:**
   * Integer values used by the operating system to keep track of open files. When a file is opened, the operating system provides a file descriptor to refer to that open file.
6. **File Operations:**
   * Common operations include creating, opening, reading, writing, and deleting files. The file system provides APIs to perform these operations.
7. **Permissions:**
   * Rules that define who can access or modify a file. Permissions are usually set for the file owner, group, and others (read, write, execute).

**File system is simulated using structures and functions to manage files and directories in a virtual** **environment. Here’s how the code relates to the file system:**

1. **Super Block:**
   * Keeps track of the overall file system state, including the total number of inodes and the number of free inodes.
2. **Inodes and File Table:**
   * Each file is represented by an inode, and the file table keeps track of open files and their associated file descriptors.
3. **File Operations:**
   * The code provides functions to create, open, read, write, and delete files. It simulates these operations using inodes and file descriptors.
4. **Commands:**
   * Commands like create, open, read, write, close, and rm interact with the simulated file system to perform various tasks.
5. **Data Management:**
   * Functions handle reading and writing data to the simulated storage, managing file data blocks, and updating file metadata.

In essence, a file system abstracts the complexity of storing and retrieving files from physical storage, allowing users and applications to interact with files in a straightforward manner. The provided code mimics this functionality in a controlled environment for educational or testing purposes.

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

Different operating systems use different file systems to manage data storage. Here’s an overview of the primary file systems used by Linux and Windows operating systems:

**Linux File Systems**

1. **Ext4 (Fourth Extended File System)**:
   * **Description**: The most commonly used file system on Linux. It offers improvements over its predecessors (Ext3) in terms of performance, capacity, and reliability.
   * **Features**: Journaling, large file support, improved file system checks.
2. **XFS**:
   * **Description**: A high-performance file system developed by Silicon Graphics. It is known for handling large files and large volumes efficiently.
   * **Features**: Journaling, scalability, support for very large files and file systems.
3. **Btrfs (B-tree File System)**:
   * **Description**: A modern file system designed to address some of the shortcomings of Ext4 and XFS. It aims to provide advanced features.
   * **Features**: Snapshots, volume management, checksums for data integrity, self-healing.
4. **ReiserFS**:
   * **Description**: Known for its efficient handling of small files and its performance characteristics. Its development has slowed down, and it is less common in newer distributions.
   * **Features**: Journaling, efficient small-file storage.
5. **F2FS (Flash-Friendly File System)**:
   * **Description**: Designed specifically for flash storage devices such as SSDs and eMMC.
   * **Features**: Optimized for flash memory, wear leveling.
6. **ZFS**:
   * **Description**: Originally developed by Sun Microsystems, it is known for its high scalability and data integrity features. It is available on Linux through third-party modules.
   * **Features**: Snapshots, data integrity checks, high storage capacity, built-in RAID functionality.

**Windows File Systems**

1. **NTFS (New Technology File System)**:
   * **Description**: The primary file system used by Windows operating systems. It offers advanced features and supports a wide range of file system operations.
   * **Features**: Journaling, large file and volume support, file permissions, encryption, compression.
2. **FAT32 (File Allocation Table 32)**:
   * **Description**: An older file system used in various Windows versions, primarily for compatibility with older systems and removable storage devices.
   * **Features**: Simplicity, compatibility with various operating systems, limited file size and volume size (4 GB file size limit, 8 TB volume size limit).
3. **exFAT (Extended File Allocation Table)**:
   * **Description**: Designed to address the limitations of FAT32 and is used for flash drives and SD cards.
   * **Features**: Larger file and volume size support than FAT32, optimized for flash memory.
4. **ReFS (Resilient File System)**:
   * **Description**: Introduced in Windows Server 2012, it is designed for high data integrity and large-scale storage.
   * **Features**: Data integrity checks, fault tolerance, large volume support, optimized for virtualization.

Each file system has its strengths and weaknesses, making them more or less suitable depending on the specific requirements and use cases.

**What are the parts of the file system?**

A file system is structured into several key components or parts, each playing a specific role in managing and organizing data on storage devices. Here’s an overview of the major parts of a typical file system:

**1. File Allocation Table (FAT) / File Indexes**

* **Purpose**: Keeps track of where files are stored on the disk.
* **Details**: In FAT file systems, the File Allocation Table is a data structure that maps out the clusters used by files. In more advanced file systems, such as NTFS or Ext4, this role is handled by complex indexing structures or databases.

**2. Directory Structure**

* **Purpose**: Organizes files into directories (folders) and subdirectories.
* **Details**: Directories contain information about files and other directories. The structure typically forms a hierarchical tree, allowing for nested directories.

**3. Metadata**

* **Purpose**: Stores information about files and directories.
* **Details**: Metadata includes file attributes like name, size, creation date, modification date, permissions, and ownership. Each file and directory has associated metadata that helps the file system manage and access the data.

**4. Superblock**

* **Purpose**: Contains information about the file system itself.
* **Details**: The superblock holds essential data such as the file system’s size, the block size, the number of inodes, and pointers to other critical structures. It is crucial for file system operations and recovery.

**5. Inodes**

* **Purpose**: Store metadata about files.
* **Details**: Each file has an inode that includes metadata such as file size, permissions, ownership, and pointers to the data blocks where the file’s content is stored. Inodes do not contain the file name or actual data.

**6. Data Blocks**

* **Purpose**: Store the actual content of files.
* **Details**: The data blocks are where the file system writes the file’s data. These blocks are managed by the file allocation table or indexing structures, and their locations are referenced by inodes or similar structures.

**7. Journaling (in Journaling File Systems)**

* **Purpose**: Helps ensure file system integrity.
* **Details**: Journaling file systems keep a log (journal) of changes that are about to be made. This helps in recovering and maintaining consistency in the event of a system crash or power failure.

**8. Free Space Management**

* **Purpose**: Manages available space on the storage device.
* **Details**: Keeps track of which blocks are free and available for use. This can be managed through various methods such as bitmap tables or linked lists.

**9. Volume and Partition Information**

* **Purpose**: Organizes and manages the physical storage.
* **Details**: Partitions divide the physical storage device into logical units, and volumes are the file system’s view of these partitions. This organization helps in managing different file systems on a single device.

**10. Access Control Lists (ACLs) and Permissions**

* **Purpose**: Manage file and directory access.
* **Details**: Defines who can read, write, or execute a file or directory. ACLs provide more granular control over permissions compared to traditional Unix-style permissions.

Each part of the file system plays a critical role in ensuring that data is stored efficiently, accessed correctly, and managed effectively.

**Explain UAREA and its contents?**

The **User Area (UAREA)**, often referred to as UAREA, is a crucial data structure in the context of operating systems, particularly in older Unix-like systems. It is used to manage information about user processes. Here’s a detailed breakdown of UAREA and its contents:

**Overview of UAREA**

* **Purpose**: The UAREA is a kernel data structure that holds information about the state of a user process. It provides a bridge between user-level processes and kernel-level operations, containing both control and accounting information.

**Contents of UAREA**

1. **Process Control Information**
   * **Process State**: Information about the current state of the process (e.g., running, waiting, stopped).
   * **Process ID (PID)**: A unique identifier assigned to the process.
   * **Parent Process ID (PPID)**: The PID of the parent process that created this process.
   * **User ID (UID) and Group ID (GID)**: Identifiers for the user and group under which the process is running, used for permission and ownership checks.
2. **Memory Management Information**
   * **Base and Limit Addresses**: Addresses indicating the range of memory allocated for the process.
   * **Page Tables**: Structures used for virtual to physical address translation.
   * **Memory Allocation Details**: Information about memory allocated to the process, including the stack, heap, and data segments.
3. **File Descriptor Information**
   * **File Descriptors**: A list of open files associated with the process, including pointers to file tables.
   * **File Offset**: Current position in the open file.
4. **Scheduling Information**
   * **Priority**: The scheduling priority of the process.
   * **Time Slices**: Information on how much CPU time the process has used and is allowed to use.
5. **Accounting Information**
   * **CPU Time**: The amount of CPU time used by the process.
   * **Resource Usage**: Statistics on resources used by the process, such as memory and I/O operations.
6. **Signal Information**
   * **Pending Signals**: Information about signals that have been sent to the process but are not yet handled.
   * **Signal Handlers**: Pointers to functions that handle specific signals.
7. **Process Environment**
   * **Command Line Arguments**: Information about how the process was invoked, including command-line arguments.
   * **Environment Variables**: Variables that influence the behavior of the process.
8. **Stack and Registers**
   * **Stack Pointers**: Addresses related to the process’s stack, including the top of the stack.
   * **CPU Registers**: Values in the CPU registers when the process is interrupted or switched.

**Importance of UAREA**

* **Context Switching**: During context switches, the kernel saves and restores the UAREA to manage process states effectively.
* **Process Management**: UAREA provides the necessary data for process management and scheduling decisions.
* **Debugging and Monitoring**: It is essential for debugging and monitoring tools to access information about processes.

**Modern Relevance**

In modern operating systems, the UAREA concept has evolved or been replaced by more sophisticated data structures and mechanisms. For instance, in Unix-like systems, similar information is now managed by process control blocks (PCBs) and various kernel data structures. However, the fundamental principles of maintaining process-related information remain similar.

In the context of your Virtual File System (VFS) project, the term **UAREA** might be used differently from traditional operating systems. Since your project involves file system management, it's possible that UAREA refers to a data structure or area used to manage information related to file system operations or process management within your virtual environment.

**Possible Contents of UAREA in a VFS Project**

1. **File Descriptor Management**
   * **File Handles**: Information about open files in the virtual file system, similar to traditional file descriptors.
   * **File Pointers**: Current position in each open file.
2. **Process Information**
   * **Process State**: Status of processes interacting with the VFS (e.g., reading, writing).
   * **Process ID and Control Information**: Information specific to processes accessing the VFS.
3. **File System Metadata**
   * **File Attributes**: Metadata for files in the VFS, such as permissions, ownership, and timestamps.
   * **Directory Information**: Details about directories and their structure within the VFS.
4. **Resource Management**
   * **Memory Allocation**: Information on memory used by the VFS for file operations and management.
   * **Caching Information**: Details about cached data to improve performance.
5. **Error Handling and Logging**
   * **Error Codes**: Information about errors encountered during file operations.
   * **Logs**: Record of actions and events related to file system operations.
6. **File System State**
   * **Current Operations**: Information about ongoing file operations.
   * **Locks and Synchronization**: Details about locks and synchronization mechanisms used to manage concurrent access.

**Implementation in Your Project**

In your VFS project, the UAREA could be implemented as a data structure or set of structures that manage all the necessary information for file operations and process interactions within the virtual file system. Here’s a simplified example of what this might look like in code:

c

Copy code

// Define a structure for managing file descriptors in the UAREA

typedef struct {

int fileDescriptor; // Unique identifier for the open file

char \*fileName; // Name of the file

int fileOffset; // Current position in the file

// Add more fields as necessary

} FileDescriptorEntry;

// Define a structure for UAREA

typedef struct {

FileDescriptorEntry fileDescriptors[MAX\_OPEN\_FILES]; // Array of open files

int processId; // ID of the process using the VFS

int processState; // State of the process (e.g., running, waiting)

// Add more fields as necessary

} UAREA;

**Key Points**

* **Customization**: The exact contents and structure of the UAREA in your project will depend on the specific requirements of your virtual file system and the operations it needs to support.
* **Integration**: Ensure that the UAREA integrates well with other components of your VFS, such as file management, process control, and error handling.

If you have specific details or requirements for your VFS project, I can provide more tailored guidance on how to implement and use the UAREA effectively.

**Explain the use of file table and its content.**

Based on your CVFS (Custom Virtual File System) project, the **file table** would be a central component in managing and tracking open files. Here's a detailed explanation of its use and content in the context of your CVFS project:

**Purpose of the File Table in CVFS**

1. **Track Open Files**: The file table maintains information about all files currently open within the virtual file system. This allows the CVFS to manage multiple file operations concurrently.
2. **Manage File Descriptors**: Each open file is assigned a unique file descriptor. The file table maps these descriptors to specific file entries, enabling processes to interact with the files using these descriptors.
3. **Maintain File State**: The file table keeps track of the state of each open file, including its current position, access mode, and other attributes required for file operations.
4. **Handle File Operations**: By providing a centralized structure for open file management, the file table helps facilitate operations such as reading, writing, and closing files.

**Contents of the File Table in CVFS**

The file table in your CVFS project would typically contain the following elements:

1. **File Descriptor**: A unique integer assigned to each open file. This descriptor is used by processes to perform operations on the file.
2. **File Pointer**: A pointer to the file’s data or control block. This might be a reference to a data structure that holds information about the file’s content or metadata.
3. **File Offset**: The current position within the file for read and write operations. This indicates where the next operation will occur within the file.
4. **Access Mode**: Information about how the file is accessed (e.g., read-only, write-only, read-write). This ensures that operations adhere to the file’s access permissions.
5. **Reference Count**: The number of processes or operations currently using the file. This helps manage when to close the file and release associated resources.
6. **File Status Flags**: Various flags that indicate the file's status or attributes, such as whether the file is locked, whether it has pending writes, etc.
7. **Pointer to Inode or Metadata**: In a system using inodes, the file table entry might include a reference to the file’s inode, which contains metadata about the file on the disk.

In file systems, a **file table** is a crucial data structure used to manage and keep track of open files. It provides a centralized way to store information about files currently being accessed by the system.

Here’s an explanation of the **file table** and its contents based on the provided code:

**File Table Definition:**

The FILETABLE structure in the code is defined as follows:

c

Copy code

typedef struct filetable

{

PINODE ptrinode; // Pointer to the inode structure of the file

int readoffset; // Offset for read operations

int writeoffset; // Offset for write operations

int count; // Number of references (file descriptors) to this table entry

int mode; // Mode in which the file is opened (e.g., read, write)

} FILETABLE, \*PFILETABLE;

**Key Fields in the FILETABLE Structure:**

1. **ptrinode**:
   * **Description**: A pointer to the INODE structure of the file.
   * **Purpose**: This field links the file table entry to the corresponding inode, allowing the system to access the file's metadata and data.
2. **readoffset**:
   * **Description**: The current position in the file for read operations.
   * **Purpose**: Keeps track of where the next read operation will start. This is important for sequential reading of file data.
3. **writeoffset**:
   * **Description**: The current position in the file for write operations.
   * **Purpose**: Keeps track of where the next write operation will append or overwrite data. This is crucial for managing data integrity during write operations.
4. **count**:
   * **Description**: The number of references (file descriptors) pointing to this file table entry.
   * **Purpose**: Manages how many processes or threads have this file open. It helps in tracking open files and handling file closing operations correctly.
5. **mode**:
   * **Description**: The mode in which the file is opened.
   * **Purpose**: Specifies whether the file is opened for reading, writing, or both. This field ensures that the appropriate operations are permitted on the file.

**Role in File Operations:**

* **File Opening**: When a file is opened, a new entry in the file table is created or updated. The ptrinode field is set to the corresponding inode, and the readoffset and writeoffset are initialized based on the file mode and existing file data.
* **File Reading/Writing**: The file table entry helps manage read and write operations by using the readoffset and writeoffset fields. The offsets are updated as data is read or written, ensuring that operations occur at the correct positions.
* **File Closing**: When a file is closed, the system decrements the count field. If the count reaches zero, it means no processes are using the file, and resources can be freed.
* **File Descriptor Management**: The file table entries are often indexed by file descriptors, which are integer values used by processes to refer to open files. The file table helps map these descriptors to the actual file data and metadata.

**Example Usage:**

Imagine a process opens a file for reading and writing. The following occurs:

1. **File Open**: An entry in the file table is created or updated with the ptrinode pointing to the file's inode. The mode field indicates that the file is open for reading and writing. The readoffset and writeoffset are set to the beginning of the file.
2. **File Read**: The read operation uses the readoffset to determine where to start reading data. After reading, the readoffset is updated to point to the next position.
3. **File Write**: The write operation uses the writeoffset to determine where to start writing data. After writing, the writeoffset is updated to reflect the new end of the file.
4. **File Close**: The count field is decremented. If the count drops to zero, the file table entry can be cleaned up.

In summary, the file table is essential for managing open files, tracking their states, and ensuring correct file operations in the file system.

**Explain the use of InCore Inode Table and its use?**

In your Custom Virtual File System (CVFS) project, the **InCore Inode Table** is a crucial data structure that maintains an in-memory representation of file inodes. Here's a detailed explanation of its use and purpose within the context of your CVFS project:

**Purpose of the InCore Inode Table in CVFS**

1. **Efficient Inode Management**: The InCore Inode Table allows your CVFS to efficiently manage inodes by keeping a live copy of the inode data in memory. This reduces the need for frequent disk accesses, which can be time-consuming and resource-intensive.
2. **Performance Optimization**: By caching inode information in memory, your CVFS can quickly access and modify file metadata without the overhead of reading from or writing to disk. This results in faster file operations and improved overall system performance.
3. **Synchronization and Consistency**: The InCore Inode Table ensures that file metadata is consistently updated and synchronized. It helps manage file system operations such as opening, reading, writing, and closing files by providing up-to-date inode information.

**Contents of the InCore Inode Table in CVFS**

The InCore Inode Table in your CVFS project typically includes the following elements:

1. **Inode Structure**:
   * **Inode Number**: A unique identifier for each inode within the file system.
   * **Metadata**: Contains file-related metadata such as size, permissions, ownership, and timestamps.
   * **File Data Pointer**: A reference to the file's data or control block, which may include file content or other file-specific information.
   * **Reference Count**: Indicates how many times the inode is currently referenced. This helps manage inode lifecycle and ensures it is not prematurely deleted.
   * **Status Flags**: Flags that represent the state of the inode, such as whether it has been modified and needs to be written back to disk.
2. **InCore Inode Table Structure**:

**#define MAX\_INODES 1024**

**typedef struct {**

**int inodeNumber; // Unique identifier for the inode**

**FileMetadata metadata; // Metadata related to the file**

**void \*fileData; // Pointer to the file's data or control block**

**int referenceCount; // Number of references to this inode**

**int statusFlags; // Status flags (e.g., dirty)**

**} InCoreInode;**

**InCoreInode inodeTable[MAX\_INODES]; // Array of in-core inodes**

**Use of InCore Inode Table in CVFS Operations**

1. **File Access and Modification**:
   * When a file is accessed or modified, the InCore Inode Table is used to quickly retrieve and update the inode information. This allows for fast and efficient file operations.
2. **File Opening**:
   * Upon opening a file, your CVFS looks up the inode in the InCore Inode Table. If the inode is not in memory, it may be read from disk and added to the table.
3. **Metadata Updates**:
   * Any changes to file metadata (e.g., size, permissions) are updated in the InCore Inode Table. Periodic synchronization ensures that these changes are written back to disk as needed.
4. **File Closing**:
   * When a file is closed, the reference count for its inode is decremented. If the reference count drops to zero and the inode is marked as dirty, it will be written back to disk.
5. **Cache Management**:
   * The InCore Inode Table acts as a cache for frequently accessed or recently used inodes. This caching mechanism helps reduce disk I/O operations and improves file system performance.
6. **Synchronization with Disk**:
   * To maintain consistency, the InCore Inode Table ensures that any updates to file metadata are eventually synchronized with the on-disk inode table. This prevents discrepancies between in-memory and on-disk representations.

**Integration with Other Components**

* **File Table**: The InCore Inode Table works with the file table to provide file descriptors that reference the correct inodes for file operations.
* **Disk I/O**: The table interacts with disk I/O functions to read and write inode data as needed, ensuring that in-memory changes are reflected on disk.

The **In-Core Inode Table** is a crucial data structure used in file systems to manage inodes, which are metadata structures associated with files. Here's an explanation of its use and purpose:

**What is an Inode?**

An inode (index node) is a data structure on a filesystem on Linux and Unix-like systems that stores information about a file or a directory. This information typically includes:

* File type (regular file, directory, symbolic link, etc.)
* File size
* Ownership (user and group IDs)
* File permissions (read, write, execute)
* Timestamps (creation, modification, access)
* Pointers to the file's data blocks on disk

**In-Core Inode Table**

The In-Core Inode Table is an in-memory representation of the inode table. Here's how it fits into the filesystem structure and its uses:

**Purpose**

1. **Efficient Access**: The in-core inode table keeps frequently accessed inodes in memory, reducing the need to repeatedly access the disk. This speeds up file operations like open, read, write, and metadata retrieval.
2. **File System Operations**: It manages the inodes that are currently active (i.e., files that are open or recently accessed). When a file is opened or modified, the in-core inode is updated to reflect the changes.
3. **Caching**: By maintaining inodes in memory, the system can cache inode information, leading to faster file operations. It helps to avoid the overhead of disk I/O for every file operation.
4. **State Management**: It maintains the state of files (like their size, permissions, etc.) while they are in use. This allows the system to efficiently track changes and manage file attributes.

**Structure**

In the provided code, the in-core inode table is implemented using a linked list of INODE structures. Each INODE structure represents an inode and contains:

* FileName: Name of the file associated with the inode.
* InodeNumber: Unique identifier for the inode.
* FileSize: Total size of the file.
* FileActualSize: Actual size of the file (might be smaller than FileSize).
* FileType: Type of the file (regular or special).
* Buffer: Pointer to the file's data.
* LinkCount: Number of hard links to the file.
* ReferenceCount: Number of references to the inode (open file descriptors).
* permission: Permissions associated with the file.
* next: Pointer to the next inode in the list.

**Usage in the Code**

* **Creation**: The CreateDILB function initializes the in-core inode table, allocating inodes and setting their initial values.
* **Access**: Functions like Get\_Inode search the in-core inode table to find and return an inode based on its file name.
* **Modification**: When files are opened or modified, the in-core inodes are updated accordingly. For example, the OpenFile function updates the inode’s reference count and sets the file’s mode.
* **Deallocation**: Functions such as rm\_File update the inode’s link count and potentially deallocate the inode if there are no remaining references.

**Summary**

The In-Core Inode Table plays a vital role in file system performance and management. It enables fast access to file metadata, maintains the current state of files, and reduces the need for frequent disk access by caching inode information in memory.

**What does inode mean?**

In the provided code, an **inode** represents a data structure used in a file system to store metadata about a file. Here’s a breakdown of what an inode means in this context:

**Inode Definition:**

An **inode** (Index Node) contains various attributes of a file or directory. It does not store the file's data or name but rather information about the file, such as its size, type, and permissions. This is crucial for managing file systems, as it allows the system to keep track of file attributes efficiently.

**Inode Structure in the Code:**

The INODE structure in the code is defined as follows:

**typedef struct inode**

**{**

**char FileName[50]; // Name of the file**

**int InodeNumber; // Unique identifier for the inode**

**int FileSize; // Size of the file (not including actual data size)**

**int FileActualSize; // Actual size of the file data**

**int FileType; // Type of file (e.g., regular file, special file)**

**char \*Buffer; // Pointer to the file data**

**int LinkCount; // Number of hard links to the file**

**int ReferenceCount; // Number of references to this inode (open file descriptors)**

**int permission; // Permissions for the file (read, write, etc.)**

**struct inode \*next; // Pointer to the next inode in the linked list**

**} INODE, \*PINODE, \*\*PPINODE;**

**Key Fields in the INODE Structure:**

* **FileName**: Name of the file associated with the inode.
* **InodeNumber**: Unique identifier for the inode, used to access the file.
* **FileSize**: Size of the file, used for allocation and management purposes.
* **FileActualSize**: Actual amount of data in the file, which can be less than the FileSize if the file is sparse or truncated.
* **FileType**: Indicates whether the file is a regular file or a special file (e.g., device file).
* **Buffer**: Pointer to the buffer that holds the file's data.
* **LinkCount**: Number of directory entries (hard links) pointing to this inode.
* **ReferenceCount**: Number of file descriptors or processes that have this inode open.
* **permission**: Permissions associated with the file (e.g., read, write, execute).
* **next**: Pointer to the next inode in the linked list, used for traversing the inode list.

**Role in File Operations:**

* **File Creation**: When a file is created, a new inode is allocated and initialized with the file's metadata.
* **File Opening**: When a file is opened, its inode is referenced, and the reference count is incremented.
* **File Reading/Writing**: The file's data is accessed through its inode, with operations affecting the buffer and offsets.
* **File Deletion**: When a file is deleted, its inode is deallocated, and the link count is decremented. If the link count reaches zero, the inode is removed.

In summary, the inode in this project serves as a crucial component for managing file metadata and maintaining file system integrity.