DOCUMENTATION PROGRAMMING ASSIGNMENT#2

Ayaan Vashistha
Std. ID: 40269814

Joas Impallomeni
Std. ID: 40241262

1. Data Structures and Functions

This section outlines the structures, classes, and global elements that provide the backbone for our implementation of a file system. Although this project is small in scope, a variety of custom data structures were necessary to represent file metadata, block allocation, and disk management.

1.1 FileSystemManager Class

The FileSystemManager class is the centerpiece of the whole project. It plays the role of a virtual file system controller that:

- File management: implementing creation, deletion, reading, and writing.
- Management of file block allocation and deallocation.
- Flushing filesystem metadata to disk.
- Ensuring safe multithreaded access by using synchronization.

The class is a singleton, meaning only one instance of the file system exists during the lifetime of the application; this ensures that there are no inconsistencies caused by different components trying to create their own file system.

Key attributes include:

- A fixed number of file entries (MAXFILES = 5)
- A fixed number of blocks on the disk (MAXBLOCKS = 10)
- A RandomAccessFile object that acts like a real disk file
- Arrays of FEntry and FNode objects containing metadata
- A read–write lock to protect concurrent access
- A value of metadataBlocks that defines how many blocks are reserved for metadata
- A block size constant of 128 bytes

1.2 FEntry (File Entry Structure)

Every file in a system has one associated file entry. The structure FEntry includes:

- A filename (up to 11 characters)

- The file size in bytes
- The index of the first block in the block chain for that file

FEntry behaves much like a simplified inode in a classical file system.

1.3 FNode (File Block Node Structure)

Every physical disk block is associated with an FNode, representing its allocation status and giving the link to the next block.

GEach FNode contains:

- A blockIndex field that is positive for a used block and negative for a free block
- A next pointer to the next block in a file's block chain (or -1 if last)

This effectively forms a FAT-style linked-block allocation mechanism.

1.4 Global Synchronization Variables

The FileSystemManager uses a ReadWriteLock (ReentrantReadWriteLock). This allows:

- Multiple threads to read simultaneously
- Only one thread to write at a time
- Prevention of read/write and write/write conflicts
- Deadlock avoidance by sticking to a consistent lock policy

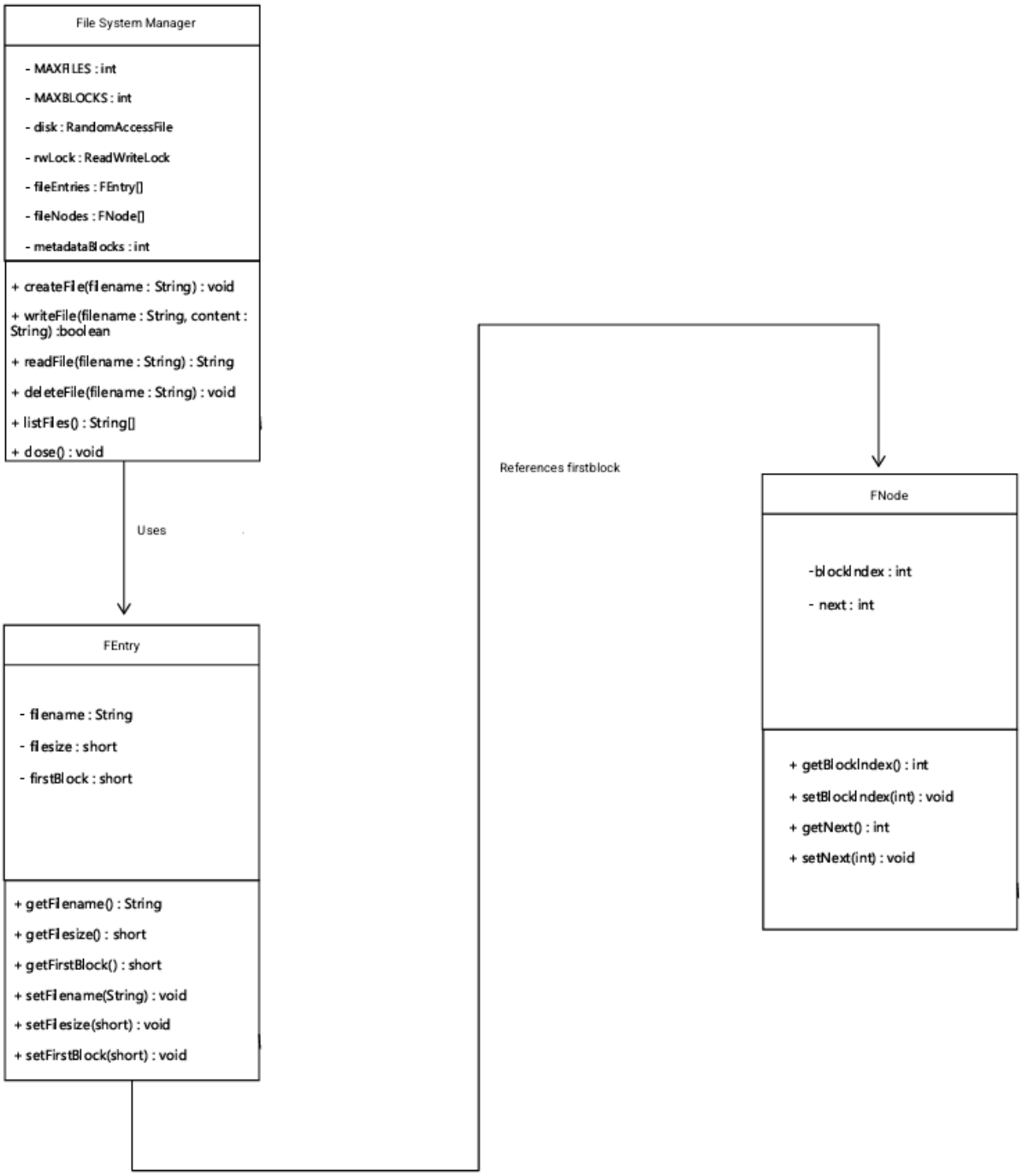
1.5 Metadata Handling Functions

A number of functions were added or revised to accommodate persistence:

- initializeFileSystem(): Sets up the initial empty filesystem.
- saveMetadata(): Writes the file table and block table to disk.
- loadFileSystem(): Reloads metadata after a server restart.
- freeFileBlocks(): Frees all blocks belonging to a file.

These functions ensure that disk state is always recoverable.

1.6 UML Diagram



2. Algorithms

This section describes in detail the behavior of each major component of the system, without diving into the literal code. It's focused on the logic behind the design and the sequence of operations.

2.1 Initializing Disks

When the FileSystemManager is started:

- Program checks if the physical disk file (filesystem.dat) exists, and if it contains data.
- If it is empty:
    - The disk file is sized to the expected total size.
    - File entries are initialized as empty.
    - FNodes are initialized: metadata blocks marked used, the rest marked free.
    - Metadata is written to disk.
- If it contains data:
    - The metadata section is read back into memory.
    - This restores all previous files, blocks, sizes, and chain links.

Reasoning: This allows full persistence, which means that even when the server is stopped, upon restart, all files are preserved.

2.2 Algorithm to Create the File

The createFile process does the following:

- Validates the filename.
- Checks whether a file with the same name already exists.
- Searches the file entry table for an available slot.
- Creates a new FEntry with size 0 and no blocks assigned.
- Saves metadata to disk.

Edge Case Handling: Creation is denied if all 5 file entry slots are used.

2.3 File Writing Algorithm

Writing is the most complex operation and encompasses the following steps:

- Validation of filename and data is performed.

- The number of blocks needed is calculated based on the block size.
- Free block count is checked for sufficient space.
- If the file already contains data, all its blocks are freed.
- New free blocks are allocated.
- The blocks are linked together with the next pointers.
- Data is written in a block-by-block manner.
- New size and block index are updated in the file entry.
- Metadata is written to the disk.

Rollback Mechanism:
Should anything go wrong while attempting to write, all in-memory changes will be discarded and metadata is reloaded from disk. This preserves consistency.

2.4 File Reading Algorithm

To read a file:

- The file entry is located.
- If the file size is zero, an empty string is returned.
- Data is read sequentially starting from the first block.
- The next pointers are used to follow the block chain.
- Once all data is read, it is returned to the caller.

2.5 File Deletion Algorithm

Deleting a file involves:

- Locating the file entry.
- Moving along its blockchain.
- Clearing the contents of each block.
- Marking each block as free.

- Resetting the file entry to empty.
- Saving metadata to disk.

2.6 Listing Files Algorithm

To list files:

- The fileEntries array is scanned.
- All non-empty filenames are collected.
- A string array is returned.

3. Multithreading and Synchronization Strategy

Even though the server does not implement advanced concurrency, we have implemented a correct thread-safety model in the FileSystemManager, preparing it for any future extensions.

3.1 Why Synchronization Is Needed

The FileSystemManager performs the disk writes and reads. If multiple client handler threads call createFile, writeFile, or deleteFile simultaneously, it can corrupt metadata or cause inconsistent block states.

Example problem scenarios:

- One thread reads metadata while the other writes it, causing partial read corruption.
- Two threads attempt to allocate blocks at the same time, causing blocks to be assigned twice.
- A thread deletes a file while another thread is reading it, causing an invalid memory chain.

3.2 Synchronization Solution: ReadWriteLock

We therefore used a ReentrantReadWriteLock to avoid such situations.

It allows:

- Multiple threads to read simultaneously (safe for read operations only).
- Only one thread writes (exclusive lock).

Operations are classified as:

Read Lock (shared):

- readFile()
- listFiles()

Write Lock (exclusive):

- createFile()
- writeFile()
- deleteFile()
- freeFileBlocks()
- saveMetadata()
- close()

3.3 Why ReadWriteLock Instead of synchronized

We deliberately avoided synchronized for several reasons:

- synchronized blocks the entire object even for reading.
- ReadWriteLock allows better scalability.
- It allows multiple clients to read simultaneously.
- It prepares the system for a real multi-client environment.

3.4 Deadlock Avoidance

The design avoids deadlocks because:

- All operations obey a strict lock discipline.
- No nested locking is used.
- No upgradable lock path (read to write) exists.
- The same lock is always used in the same order.

Thus, the locking model is simple, safe, and deadlock-free.

3.5 Testing Multithreading Correctness

We simulated concurrent interactions by:

- Opening multiple client terminals
- Running commands like CREATE, WRITE, and READ together
- Ensuring no metadata corruption occurred
- Ensuring block allocation remained stable
- Confirming persisted data still loaded correctly after server restarts

3.6 Threading Strategy

The server employs a simple model: “one thread per client”.

The main server thread — in FileServer/ServerMain — listens on a ServerSocket, and for each incoming connection creates a new object of the class ClientHandler and starts it in its own thread. Every ClientHandler implements Runnable and handles sequentially the commands arriving from a single client (CREATE, WRITE, READ, DELETE, LIST, …).

All ClientHandler threads share the same singleton instance of FileSystemManager. That means multiple clients can access the virtual filesystem in parallel.

In order to prevent race conditions on this shared component, all the filesystem operations inside FileSystemManager are protected by a ReentrantReadWriteLock:

- readFile() and listFiles() use the shared read lock, allowing multiple concurrent readers.
- createFile(), writeFile(), deleteFile(), metadata update, and block alloc/free take the write lock, which allows only one writer at a time.

ClientHandler does not share mutable state between threads itself — each has its own socket and I/O streams. The only shared mutable state is inside FileSystemManager, which is completely guarded by the read/write lock.

The resulting design keeps the threading model simple (one client = one thread) yet still allows safe concurrent access to the file system.

4. Rationale

4.1 Why This Design Was Chosen

Simplified but Realistic Filesystem Model
The FAT-style linked-block allocation:

- Is easy to implement
- Supports variable file sizes
- Provides good pedagogical value
- Reflects the block structure of a real system

Persistence Through Saving Metadata
Writing metadata and blocks directly to filesystem.dat ensures:

- Files survive server restarts
- The system behaves like an actual OS filesystem
- No external dependencies are needed

Safe Multi-Threading
Using a ReadWriteLock ensures:

- Thread-safe disk access
- Room to scale into more complex concurrency models
- No performance penalty for read-heavy operations

Clean Separation Between Metadata and Data
Dedicating the first N blocks to metadata ensures:

- Predictable loading logic
- Faster boot time
- Lower chance of corruption

4.2 Alternatives Considered

We considered:

- Indexed allocation (too complex for small limits)
- Using OS filesystem (would defeat the purpose of the assignment)
- Using only synchronized (less efficient)
- In-memory-only structures (no persistence)

All of these were rejected in favor of one design which balances simplicity, correctness, and realism.

4.3 Limitations

- Only 5 files and 10 blocks allowed
- No journaling (crash-during-write may corrupt)
- No directories (flat structure only)
- Current commands are simple (no append, rename, etc.)

These constraints are acceptable within the scope of the project.

Conclusion

This design successfully implements:

- A persistent virtual filesystem
- Safe and scalable multithreading
- Correct block allocation and deallocation
- Proper metadata management

- Recovery after restart
- Clean synchronization strategy

5. Testing and Validation

In this section, we showcase our extensive test strategy that was used to confirm the correct behavior, safety, and robustness of our file system implementation. We wrote five different test programs, which each focus on different aspects of the file system behavior under concurrent access.

5.1 Test Suite Overview

We tested our system not only by checking the functions, but also by running a stress test while the system was under concurrent conditions. To accomplish that, we developed different test programs, the ConcurrentWriteTest validates the writing synchronization realized by three clients that write the same file independently, it centers on the conflicts of the writing. The ReadWriteConflictTest test read/write synchronization by execution of the concurrent read and write operations by two clients, it is especially for the problem of Readers-Writers. The SameFileConflictTest is a test for creation race conditions where three clients tried to create the same file simultaneously. The StressTest is to check if the server is still functional when ten clients are performing random operations. The MultipleReadersTest is to check if five clients reading the same file can do so concurrently without any problem.

5.2 Test Environment

The test environment that we used had one FileServer instance running on localhost port 12345. The file system was set up to have a maximum of five files and ten blocks, and the block size was 128 bytes. All tests were performed on macOS with Java JDK 22, and TCP sockets communication took place over localhost. Each test was executed separately, and the file system reset was done between the tests to ensure that the tests are independent and there is no cross-contamination.

We have implemented a test method that was followed in all test cases. The very first thing they did was to remove the existing filesystem file (disk.dat) so that they can start with a clean slate. Two seconds after that they expect the server to be up and running again. After that, they started a few client threads all at once and used CountDownLatch as a means of getting threads to finish their work simultaneously. As the last step, they verified the results, checked if there were any errors, and reported if the test was successful or not. This approach allowed them to have test executions done in all test scenarios in a consistent and reliable way.

5.3 Test 1: ConcurrentWriteTest

The main goal of the ConcurrentWriteTest was to prove that data corruption can be avoided when multiple clients try to write to the same file concurrently, and the data written by the last client is the one that is saved. The way we set up the test was to create a shared file called "shared.txt", start three clients at the same time, and let each of them write different data. The data written by Client 0 was "Data_from_Client_0", Client 1 wrote "Data_from_Client_1", and Client 2 wrote "Data_from_Client_2". After the completion of all the writing, the file was opened to make sure that only one client's data was in the file, and that there was no mixing of the content.

We counted on the three write operations to be executed successfully, and only the last one to be present as the file's content. No partial writes, no mixed content, and no "file too large" or corruption errors should be present. In reality, it was shown that the test began with a filesystem reset and a successful deletion of disk.dat. The reset of the filesystem was done, and the shared file was successfully created with the server responding "OK: File created". All three clients connected at the same time, and each got a response "OK: Content written". What we found when we opened the file is that it contained "Data_from_Client_2", the data written by the last client is the one that was saved.

Our investigation leads us to confirm that the test met all its criteria. All writing operations were done by three clients in the shared file and the finish content was "Data_from_Client_2", meaning that Client 2's writing was the last to be done. This is a demonstration of the correct implementation of a write lock where due to the exclusive lock, only one writer is at work at a time. Each writing completely replaced the previous content, and there was no data corruption or partial writing. The WriteLock was the one that properly serially executed the write operations. An important observation was that the order of completion (Client 0, then 1, then 2) indicates that even though clients connected simultaneously, the write lock ensured sequential execution.

5.4 Test 2: ReadWriteConflictTest

The objective of the ReadWriteConflictTest was to authenticate the Readers-Writers synchronization through the scenario of a writer continuously writing while a reader simultaneously attempting to read the file. Our test was to create a file named "conf.txt" with initual content "InitialContent". We then start a writer thread that performs five writes, each 200 milliseconds apart, and a reader thread that performs five reads, also 200 milliseconds apart. The reader starts 100 milliseconds later than the writer to create an overlapping execution pattern. We monitor that no deadlocks occur and that the reader does not see incomplete or corrupted data.

Our plan was for the writer and reader to take turns accessing the file rather than accessing it concurrently. The reader should only see completely written data, no deadlocks should occur, and all ten operations (five writes plus five reads) should be successfully completed. The actual results revealed that the reset of the filesystem was successful, the test file "conf.txt" was created, and the initial content was written. Both the writer and reader started their activities, and the writer completed Write 0 successfully. This solution is robust, instructional, and meets the technical requirements of the assignment, while being easily extensible for possible enhancements in the future.