UNIVERSITÉ Concordia UNIVERSITY

COEN 346 -

Operating Systems

Programming Assignment #2 -

File Sharing Server

Joyal Biju Kulangara

ID: 40237314

Le Tuong Vy Pham

ID: 40248938

Team Name: Team 101

Professor: Dr. Paula Lago

Table of Contents

1. Introduction

This project focuses on a small file system that runs over a client-server architecture. The main goal was to simulate how a real operating system handles files, blocks, metadata, and concurrent access.

Our file system stores all the data inside a single disk file (*filesystem.data*) and uses fixed-size blocks along with a linked-list type structure to represent all the file contents. The metadata for each of the file is stored in an inode table, whereby the actual data is managed through a block table. The logic of the file contents are handled inside the *FileSystemManager*, which is the core of the system.

To interact with the filesystem, a TCP server (*FileServer)* was implemented, whereby it aims to accept commands such as CREATE, WRITE, READ, DELETE, and LIST. A separate client program (*FileClient*) connects to the server and sends these requests. This setup allowed us to seamlessly test multiple clients at the same time, which helps in verifying that our locking mechanism works even when several threads access the system together.

2. Data Structures and Functions

This section discusses all classes, fields, and helper functions used in the file system implementation. It also includes the UML diagrams for the main components.

2.1. UML Diagrams

- **FileSystemManager**

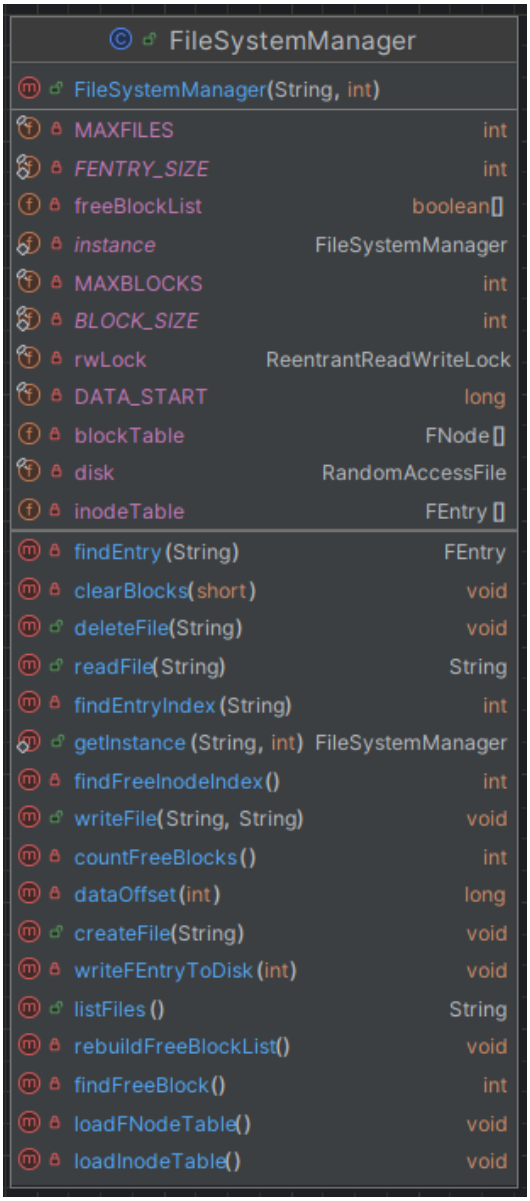

Figure 1. FileSystemManager UML Diagram

- **FileServer**

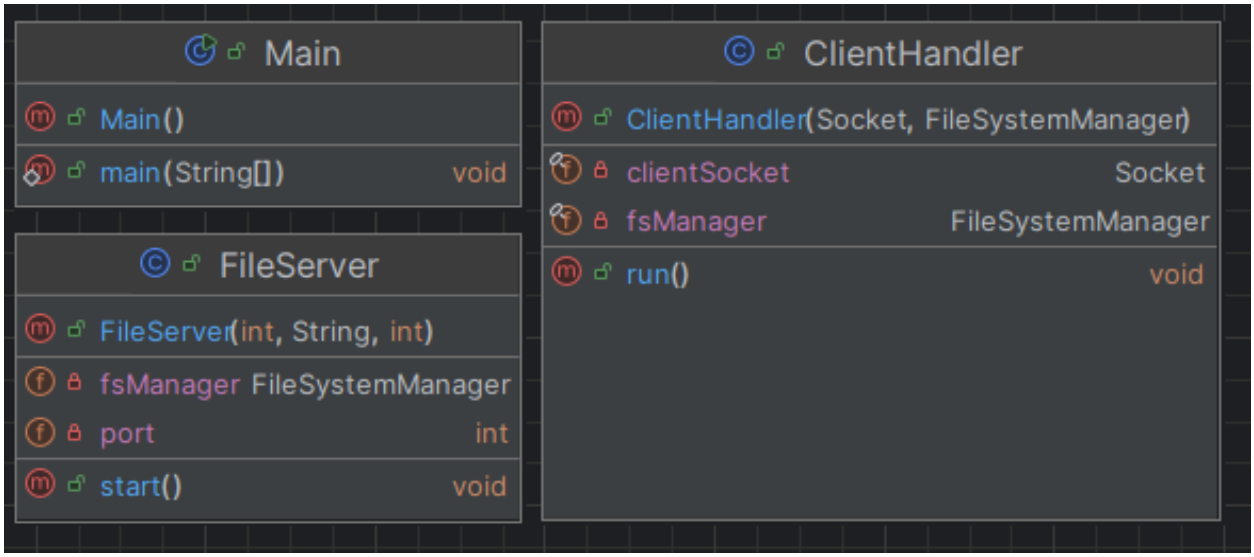

Figure 2. FileServer UML diagram

- **FileClient**

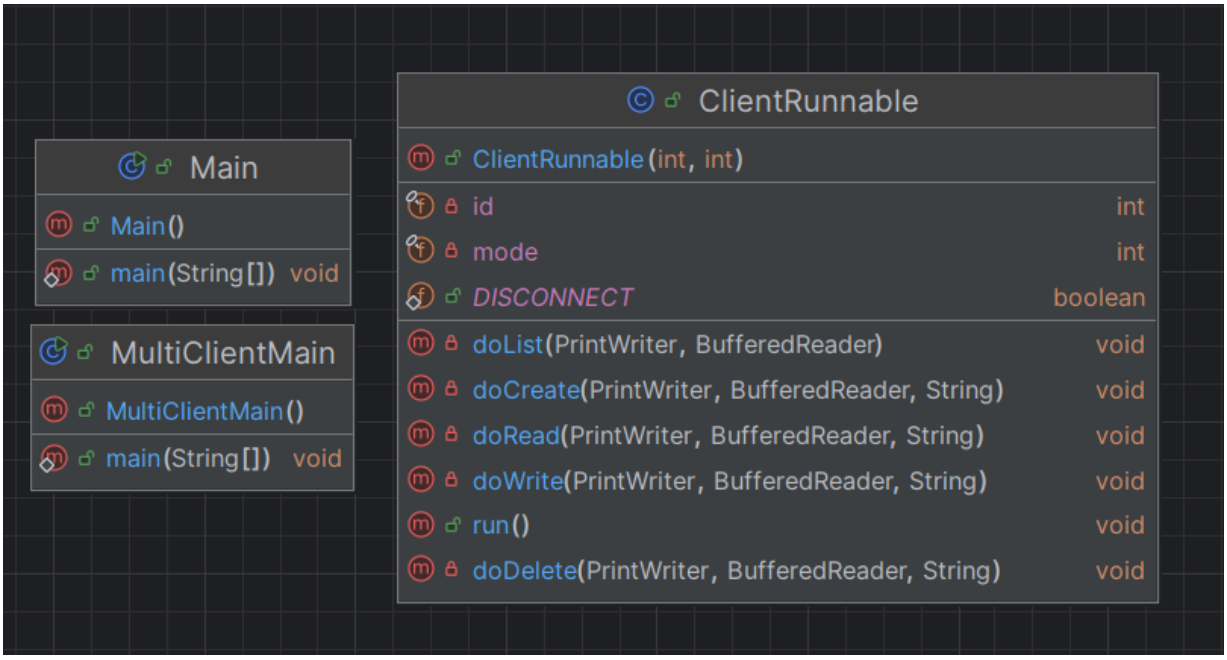

Figure 3. FileClient UML diagram

- **FEntry and FNode**

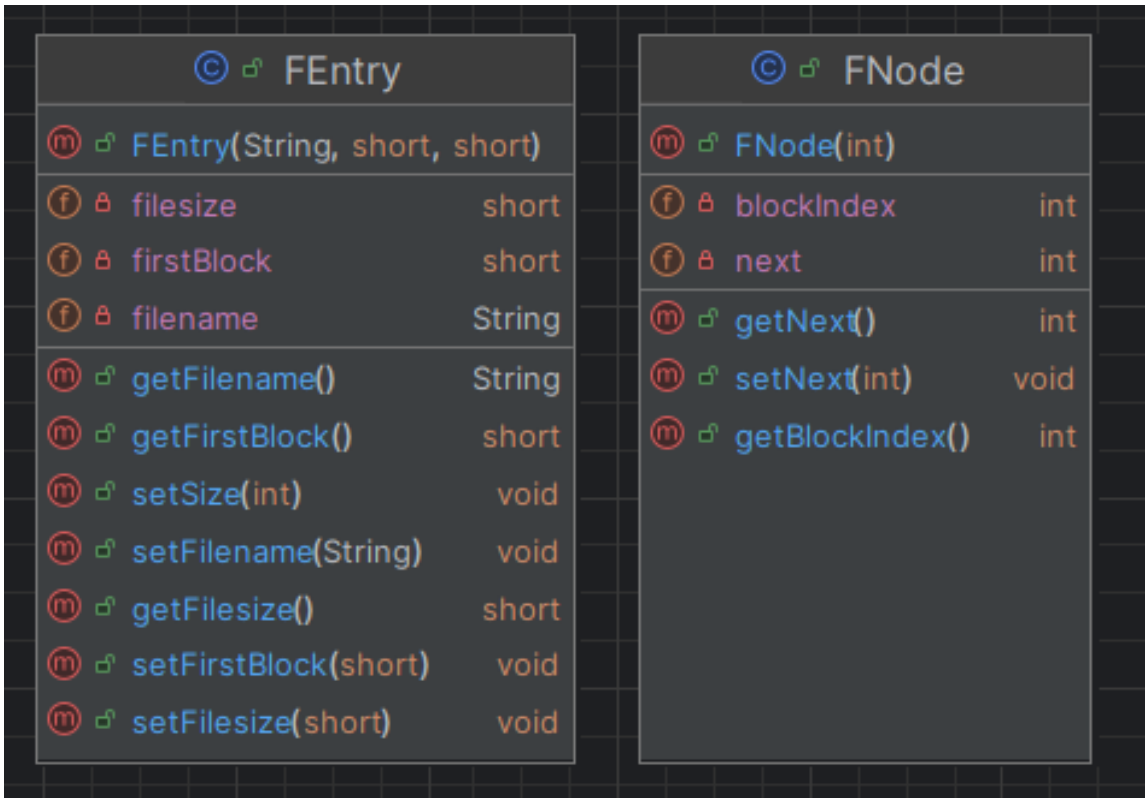

Figure 4. FEntry and FNode UML diagram

n.b: *These UML Diagrams were curated directly from Intellij IDE.*

2.2. Class Explanations

2.2.1. FileSystemManager

The FileSystemManager class contains the core logic for the filesystem. It is responsible for:

- Managing the disk file *(filesystem.dat)* and allocates and frees data blocks.
- Maintains the inode table (FEntry[ ])
- Handles file operations: CREATE, WRITE, READ, DELETE, LIST
- Ensures thread safety using a global *ReentrantReadWriteLock*

Core Fields (Conceptual):

Field	Explanation
MAXFILES = 5	Maximum number of files allowed (fixed *idnode* table size)
MAXBLOCKS =10	Total number of data blocks available on disk
BLOCK_SIZE = 128	Size of each block in bytes
FENTRY_SIZE = 15	Size of each *FEntry* on disk (11-byte name + 4-byte size + 1-byte first block)
FEntry[ ] inodeTable	Stores metadata for each file (filename, filesize, firstBlock)
FNode[ ] blockTable	Represents each disk block and its next pointer
Boolean[ ] freeBlockList	Tracks which blocks are free or used
RandomAccessFile disk	Low-level access to the persistent disk file
ReentrantReadWriteLock rwLock	Ensures thread-safe reads/writes (race condition)
static FileSystemManager instance	Singleton shared by all server threads

Constructor Overview:

When the filesystem starts, the constructor:

- Opens or creates the disk file *(filesystem.dat)* disk file.
- Initializes the block table and reconstructs freeBlockList after a restart.
- Loads inode table from disk

Helper Functions Summary:

Helper Function	Purpose
findEntry(name)	Search inode tale for matching filename
findEntryIndex(name)	Return index of file in inode table or -1 not found
clearBlocks(firstBlock)	Free the entire blockchain when overwriting
getInstance(String, int)	Free the entire blockchain when overwriting
findFreeInodeIndex()	Find empty FEntry
countFreeBlocks()	Count available free blocks
dataOffset(int)	Compute byte offset inside disk for a given block index
writeFEntryToDisk()	Persist file metadata to disk
rebuildFreeBlockList()	Mark which blocks are used and free based on existing files
findFreeBlock()	Return available block index
loadFNodeTable()	Return the singleton FileSystemManager instance
loadInodeTable()	Load metadata for all files from disk

2.2.2. FileServer

The *FileServer* class acts as the communication bridge or middleman between clients and the file system. It always listens through the port 12345 and accepts incoming client connections (eg, can use nc, telnet, …).

For each new connection (every new client):

- The server wraps the socket in *BufferedReader* and *PrintWriter*.
- It creates a *ClientHandler* running in its own thread.
- The handler loops and processes commands until the client disconnects.

Supported commands:

CREATE <filename>

WRITE <filename> <content>

READ <filename>

DELETE <filename>

LIST

QUIT

Our design also provides a welcome header with the instruction format for each command. This menu helps guide the client and ensure command format consistency.

```
Hello and welcome!
Connected to the server at localhost:12345
Response from server: CONNECTED: You are connected to FileServer on port 12345
Response from server: Available commands:
Response from server:   CREATE <filename>
Response from server:   WRITE <filename> <text>
Response from server:   READ <filename>
Response from server:   DELETE <filename>
Response from server:   LIST
Response from server:   QUIT
Response from server: ----------------------------------------
>
```

Figure 5. Single Client Welcome Header

The server parses these commands, validates the input format, and then calls the corresponding method in the *FileSystemManager*. All the processing is built within a loop so that the client can issue multiple commands without connecting.

The error handler is also implemented in this class. If a command fails due to an exception, it sends a clear error message to the client.

2.2.3. FileClient

The *FileClient* is a class that connects to the server at *localhost:12345* and sends a user list of commands. It prints all responses returned by the server so the client can keep track of their request and the results of each operation.

To avoid delays or blocking issues, after each command is sent to the server, the client waits and then reads all available lines from the server using: *reader.ready().*

For multiple-threading tests, we added a *MultiClientMain* that does the following:

- Asks the user how many threads to spawn
- Uses an ExecutorService thread pool to start multiple
- Waits for them to finish and then triggers a clean shutdown

It also has its own *ClientRunnable* class which:

- Implements *Runnable*.
- Connects to the server and runs scripted sequences of CREATE, WRITE, READ, DELETE, LIST or QUIT commands depending on the chosen mode.
- Uses the shared static flag public static volatile boolean DISCONNECT to stop loops when testing is done.

This interaction is faster and avoids a huge delay. Our design also replaces the method to exit the connection between the client and server, from the “enter” action can end the connection by repeating this process in a loop until the user types *QUIT/EXIT.*

2.2.4. FEntry and FNode

FEntry represents the metadata of a single file in the filesystem. It stores a fixed 11-byte filename, the file’s total size in bytes, and a pointer to the first block where the actual content begins. No modifications were made to this class, as it already fulfils the core functionality of tracking per-file metadata.

FNode represents one physical data block in storage and contains two main sections: the block’s index and a pointer to the next block in the file chain. Multiple *FNodes* from a linked list represent the complete content of a file, just like the FAT system. No modifications were made to this class, as it already fulfils the core functionality of supporting the blockchain correctly.

3. Algorithms

This section mainly focuses on the logic behind the file system implementation.

3.1. File Creation Algorithm - fileCreate

The *fileCreate* operation is responsible for creating a new empty file in the filesystem. It only sets up the metadata (an *FEntry*) and does not allocate any data block yet. This keeps the creation fast and only uses the space in the inode table until the file is written.

Summary table:

Step	Action	Description
1	Acquire write lock	Block other readers/writers while modifying the inode table.
2	Validate filename	Check for null/empty/invalid length, also reject bad names early.
3	Check duplicates	Use `findEntryIndex(name)` and if it exists, return "file already exists".
4	Find free inode	Call `findFreeInodeIndex()` none are available, return "no free slot"
5	Initialize `FEntry`	Set `filename`, `filesize = 0`, `firstBlock = -1` for the new file.
6	Persist Metadata	Call `writeFEntryToDisk(index)` to save the new inode into `filesystem.dat`.
7	Release lock & report result	Unlock and return success/failure to the caller.

Since *fileCreate* does not allocate any data blocks, the *freeBlockList* and *FNode[ ]* structures are unchanged by this operation. Actual block allocation is deferred to the write algorithm (*writeCreate*), which keeps file creation as lightweight and efficient as possible.

3.2. File Writing Algorithm - writeCreate

Overwrite the old file content by clearing its previous blocks, writing new data, rebuilding the block chain, and updating metadata.

Summary table:

Step	Action	Description
1	Clear old blocks	If the file already occupies blocks, call clearBlocks() → Frees every block in its old chain and resets next =-1.
2	Compute required blocks	Convert text → bytes: blocksNeeded = ceil(size/128)
3	Check free space	If blocksNeeded>countFreeBlocks(), throw "file too large" → Prevents overflow.
4	Allocate blocks	For each required block: find a free block → mark it used → write up to 128 bytes inside it.
5	Write data chunk-by-chunk	Use disk.seek(dataOffset(newBlock)) and write portions of the byte array in sequential blocks.
6	Build blockchain (FNode)	Link blocks usingprevBlock.setNext(newBlock).
7	Update FEntry	Set filesize, firstBlock in the FEntry → new content
8	Persist metadata	Updated FEntry back to disk: writeFEntryToDisk(index) → ensuring persistence after restart.

3.3. File Reading Algorithm - readFile

Follow the file's FNode chain and read the number of bytes stored in the file.

Summary table:

Step	Action	Description
1	Locate FEntry	Use findEntry(filename). If not found → error. If firstBlock == -1 the file is empty.
2	Prepare buffer	Create a byte array of size filesize → Prevents reading extra unused bytes.
3	Traverse blockchain	Start at firstBlock. For each block: read at most 128 bytes → move to next
4	Stop if all bytes are read	Continue reading until remaining<=0 or next ==-1
5	Return content	Use disk.seek(dataOffset(newBlock)) and write portions of the byte array in sequential blocks.

3.4. File Delete Algorithm - deleteFile

The *deleteFile* operation removes a file from the filesystem. It does two main things, including:

- Freeing all the data blocks that belong to the file
- Clears the corresponding *FEntry* in the inode table so that the filename can be reused.

This operation is used to update both the metadata (*inode* table) and the data structures that track block usage. It is protected by a write lock to keep the thread safe when multiple clients access the filesystem.

Summary table:

Step	Action	Description
1	Acquire write lock	Block other readers/writers during the delete operation.
2	File file inode	Use `findEntryIndex(filename)`: if `-1`, return “file not found”.
3	Check free space	Determine whether the file has a data block chain to free.
4	Allocate blocks	Call `clearBlocks(firstBlock)`to walk the chain and mark blocks as free.
5	Write data chunk-by-chunk	Reset *filename, filesize* = *0*, and `firstBlock = -1.`
6	Build blockchain (FNode)	Call `writeFEntryToDisk(index)` update `filesystem.dat.`
7	Update FEntry	UserebuildFreeBlockList() if a consistent refresh is needed.
8	Persist metadata	Unlock and notify the caller of successful deletion.

By fully freeing the block chain and clearing the *inode* entry, *deleteFile* makes sure that both the storage space and *inode* slots can be reused safely by future files.

3.5. File Delete Algorithm - listFile

The *listFile* operation returns a summary of all the files that are currently stored in the filesystem. It does not touch any of the data blocks, only reading the *inode* table (*FEntry[])* and building a readable list for the client to see.

The algorithm scans each entry in the *FEntry[]* inodeTable. It then selects only valid (non-empty) entries from the block. For each file, the name, size and other information (like the block index) is collected, and thus returns a formatted list as a string to the caller (the FileServer), which then sends it to the client.

To be able to safely support concurrent access, *listFile* uses the read part of the *ReentrantReadWriteLock*, since it is a read-only operation.

Summary table:

Step	Action	Description
1	Acquire a read lock	Allow multiple concurrent readers and blocks if a writer is active.
2	Initialize output/empty flag	Prepare a buffer (*StringBuilder*) and tracks to see if any file is found.
3	Iterate over *iNodeTable*	Scan each *FEntry* from 0 to MAXFILES - 1.
4	Select valid entries	Skip empty/unused entries, by only processing entries with a real filename.
5	Append file info	Adds *filename* and `filesize` (and optionally *firstBlock*) to the output.
6	Handle “no files” case	If no valid entries were found, return *"No files found".*
7	Release lock & return string	Unlocks and returns the final listing to the server/client.

listFile reads the metadata and never modifies it, thereby using the read lock allow the operation to run efficiently whilst still avoiding inconsistent viewing of files, especially if another thread is creating, writing, or deleting files at the same time.

4. Rationale

This section illustrates the reason why we chose this design, how it improves correctness, performance, and clarity compared to alternative options. We focus on the debugging convenience and synchronization correctness.

4.1. FileServerManager

The helpers are organized this way because:

- Each helper does one small job → easy to debug.
- Structure is simple to extend.
- Reduces coding complexity by avoiding one giant method.
- Makes error cases easier to isolate and test.

The *ReadWriteLock* is used instead of the global lock because:

- Multiple reads can happen in parallel → improves performance.
- Writes remain fully protected → avoids race conditions.
- Ensures proper synchronization for create/delete/write.
- Fair lock mode prevents thread starvation under heavy load.

The race condition is reduced because:

- All modification operations use the exclusive write lock.
- Read operations only see stable, fully updated states.
- Delete/write safely, free and reassign blocks without corruption.
- Only one writer at a time

- Implement a system to be able to send an error message to the clients if they are trying to write when the other client is working on it.

Complexity:

- Time complexity: Searches are O(n), but n is small (5 files, 10 blocks).
- Space complexity: Fixed array → O(1)
- ReadWriteLock is more complex than a simple lock, but improves concurrency.
- Still relatively easy to understand because everything is centralized.

Limitations:

- Our design is not very optimized for extremely large files and disks.

4.2. File Server

The reason our design chose to divide *FileServer* and *ClientHandler*

- Accept connections in one and process commands in another → more clear
- Easier to read, easier to debug.
- Each client is handled independently in its own thread.
- Makes error handling much clearer

We decided to add the menu header, because:

- The user immediately knows how to use the system.
- Reduces input errors and retyping mistakes.
- Clients see clear success/error messages.

Implement add backlog + thread pool:

- Prevents refused connections when many clients connect at once.
- Makes the server more stable under heavy load.
- A thread pool is more efficient than creating unlimited threads.
- Greatly improves performance in 1000-client tests.

Complexity:

- Thread pool + ClientHandler is more work to implement, but offers much more stability.
- Time complexity: Each command still runs in *O(1) or O(n)* small-scale operations.
- Thread pool reduces thread creation overhead.
- Backlog reduces rejection rate.

4.3. File Client

Decide to create ClientRunnable + MultiClientMain because:

- Provides a realistic simulation of many independent clients.
- Implement the mode to choose which makes the testing purpose simpler
- Makes concurrency testing easier to observe.
- Cleaner than one loop, creating many raw sockets.
- Improves debug visibility for each client's behaviour.
- Add the auto-generated file name as file#.txt using create if none exist
- Write follows the format "Thread # is writing in this filename"

This design makes debugging easier.

Add reader draining + small delay:

- Ensures the client receives all server messages.
- Prevents output mixing and missing lines.
- Avoids blocking issues where the client waits forever.

Add a delay when quitting many threads:

- Prevents all clients from sending QUIT at the same time.
- Avoids server-side socket errors (“connection reset”).
- Allows the server time to close each handler safely.
- Eliminates most *ClientHandler* error spam during mass disconnect tests.
- Implement the multiple program test can only disconnect when we press ENTER at the end.

Complexity:

- Runnable + thread pool + delay logic is more complex → more reliable
- Time complexity: Each client still sends simple commands → fast.
- Delay avoids race conditions during mass-disconnect.

Limitation:

- Added delays slightly slow down mass testing.
- The text-based interface is simple but not interactive.

5. Scenario

5.1. Apply the lock, thread pool and delay:

No lost connection and quitting concurrently caused an error even with 1000 threads.

From the multiple threading test:

```
MULTIPLE THREADING TEST

How many threads do you want to launch? 1000
Choose a mode:
1 – CREATE
2 – WRITE
3 – READ
4 – DELETE
5 – LIST
6 – QUIT
Enter mode: 2
Press Enter after finish to end the program.

Launching 1000 clients...

[WRITE] Thread 5 sent: "Thread 5 is writing to vy.txt"
[WRITE] Thread 4 sent: "Thread 4 is writing to file5.txt"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
[WRITE] Server -> SUCCESS: File written.
[WRITE] Thread 10 sent: "Thread 10 is writing to vy.txt"
[WRITE] Thread 8 sent: "Thread 8 is writing to file4.txt"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
[WRITE] Server -> SUCCESS: File written.
[WRITE] Thread 1 sent: "Thread 1 is writing to Available commands:"
[WRITE] Thread 3 sent: "Thread 3 is writing to   WRITE <filename> <text>"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
```

Figure 6. Multiple Threading Test

From Server - quitting without error:

```
[Socket[addr=/127.0.0.1,port=49696,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49696,localport=12345]
[Socket[addr=/127.0.0.1,port=49705,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49705,localport=12345]
[Socket[addr=/127.0.0.1,port=49713,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49713,localport=12345]
[Socket[addr=/127.0.0.1,port=49695,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49695,localport=12345]
[Socket[addr=/127.0.0.1,port=49697,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49697,localport=12345]
[Socket[addr=/127.0.0.1,port=49711,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49711,localport=12345]
[Socket[addr=/127.0.0.1,port=49715,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49715,localport=12345]
[Socket[addr=/127.0.0.1,port=49702,localport=12345]] Received: QUIT
[Socket[addr=/127.0.0.1,port=49716,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49716,localport=12345]
Client disconnected: Socket[addr=/127.0.0.1,port=49702,localport=12345]
[Socket[addr=/127.0.0.1,port=49717,localport=12345]] Received: QUIT
[Socket[addr=/127.0.0.1,port=49706,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49717,localport=12345]
Client disconnected: Socket[addr=/127.0.0.1,port=49706,localport=12345]
[Socket[addr=/127.0.0.1,port=49700,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49700,localport=12345]
[Socket[addr=/127.0.0.1,port=49710,localport=12345]] Received: QUIT
Client disconnected: Socket[addr=/127.0.0.1,port=49710,localport=12345]
```

Figure 7. No Errors viewed from Server for MultiClient Server Output

5.2. Test the Race condition with multiple threading and synchronization:

The system sends the message if the threads try to access a file not available.

From Multiple Threading:

```
Another                                                    1/103
5 – LIST
6 – QUIT
Enter mode: 2
Press Enter after finish to end the program.

Launching 1000 clients...

[WRITE] Thread 5 sent: "Thread 5 is writing to vy.txt"
[WRITE] Thread 4 sent: "Thread 4 is writing to file5.txt"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
[WRITE] Server -> SUCCESS: File written.
[WRITE] Thread 10 sent: "Thread 10 is writing to vy.txt"
[WRITE] Thread 8 sent: "Thread 8 is writing to file4.txt"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
[WRITE] Server -> SUCCESS: File written.
[WRITE] Thread 1 sent: "Thread 1 is writing to Available commands:"
[WRITE] Thread 3 sent: "Thread 3 is writing to   WRITE <filename> <text>"
[WRITE] Server -> Another user is writing. Retrying in 3 seconds...
[WRITE] Server -> ERROR: ERROR: file  does not exist
[WRITE] Thread 9 sent: "Thread 9 is writing to file5.txt"
[WRITE] Server -> SUCCESS: File written.
[WRITE] Thread 2 sent: "Thread 2 is writing to   CREATE <filename>"
[WRITE] Server -> ERROR: ERROR: file  does not exist
[WRITE] Thread 0 sent: "Thread 0 is writing to CONNECTED: You are connected to FileServer on port 12345"
[WRITE] Server -> ERROR: ERROR: file CONNECTED: does not exist
[WRITE] Thread 6 sent: "Thread 6 is writing to file0.txt"
```

Figure 8. Multiple threading not available file output message

Conclusion

This project gave us a good understanding of how a basic filesystem works. By implementing our own *inode* table, block structure, and client–server interface, we were able to see how file operations such as CREAT, WRITE, READ, DELETE, and LIST are handled at a lower level. Building the system with fixed-size blocks, a linked-list allocation method, and clear metadata structures made it easier to test, debug, and reason about the filesystem's behavior.

The use of a *FileSystemManager* as the central component, combined with proper synchronization through a read–write lock, helped us maintain correctness even when multiple clients accessed the server at the same time. Separating the server and client logic also made the system cleaner and work better with concurrency.

Overall, the project strengthened our understanding of file allocation, synchronization, and the importance of organized system design. It also showed how small design decisions such as storing metadata or handling threads, can significantly affect reliability and clarity in a multi-client environment.