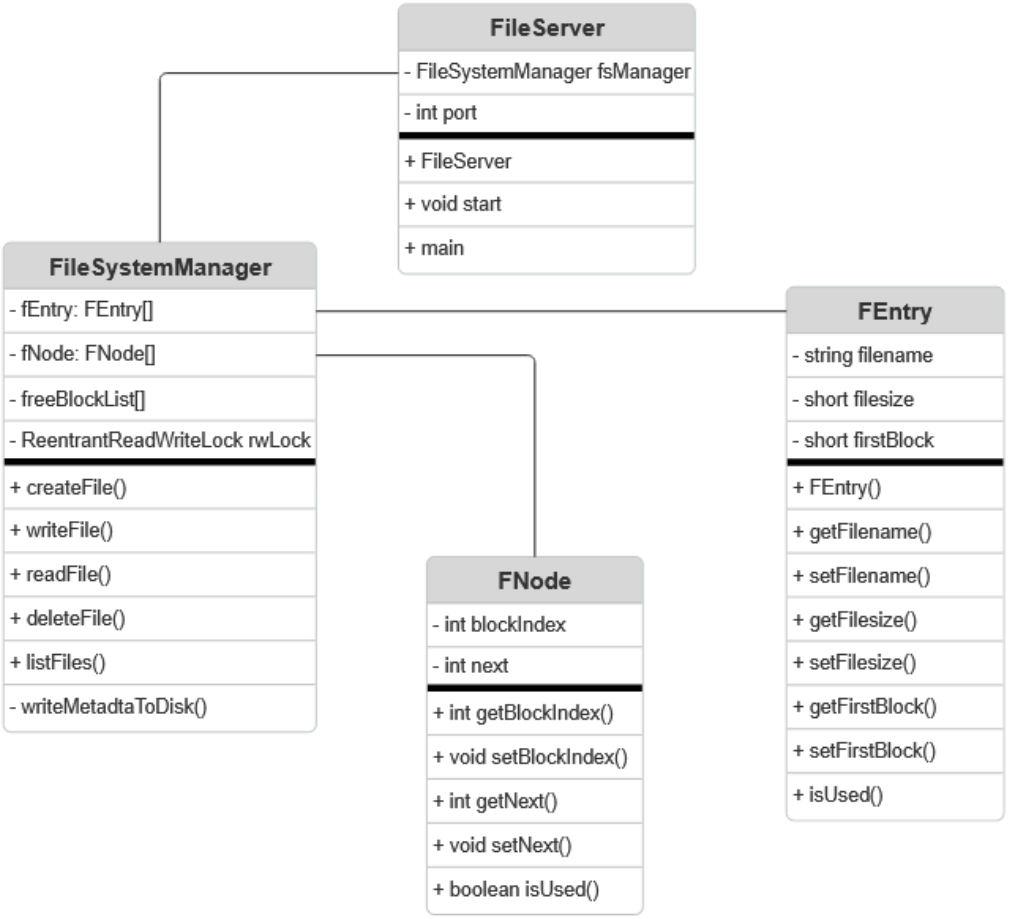
Design Report

Matthew Lunt 40137949
Riad Rakan 40264879

1- Data Structures and Functions



The following UML diagram consists for four main classes that we modified:

- FileSystemManager manages file operations, metadata, block allocation, and disk allocations,
- FEntry represents each file metadata such as filename, filesize, and first data block,
- FNode represents a node in the block's linked list in which file contents are stored,
- FileServer listens to user connection, and launches handles threads,
- ClientHandler handles the commands and interacts with the FileSystemManager,
- And reentrantReadWriteLock provides concurrency control allowing multiple user readers, and one writer.

2- Algorithms

2.1 File Creation Algorithm

The file creation algorithm is split up into five main steps.

1. Ensure the file name length is within the provided limit of <= 11 characters.
2. Check if the file name is already in use.
3. Locate the first unused FEntry.
4. Initialize the entry through size and location.
5. Link this meta data with the disk by using a write lock to ensure mutual exclusion.

2.2 File Writing Algorithm

The file writing algorithm is composed of eight steps to ensure there is mutual exclusion and optimized memory storage. Only one writing algorithm can be run at once.

1. Calculate the number of blocks required for the new data.
2. Count the free blocks, and the ones currently owned by the file.
3. Check both values to ensure it is within the requirements, otherwise abort.
4. Free the file blocks and overwrite them with zeros.
5. Allocate the new free blocks and write the data sequentially.
6. Update the FEntry which contains the filesize and the first block location.
7. Move the metadata to disk.
8. Ensure all critical steps are done within a lock for mutual exclusion.

2.3 File Reading Algorithm

The file reading algorithm can be run by multiple users at the same time since it has no critical region in the memory. The file reading algorithm is one of the simple algorithms and is composed of only four steps.

1. Located the FEntry that matches the file name.
2. Use Fnode.next to traverse the file blocks sequentially.
3. Ready only filesize bytes.
4. Extract all data and return.

2.4 File Deletion Algorithm

The file deletion algorithm is complex and requires a write lock. It is split into seven main steps.

1. Located the FEntry file using the file name.
2. Traverse all the blocks in the file.
3. Overwrite each file block with zeros.
4. Mark each block as free.
5. Reset the FEntry.
6. Save the metadata.
7. All should be done under the protection of a write lock.

2.5 List Operation

The list operation scans FEntry and returns all the file names it contains, excluding the empty containers. This requires only a read lock.

2.6 Multithreaded Design

This server uses multithreading to allow multiple users to connect. Moreover, a multithreaded design ensures continuity, and proper error management between user and server. There are four steps that produce this:

1. ServerSocket.accept() waits for a client to connect through a given port.
2. Once a user is connected a new ClientHandler thread is created (done for each connection).
3. The ClientHandler takes the commands and calls to the FileSystemManager functions.
4. Any errors are returned to the user, but the server stays running.

2.7 Synchronization Strategy
A ReentrantReadWriteLock enforces the readers–writer rules:
• Unlimited concurrent readers.
• Only one writer at a time.
• Readers cannot run while a writer is active.
• Prevents data races and ensures file system consistency.
• Deadlock-free because only one lock object is used, with fixed acquisition order.

3- Rationale

One of the main changes we have made to the original code was using a reentrant read write lock rather than just the reentrant global lock. We decided to do this because it would be much more efficient and in line with the requirements of having multiple reads and exclusive writes. We believe this will be much more efficient and benefit the concurrency model. We also isolated sections within methods that used a right so that only the critical section of writing would be blocked with the right lock and the setup for it would only be blocked with a read lock. This can be seen in the write file method where a read lock is used during the calculation of space when checking if the amount of free blocks and blocks to be overwritten will be enough, which is an expensive method,and then changing to a write lock when it begins to edit the storage.

We found that making use of this system would be better than the global lock as it would allow more processes to run at the same time without delay as the global lock would shut down all access to memory whether it be read or write slowing down the system so it would not be ideal.This significantly increases overall system throughput and reduces unnecessary blocking. Reads such as LIST and READ are the most commonly used operations, allowing them to run simultaneously improves responsiveness for all users. By using a single read/write lock across the entire filesystem, we eliminate ordering problems entirely. The result is a simple, clean, and deadlock-free design that still supports concurrency. Since metadata and block storage are updated under the same lock, clients won't be able to see a half-written state. This eliminates file corruption issues such as ghost files, orphaned block chains, or readers retrieving inconsistent data. Every update is atomic from the perspective of other threads.

From a testing and correctness standpoint, our design allows clear validation scenarios such as concurrent read stress tests, read/write interleaving tests, and deletion during read. Having all writes pass through a single write lock ensures that even under heavy concurrency, correctness should never be compromised. The predictability of locking behavior makes fault injection tests and race-condition detection much easier.

The overall time complexity of operations remains efficient: metadata lookup is O(1), while block traversal for read/write/delete is O(n), where n is the number of blocks in the file. Because read operations are non-blocking with respect to other readers, scalability improves significantly as clients increase. A global lock would have serialized all operations regardless of type, creating delays and reducing usability.

Overall, this design provides a strong balance between simplicity, safety, and concurrency. It is straightforward to conceptualize, avoids unnecessary locking complexity, and is extensible for new features. Most importantly, it guarantees correctness while still providing efficient multi-client performance.