National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Faculty of Computing

CS-330 Operating System

BESE – 14B
4th May 2025

SYSTEM DESIGN

Lab Engineer: Mr. Junaid Sajid
Instructor: Engr Taufeeq Ur Rehman

Name	CMS ID
Osama Ayaz	459700
Anoosheh Arshad	459658

Project Link: https://github.com/usama-codes/virtual-file-system

National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Overview

This Virtual File System (VFS) project simulates a **simple, lightweight file system** inside a single file (sample.dat). It uses structures like **Superblock**, **Inodes**, **Directory Entries**, and **Block Bitmaps** to manage files and directories efficiently.

Users can create, delete, move, and manipulate files and directories — all through a sleek **Dark Mode GUI** built with **Tkinter**.
The system maintains an internal hierarchy starting from a **root directory**, and each file/directory is mapped using an **inode**.

Major Components

Some of the major components of this virtual file system are:

1. Superblock
The Superblock is a critical structure in the file system that contains metadata about the file system itself. It acts as the "blueprint" for the file system.
Purpose:

- Tracks the layout of the file system.
- Helps locate key structures like the inode table, free space bitmap, and root directory.

2. Inodes
An Inode represents a file or directory in the file system. It stores metadata about the file or directory and pointers to the data blocks.
Purpose:

- Stores metadata about files and directories.
- Points to the data blocks where the file's content is stored.

3. Directory Entries
A DirectoryEntry represents an entry in a directory. Each entry maps a file or subdirectory name to its corresponding inode.
Purpose:

- Provides a mapping between file/directory names and their inodes.
- Enables hierarchical organization of files and directories.

4. Bitmaps

Bitmaps are used to track the allocation status of inodes and data blocks.

Purpose:

- Efficiently manage free and allocated resources.
- Prevent fragmentation and ensure proper allocation.

5. Data Blocks

Data blocks store the actual content of files or the directory entries for directories.

Purpose:

- For files: Store the file's content.
- For directories: Store a list of DirectoryEntry objects.

Directory Structure

The directory structure of the Virtual File System (VFS) is hierarchical, utilizing inodes and directory entries to efficiently manage files and directories. Here's an overview:

1. Root Directory

- The root directory is the entry point of the file system, represented by a special inode.
- It contains a list of directory entries, each pointing to either a file or a subdirectory.

2. Directories

- Directories are special files that store lists of entries, where each entry links to a file or another directory.
- Directories are identified by a flag in their metadata, indicating whether they contain files or other directories.

3. Files

- Files are also represented by inodes, similar to directories, but with a flag indicating they are not directories.
- A file’s inode contains metadata and pointers to data blocks where the file’s content is stored.

4. Hierarchical Structure

- The system's hierarchical structure is achieved by linking directories and files through directory entries.
- Each directory can contain both files and subdirectories, enabling a nested file organization.

5. Directory Operations

- Common operations on directories include creating new directories, navigating between directories, and listing the directory contents.
- Navigation is achieved by updating the reference to the current directory, allowing users to move through the file system.

Threading Logic

1. Threading Logic Overview

- A **master input file** is selected for which copies are made for each thread.
- Reads from a file like **input_thread<x>.txt**.
- Executes commands line-by-line using `execute_command()`
- Writes results (including memory map, outputs, or errors) to **output_thread<x>.txt**.
- Each thread shares the **same virtual file system** (sample.dat), so concurrent access must be synchronized to avoid data corruption.

2. Locking Strategy

- **How it works:** A global `fs_lock = threading.Lock()` is used for mutual exclusion. The `fs_lock` ensures that only **one thread** at a time can perform operations that modify or read the shared filesystem image or shared data structures.
- It prevents race conditions in shared structures like the open file table or the filesystem image.

3. Open File Table and Race Conditions:

- `threading.local()` creates an object that stores data **specific to each thread**. Each thread sees its **own independent copy** of any attributes set on it.
- When one thread sets `thread_local_data.open_file_table`, **other threads don't see it** — they each get their own `open_file_table.`
- This prevents **race conditions** by isolation rather than locking.
- This also helps prevent shared state issues

Functions

1. **thread_worker(thread_id)**

Parameters:

- **thread_id:** The ID of the thread (integer). Used to differentiate between input/output files.

How it Works with Components:

- **Filesystem Lock:** The function uses filesystem_lock to ensure thread synchronization when accessing the shared file system resources, preventing race conditions when multiple threads are executing commands.
- **File Operations:** Each thread reads commands from a specific input file (input_thread<x>.txt) and writes the results to an output file (output_thread<x>.txt).
- For **each command**, the thread acquires the filesystem_lock, executes the command using **execute_command(command, outfile)**, and writes the result to the output file.

2. **execute_command(command, outfile=None)**

Parameters:

- **command**: The command string to be executed (string). This contains the operation and its associated parameters.
- **outfile:** The file to write output to (optional). If provided, results can be logged to this file.

How it Works with Components:

- **Command Parsing**: The function begins by splitting the command into parts using **command.split(),** which separates the command keyword (e.g., create, open, write_to_file, etc.) and the parameters associated with it.

3. **Create(fName)**

Parameters:
- **fname:** Name of the file that you want to create (string)

How it Works with Components:

- **Superblock:** Uses block_size to calculate the number of blocks required for the file content.

- **Inode:** Allocates a new inode for the file. Updates the inode's file_size, creation_time, modification_time, and direct_blocks with the allocated blocks.
- **Free Block Bitmap:** Reads the bitmap to find free blocks for the file content. Updates the bitmap to mark the allocated blocks as used.
- **Inode Bitmap:** Reads the bitmap to find a free inode. Updates the bitmap to mark the allocated inode as used.
- **Directory Entries:** Adds a new DirectoryEntry for the file in the parent directory's data block.

4. Delete(fName)

Parameters:

- **fname:** Name of the file to be deleted (string)

How it Works with Components:

- **Superblock**: Uses block_size to locate the file's data blocks.
- **Inode:** Reads the inode of the file to get its metadata and data block pointers. Resets the inode to a default state after deletion.
- **Free Block Bitmap:** Updates the bitmap to mark the file's data blocks as free.
- **Inode Bitmap:** Updates the bitmap to mark the file's inode as free.
- **Directory Entries:** Removes the DirectoryEntry for the file from the parent directory's data block.

5. Mkdir(dirName)

Parameters:

- **dirName:** Name of the directory to create(string)

How it Works with Components:

- **Superblock:** Uses block_size to allocate a block for the new directory's entries.
- **Inode:** Allocates a new inode for the directory. Sets is_directory to True and assigns a data block for the directory entries.
- **Free Block Bitmap:** Updates the bitmap to mark the allocated block as used.
- **Inode Bitmap:** Updates the bitmap to mark the allocated inode as used.
- **Directory Entries:** Adds a new DirectoryEntry for the directory in the parent directory's data block.

6. **chDir(dirName)**

Parameters:

- **dirName**: Name of the directory to change to(string)

How it Works with Components:

- **Superblock**: Uses block_size to locate the directory's data block.
- **Inode:** Reads the inode of the target directory to verify it is a directory. Updates the cwd_inode to the target directory's inode number.
- **Directory Entries:** Searches the parent directory's entries to find the target directory.

7. **Move(source_fName, target_dirName)**

Parameters:

- **source_fName**: Source file name to be moved (String)
- **target_dirName**: Name of the target directory (file to be moved there) (String)

How it Works with Components:

- **Superblock:** Uses block_size to locate the source and target directories' data blocks.
- **Inode:** Reads the inode of the source file or directory. Updates the parent directory's entries to remove the source and add it to the target directory.
- **Directory Entries:** Removes the DirectoryEntry for the source from the current directory and adds it to the target directory.

8. **Open(fName,mode)**

Parameters:

- **fname:** Name of the file to open (String)
- **mode**: The mode in which to open(read, write etc) (char)

How it Works with Components:

- **Superblock:** Uses block_size to locate the file's data blocks.
- **Inode:** Reads the inode of the file to get its metadata and data block pointers.
- **open_file_table:** Creates a FileObject for the file and adds it to the open_file_table.

9. close_file(fName)

Parameters:

- **fName:** Name of the file to close (string).

How it Works with Components:

- **open_file_table:** Checks if fName exists in open_file_table.
- **FileObject:** If present, closes the file's open file stream (fileObj.fs.close()). Deletes the entry from open_file_table.
- **Superblock/Inodes/Bitmaps:** No interaction. This function only manages in-memory structures.

10. fileObj.Write_to_file(text) / fileObj.write_to_file(write_at, text):

Parameters:

- **text:** Text to write (string).
- **write_at** (optional in second version): Byte offset where writing should begin (integer).

How it Works with Components:

- **Superblock:** Reads block_size to manage block boundaries.
- **Inode:** Reads existing file size from inode. Updates file_size and modification_time after writing.
- **Direct Blocks (Inode direct_blocks[]):** Writes data into blocks. If not enough blocks are assigned, it allocates new ones.
- **Free Block Bitmap**: Reads bitmap to find free blocks. Updates bitmap to mark new blocks as used.
- **open_file_table:** Writes directly into the file object opened earlier.

11. fileObj.Read_from_file() / fileObj.Read_from_file(start, size):

Parameters:

- **start:** Start byte offset (integer, optional).
- **size:** Number of bytes to read (integer, optional).

How it Works with Components:

- **Superblock:** Uses block_size to calculate which block to start reading from.
- **Inode:** Reads inode to know file size and direct blocks.
- **Direct Blocks:** Reads data from corresponding blocks sequentially starting at start.

- **open_file_table:** Reads through the already opened file object (no direct bitmap modification).

12. fileObj.Move_within_file(start, size, target)

Parameters:

- **start:** Starting byte offset of the segment to move (integer).
- **size:** Number of bytes to move (integer).
- **target:** Byte offset to insert the moved segment (integer).

How it Works with Components:

- **Superblock:** For block size info to manage block alignment.
- **Inode:** Reads file size. After movement, updates inode metadata if necessary (modification time).
- **Direct Blocks:** Reads blocks, modifies content in memory. Overwrites the blocks with the new updated content.
- **open_file_table:** Temporary read and write done through FileObject stream.

13. fileObj.Truncate_file(maxSize)

Parameters:

- **maxSize:** New file size (integer).

How it Works with Components:

- **Superblock:** Calculates blocks needed based on block size.
- **Inode:** Shrinks/expands file to match maxSize. Updates file_size and modification_time.
- **Direct Blocks:** If file is shrunk: Releases unneeded blocks and Sets released block entries to None.
- **Free Block Bitmap:** Updates bitmap to mark released blocks as free if file shrinks.

14. show_memory_map(fs_image)

Parameters:

- **fs_image:** Name of the filesystem file (like sample.dat) (string).

How it Works with Components:

- **Superblock:** Fetches block size for offset calculations.
- **Inode Table:** Reads all inodes one by one. Shows which inode belongs to which file or directory.
- **Free Block Bitmap:** Displays which blocks are used (for file data or directory entries).
- **Directory Entries:** Reads directory blocks to map filenames with inode numbers.
- **Root Directory:** Specifically lists the contents of the root directory (/).