Project 3 Report

Xiaoyu Chen (Student ID: 523031910681)

May 21, 2025

Contents

1 Step 1: Disk-storage system

1.1 Description

Implement a simulation of a physical disk, which is organized by cylinder and sector. Assume the sector size is ffxed at 256 bytes, and store the actual data in a real disk file. Additionally, use something to account for track-to-track time.

1.1.1 Command line

```
./disk <#cylinders> <#sector per cylinder> <track-to-track delay> <#disk-storage-filename>
```

1.1.2 Protocol

- I: Information request. The disk returns two integers representing the disk geometry: the number of cylinders and the number of sectors per cylinder.
- R c s: Read request for the contents of cylinder c sector s. The disk returns Yes followed by a whitespace and those 256 bytes of information (hex), or No if no such block exists.
- W c s data: Write a request for cylinder c sector s. The disk returns Yes and writes the data (hex) to cylinder c sector s if it is a valid write request or returns No otherwise.
- E: Exit the disk-storage system.

1.2 Design and implementation

Initialization and File Mapping The program accepts four arguments: number of cylinders, sectors per cylinder, track-to-track delay (currently unused), and the name of the disk file. The total disk size is computed as:

$$\text{FILESIZE} = \#\text{cylinders} \times \#\text{sectors per cylinder} \times 256$$

The file is stretched to this size using `lseek()` and then mapped into memory using `mmap()`, allowing direct read/write access to the "disk" via pointer arithmetic. This design significantly reduces I/O overhead compared to traditional `read()` or `write()` syscalls.

Command Parsing and Protocol Handling The disk server continuously reads commands from `stdin`, parses them using `strtok()`, and executes them via a `switch` statement based on the operation type:

- **I (Information)**: Returns the number of cylinders and sectors per cylinder. Useful for clients to understand disk geometry.

- **R c s (Read)**: Calculates the offset in memory via offset $= c \times n \times 256 + s \times 256$, reads 256 bytes into a buffer, and logs the result. Boundary checks ensure access is within disk limits.
- **W c s data (Write)**: Similar offset logic as read; the data is zero-padded to 256 bytes and written directly into the mapped region. The disk logs the result.
- **E (Exit)**: Flushes logs, unmaps the disk, and exits gracefully.

Logging and Robustness All operations are logged into `disk.log`, which serves as both a debug trace and a behavior audit. The implementation includes robust error handling: invalid commands and out-of-bound accesses are detected and result in log entries of `NO`, while valid actions are confirmed with `YES`.

2 Step2: File system

2.1 Description

Implement a memory-based file system. The file system should provide basic file operations, described in the protocol.

2.1.1 Command line

```
./fs
```

2.1.2 Protocol

- **f**: Format. This will format the file system on the disk, by initializing any/all of the tables that the file system relies on.
- **mk f**: Create file. This will create a file named `f` in the file system.
- **mkdir d**: Create directory. This will create a subdirectory named `d` in the current directory.
- **rm f**: Delete file. This will delete the file named `f` from the current directory.
- **cd path**: Change directory. This will change the current working directory to `path`. The path follows Linux format and can be either relative or absolute. The initial working path is `/`. You must handle `.` and `..`
- **rmdir d**: Delete directory. This will delete the directory named `d` in the current directory.
- **ls**: Directory listing. This will return a listing of the files and directories in the current directory. You are also required to output additional information, such as file size, last update time, etc.

- **cat f**: Read file. This will read the file named `f`, and return the data contained in it.
- **w f l data**: Write file. This will overwrite the contents of the file named `f` with `l` bytes of `data`. If the new data is longer than the old one, the file will grow; if shorter, it will be truncated.
- **i f pos l data**: Insert into file. This will insert `l` bytes of `data` into the file `f` at position `pos` (0-indexed). If `pos` exceeds the file size, insert at the end.
- **d f pos l**: Delete in file. This will delete `l` bytes from file `f` starting at position `pos` (0-indexed), or delete to the end if fewer than `l` bytes remain.
- **e**: Exit the file system.

2.2 Design and implementation

2.2.1 Block Design

The block management module is responsible for simulating disk space allocation, deallocation, and read/write operations. It provides the foundational data handling for the file system by abstracting the low-level block-based storage operations.

Superblock Structure The `superblock` is a critical structure that holds metadata about the file system layout. It contains the following fields:

- `magic`: a magic number used to identify the file system format.
- `size`: the total number of blocks available on the disk.
- `bmapstart`: the starting block index of the bitmap used to track block allocation.

This structure is defined globally as `sb`, and is updated when retrieving disk geometry via `get_disk_info`.

Disk Simulation The disk is simulated as a 2D byte array:

```
uchar disk[MAX_BLOCKS][BSIZE];
```

Each block is of size `BSIZE`, and the maximum number of blocks is defined by `MAX_BLOCKS = 1024`. The function `init_disk()` initializes the entire disk content to zero.

Block Bitmap Management Block allocation status is maintained using a bitmap stored at block `sb.bmapstart`. Each bit represents the usage state of a block:

- `mark_block_used(bno)` sets the bit corresponding to block `bno`, indicating it is allocated.
- `mark_block_free(bno)` clears the bit, marking the block as free.
- `is_block_used(bno)` checks whether the given block is currently in use.

These functions operate by reading and updating the bitmap block using `read_block` and `write_block`.

Block Allocation and Deallocation `allocate_block()` scans the bitmap to find the first free block starting from block number 2 (reserving blocks 0 and 1), marks it as used, zeroes out its contents using `zero_block()`, and returns the block number. If no blocks are free, a warning is issued.

`free_block(bno)` validates the block number and ensures it is currently allocated before freeing and zeroing it.

Block Read and Write `read_block()` and `write_block()` respectively perform read and write operations on the disk array, with bounds checking based on `sb.size` to ensure safe access.

Disk Geometry The disk is configured with a geometry of 16 cylinders and 64 sectors per cylinder, yielding a total of 1024 blocks. The function `get_disk_info()` sets this information and ensures that it does not exceed the `MAX_BLOCKS` limit.

2.2.2 Inode Design

The inode (index node) serves as the fundamental metadata unit for file system objects, storing ownership, permission, timestamps, and block mapping information independently of directory entries. It abstracts file data layout from file names, allowing flexible and efficient file system management.

Data Structure Two structures are used to represent inodes:

- `struct dinode`: the on-disk format of the inode. This structure resides in disk blocks and contains persistent metadata and pointers to data blocks.
- `struct inode`: the in-memory cached version of a `dinode`, including runtime metadata such as reference count, inode number, and lock state.

The `dinode` structure includes the following key fields:

- **Type and Metadata**: `type` (file or directory), `nlink` (link count), `owner_id`, `group_id`, `mode` (file permissions), `atime`, and `mtime`.

- **Size and Block Mapping**: `size` indicates the total size of the file in bytes. The file data is located using:
  - `addrs[10]`: Direct pointers to data blocks.
  - `indirect`: A single indirect block pointer.
  - `dindirect`: A double-indirect block pointer.

Block Mapping Strategy Logical file blocks are translated to physical disk blocks using the `bmap()` function, which implements a hierarchical mapping scheme:

- Direct blocks (first 10 blocks) are accessed via `addrs[0]--addrs[9]`.
- The 11^{th} block is accessed through a single-level indirect pointer.
- The 12^{th} and beyond are handled via a double-indirect pointer to access a second-level index.

This strategy optimizes access performance for small files while supporting large files with minimal memory overhead.

Inode Lifecycle: Allocation, Use, and Release

- `ialloc()`: Allocates a free inode from the inode bitmap, initializes its metadata, and returns a pointer to the in-memory `inode` structure.
- `iupdate()`: Synchronizes the in-memory inode back to the on-disk `dinode` structure. Called after inode modifications such as file writes or metadata changes.
- `iput()`: Decreases the reference count of an in-memory inode. If the count reaches zero and the inode has no links, it triggers `deletei()` to recycle all blocks and clears the inode for reuse.

The `iput()` function plays a key role in inode reference management. It is often invoked after a command finishes using an inode (e.g., after reading, writing, or deletion), ensuring correct memory and resource cleanup.

Inode Loading from Disk `iget()` loads an inode from disk into memory, returning a pointer to a newly allocated in-memory `inode` structure. It reads the on-disk `dinode` block corresponding to the given inode number (`inum`), validates it, and copies all relevant metadata and block pointers. The returned `inode` starts with a reference count of one and is marked valid and clean. If the inode number is invalid or the inode is unused (type 0), it returns `NULL`.

Inode Access and File Deletion

File I/O is handled through:

- `readi()`: Reads file content by resolving logical offsets via `bmap()`, supporting partial reads.
- `writei()`: Writes to a file, allocating new blocks as needed via `bmap()`, and updating the inode's size accordingly.
- `deletei()`: Recursively deallocates all data blocks pointed to by the inode, including those reached through indirect and double-indirect pointers. The inode is then marked free.

The `deletei()` function is directly used by the file system command `cmd_d()`, which handles file deletion requests from the user.

2.2.3 File system design

Some Important Ideas

The design of our file management system is centered around simplicity, full utilization of blocks, and ease of management.

To begin with, we designate block 0 for storing the `superblock`, and block 1 for the `bitmap` that tracks free and used blocks. Beyond that, we make no strict distinction between inode blocks and data blocks. However, during the execution of `cmd_f()`, the root directory is created by default, and its inode is stored in block 2. To simplify the system, we assume that each inode occupies exactly one block, and treat the inode number (`inum`) as the corresponding block number.

A key feature of our design is that all operations are performed relative to the current working directory. For this purpose, we maintain a global pointer `cwd` that holds the inode of the current directory. When executing any file operation, the system first locates the target inode (of a file or subdirectory) starting from `cwd`, and then accesses the associated data blocks based on the information stored in that inode.

Another important aspect of the design is that files and directories are structurally unified. Both are represented by inodes, with the difference being in the type of data they point to. A file inode refers to binary data blocks, while a directory inode points to data blocks containing an array of `dirent` structures. As a result, many file and directory operations can share common logic, with only minor differences in the way data is interpreted.

To prevent stack overflows during directory traversal and listing, we deliberately designed the `dirent` structure to occupy exactly 32 bytes. This ensures safe and efficient memory use when reading directory contents.

Global Variables

- `inode *root`: The root inode pointer, representing the root directory of the file system. It is initialized at system startup and serves as the entry point for absolute path resolution.

- `inode *cwd`: The current working directory inode pointer. It changes with user navigation (e.g., via `cmd_cd`) and determines the context for relative path operations.
- `int current_uid`: Represents the current user ID, used for permission checking during read/write operations and user-specific access control.
- `uint direntoff, direntblock`: These two variables are used internally during directory traversal to record the offset and block number of a directory entry, especially during operations like lookup and insertion.

entry Structure

```
typedef struct {
    short type;
    char name[MAXNAME];
    uint inum;
    uint size;
    ushort mode;
    uint mtime;
} entry;
```

This structure is used in the `cmd_ls` command to represent a high-level view of a directory entry. Each `entry` holds not only the file name and inode number but also metadata such as type (file or directory), size, access permissions, and last modified time. It acts as a convenient abstraction layer between low-level directory parsing and user-facing output formatting.

dirent Structure

```
struct dirent {
    uint inum;
    char name[MAXNAME];
};
```

The `dirent` structure represents the raw on-disk directory entry format. Each entry maps a file or directory name to an inode number. The file system traverses and manipulates `dirent` records directly when performing directory-related operations such as creation, deletion, and lookup.

Key Function Declarations Several core command interfaces are exposed by the file system, with each function typically corresponding to a shell-like command:

- `cmd_f`: Formats the disk and initializes the file system. It calls `ialloc` to allocate the root directory inode, uses `dirlink` to write the "." and ".." entries, and then writes the superblock, bitmap, and inode table to disk using `iupdate` and `write_block`.

- `cmd_mk`: Creates a regular file. It uses `namei` and `dirlookup` to check if the file already exists. If not, it allocates a new inode with `ialloc` and adds the file to the current directory using `dirlink`.

- `cmd_rm`: Removes a regular file. It calls `namei` to locate the file's inode, verifies the inode type, and then proceeds to update the directory. Instead of using the common approach of replacing the deleted entry with the last one, it reads all directory entries, removes the target entry, and rewrites the remaining entries back to the block in a compact form. After updating the directory, it decreases the inode's link count, releases it with `iput`, and calls `deletei` to update the inode's block pointers accordingly.

- `cmd_mkdir`: Creates a subdirectory. It allocates a new directory inode via `ialloc`, initializes the "." and ".." entries, and links the new directory into its parent using `dirlink`.

- `cmd_rmdir`: Removes a directory. It locates the inode via `namei` and recursively traverses its contents. If a file is encountered, it calls `cmd_rm` to remove it; if a subdirectory is found, it descends into that directory and repeats the removal process. This recursive approach ensures that all files and subdirectories under the target directory are properly deleted and their resources reclaimed before the directory itself is removed.

- `cmd_cd`: Changes the current working directory. It first checks whether the input path is an absolute path (i.e., starts with '/'). If so, it resets `cwd` to the root directory before processing the rest of the path. The function then splits the path by '/' and sequentially resolves each component. Special cases like "." (current directory) and ".." (parent directory) are explicitly handled. For regular directory names, it uses `namei` to resolve the inode. If any component cannot be resolved, it restores the original `cwd` to prevent inconsistent state. On success, `cwd` is updated to the target directory.

- `cmd_ls`: Lists the entries in the current working directory. It first scans the directory to count valid entries, excluding the special entries `"."` and `".."`. Then, it allocates memory to store the result. In the second pass, it reads each valid directory entry, retrieves the corresponding inode using `iget`, and collects its metadata—such as type, inode number, size, access mode, and modification time—into an `entry` structure array. This structured output can be used to display detailed directory contents.

- `cmd_cat`: Displays the contents of a file. It uses `namei` to find the inode and reads the file's data block by block using `read_block`, printing each byte to standard output.

- `cmd_w`: Overwrites a file with new data. It locates the target file via `namei` and uses `writei` to write the specified content starting from offset 0, effectively replacing the file's existing contents. After writing, it updates the file's modification time using `iupdate`.

- `cmd_i`: Inserts data into a file at a specified position. It first resolves the target file using `namei` and verifies that the insertion position is valid. Then, it reads the original file content using `cmd_cat`, allocates a new buffer, and constructs the new file content

by inserting the given data at the specified offset. Finally, it writes the modified content back to the file with `writei`, updates the modification time, and releases the inode.

- `cmd_d`: Deletes a specified range of data from a file. It locates the file via `namei`, verifies that the deletion range is valid, and reads the original file content using `cmd_cat`. A new buffer is constructed by skipping the specified range, and the modified content is written back using `writei`. Finally, it calls `deletei` to update the inode's block pointer usage and frees allocated memory.

- `cmd_login`: Logs in a user by setting the global `current_uid`. It checks if the given user ID is valid (non-negative) and, if so, assigns it to `current_uid`, allowing subsequent operations to proceed under that user's identity.

- `lookup`: Locates a directory entry by name within a given directory inode. It traverses the directory's data blocks to find a matching name, sets global variables `direntblock` and `direntoff` to record the block and offset of the entry, and returns the inode number if found, or 0 otherwise.

- `dirlink`: Adds a new entry to a directory. It first checks if the name already exists using `lookup`. If not, it scans for a free directory entry or appends a new one at the end, sets its inode number and name, and writes it back to disk using `writei`.

- `fs_shutdown`: Shuts down the file system by releasing the current working directory and root inodes using `iput`, and sets both `cwd` and `root` to `NULL`.

- `sbinit`: Initializes the file system by reading the superblock from disk into the global variable `sb`. It then loads the root inode and sets the current working directory (`cwd`) to the root.

- `namei`: Resolves a file or directory name within the current working directory to its inode pointer. It uses `lookup` to find the inode number, and then returns the corresponding inode structure using `iget`.

These foundational definitions provide both a high-level interface for user operations and low-level tools for inode and directory management, supporting a simple yet extensible file system framework.

2.2.4 Operation Design

The file system operation layer is implemented as a command-driven interface, where each supported command corresponds to a specific handler function responsible for parsing command arguments and invoking underlying file system APIs.

The main command handlers, such as `handle_mk`, `handle_mkdir`, `handle_rm`, `handle_cd`, `handle_ls`, and others, follow a similar pattern: they parse arguments using `strtok()`, perform necessary validity checks, and then call corresponding `cmd_*` functions to execute the requested operation. Success or failure responses are standardized through macros like `ReplyYes()`, `ReplyNo()`, and `ReplyDone()`, which handle both output messages and logging.

A key design aspect is the use of a `cmd_table` — a static array of structures mapping command strings to their handler functions:

```
static struct {
    const char *name;
    int (*handler)(char *);
} cmd_table[] = {
    {"f", handle_f},
    {"mk", handle_mk},
    {"mkdir", handle_mkdir},
    {"rm", handle_rm},
    {"cd", handle_cd},
    {"rmdir", handle_rmdir},
    {"ls", handle_ls},
    {"cat", handle_cat},
    {"w", handle_w},
    {"i", handle_i},
    {"d", handle_d},
    {"e", handle_e},
    {"login", handle_login}
};
```

This command dispatch table allows the main input loop to efficiently parse user input lines, extract the command token, and locate the corresponding handler by simple string comparison. The handler is then called with the remainder of the input line as its argument, enabling modular and extensible command processing.

The `main()` function continuously reads user commands from standard input, strips trailing newlines, logs the input, and looks up the handler in `cmd_table`. If a matching command is found, it delegates processing to the handler; otherwise, it reports an unknown command. The loop terminates on explicit exit commands or end-of-file.

3 Step3: Work together

3.1 Description

In this part, we will combine the disk-storage and file systems.

First, change the disk-storage system to a disk-storage server, and let the file system be the client of the server.

Second, treat the file system as a network file system server, and write a client for this server.

3.1.1 Command line

```
./disk <#cylinders> <#sectors per cylinder> <track-to-track delay> <#disk-
    storage-filename> <DiskPort>
```

```
./fs <DiskPort> <FSPort>
./client <FSPort>
```

3.2 Design and implementation for multiple users

In this section, we introduce support for a multi-user file management system. The system allows multiple users to operate within a shared environment while maintaining access control. The user with an `owner_id` of 0 is designated as the administrator and is granted full privileges to operate on all files and directories. In contrast, regular users are restricted to accessing and modifying only the files and directories they have created. Despite these access limitations, all users share a common root directory, enabling a unified file system structure.

Since socket communication was already used and thoroughly discussed in Project 1, we will not elaborate on it again here. Instead, we focus on how the file system (fs) is designed to function as both a client and a server, and the specific configurations and adjustments we made to support multi-user functionality.

3.2.1 Server Thread and Client Handling

To enable the file system to serve multiple users concurrently, we implemented a multi-threaded server model. The main process first establishes a client connection to the disk server using a TCP socket. After successful initialization and superblock loading, the file system then launches its own server thread to accept client connections from users.

fs_server_thread This function sets up a listening TCP socket on the file system's designated port. It continuously accepts incoming connections using `accept()`, and for each successful connection, it allocates memory to store the client file descriptor and spawns a dedicated thread using `pthread_create`.

- The server socket is initialized with `SO_REUSEADDR` to allow fast reuse after restarts.
- Each accepted connection is detached (`pthread_detach`) to enable automatic cleanup after the thread exits.
- If `pthread_create` fails, the connection is closed immediately to avoid leaks.

handle_client_thread This function is responsible for maintaining the lifecycle of a single client connection. Its main responsibilities include:

1. **Session Initialization:** Upon start, the function increments the global `active_clients` counter (protected by a mutex), and initializes a `User` object corresponding to the `clientfd`. The user is assigned:
   - `uid = -1`, indicating the user is unauthenticated;
   - `current_dir = root->inum`, placing them at the root directory;
   - `client_id = clientfd`, used for indexing in the global user table.

2. **Command Loop:** The thread enters a loop where it continuously:
    - Receives data from the client via `recv()`;
    - Trims newline characters and tokenizes the command;
    - Matches the command against a predefined `cmd_table`;
    - Executes the handler function under a global mutex `fs_lock` to ensure file system consistency;
    - Sends the response back using `send()`.

    If the command is unrecognized, a generic `"No"` response is returned.
3. **Thread Termination:** If the client disconnects or the command handler returns a negative value, the loop exits. The file descriptor is closed and the `active_clients` counter is decremented. If no clients remain, the server exits gracefully using `exit(0)` to prevent resource leakage.

3.2.2 User Identity and Access Control

When the client sends a command to the file management system, it must specify which client is issuing the request. The system uses this `client_id` to locate the corresponding user and obtain their `uid`.

We follow Linux-style permission settings: the root directory is assigned a default `mode` of `0777`, allowing all users to read, write, and execute. All other files are created with `0700`, and directories with `0701`, so that users are fully isolated and cannot interfere with each other.

We implement two permission-checking functions:

```
int check_read_permission(inode *ip, int uid) {
    if(ip == NULL)
        return 0;
    if(uid == 0)
        return 1;

    if(ip->owner_id == (uint)uid)
        return (ip->mode & 0400) != 0;

    return (ip->mode & 0004) != 0;
}

int check_write_permission(inode *ip, int uid) {
    if(ip == NULL)
        return 0;
    if(uid == 0)
        return 1;
```

```
    if(ip->owner_id == (uint)uid)
        return (ip->mode & 0200) != 0;

    return (ip->mode & 0002) != 0;
}
```

Here, `uid == 0` is treated as the superuser (administrator) and is always granted access.

The `User` structure is defined as follows:

```
typedef struct{
    int client_id;
    int uid;
    int current_dir;
} User;
```

Our core design principle is that all file operations are performed relative to the user's current working directory. Although we maintain a global variable `cwd`, we ensure that `cwd` is temporarily set to `users[i].current_dir` before handling each command. This prevents confusion and inconsistency from managing multiple `cwd` instances simultaneously.

In addition, we enforce that only the administrator user (with `uid == 0`) is allowed to execute the `cmd_f` command, in order to ensure system safety.

4 Conclusion

In this project, we have progressively developed a network file storage system from the ground up. Among the various stages, Step 2 was particularly time-consuming and challenging, yet it proved to be an invaluable learning experience. Through this process, I significantly improved my proficiency in string manipulation, C programming, and debugging by carefully analyzing error logs.

During development, the two most frequent issues I encountered were stack overflow and memory leaks. Often, seemingly trivial mistakes—such as an incomplete instruction or improper buffer management—could lead to these critical problems. This experience highlighted the complexity and rigor required in system design, especially when dealing with low-level memory and concurrency concerns.

Moreover, the project deepened my understanding of client-server communication, multi-threaded programming, and synchronization mechanisms essential for ensuring data consistency and thread safety in concurrent environments. Implementing the server to handle multiple clients simultaneously while maintaining filesystem integrity was a key milestone.

Looking forward, there are several avenues for further enhancement, including improving error handling to provide more informative feedback to clients. Additionally, integrating persistent storage mechanisms with better fault tolerance and exploring caching strategies could significantly improve system performance and reliability.

Overall, this project not only strengthened my practical programming skills but also gave me a deeper appreciation for the challenges inherent in building robust and secure distributed storage systems.