Programming Assignment 2 - Design Report

COEN 346

Operating Systems

Section: F

Zineb Bamouh - 40263096

Ileass Bouhout - 40263288

Date Due: November 16, 2025

"I certify that this submission is my original work and meets the Faculty's Expectations of Originality"

Ileass Bouhout and Zineb Bamouh

November 16 2025

Data Structures and Functions:

Data Structures:

FNode:

Represents a data block in the file storage. Each contains a **blockIndex** and a **next** pointer to the next FNode in a file's chain. All FNodes are stored in an array and each node's **next** is set to -1 initially. Getter and Setter methods were created for FNode. Table 1 shows the getter and setter methods for FNode along with their arguments and return values.

Table 1: Getter and Setter functions for FNode.

Method	Argument type	Return Value
getBlockIndex( )	-	int
setNext(next)	int	-
getNext( )	-	int

FEntry:

Represents an "inode" for each file and stores the file's metadata: the filename, the file size, and an index of the first data block. Getter and Setter methods were created to get and set the relevant information for each file. Table 1 shows the getter and setter methods for FEntry along with their arguments and return values.

Table 2: Getter and Setter functions for FEntry.

Method	Argument type	Return Value
getFilename( )	-	String
setFilename(filename)	String	-
getFilesize( )	-	short
setFilesize(filesize)	short	-
getFirstBlock( )	-	short
setFirstBlock(firstBlock)	short	-

FileSystemManager:

Data Structures:

inodeTable: Serves as the single directory of the system. It is an array of FEntry objects of size MAXFILES (5 in this case). Each index holds a file's metadata or **null** if it is free.

fnodeTable: An array of FNode objects of size MAXBLOCKS (10 in this case). Each index represents a data block on the disk. The FNode at this index holds the info necessary for linking that block of memory.

freeBlockList: A boolean array of length MAXBLOCKS, tracking free and used blocks, working as a bitmap. A value of **true** means that the block is free, while **false** means the block is allocated.

rwLock: A ReentrantReadWriteLock object which enables synchronization of the filesystem by allocating locks to threads when performing operations on the files and the disk.

disk: A RandomAccessFile object which takes care of the persistence of the filesystem data after the server is closed. It holds the FEntry

Functions:

createFile(String filename): Takes in a String argument and creates a file in the disk, by using the argument as the file's name.

writeFile(String filename, byte[] content): Takes in a String **filename** and a byte array, **content**, to write data into the file that has the same name as the **filename** argument.

readFile(String filename): Takes in a String argument and reads the contents of the file in the disk that is named **filename**, and returns an array of bytes such that the contents can be displayed to the user.

deleteFile(String filename): Takes in a String argument and deletes the file named **filename**, and sets its contents to 0 to avoid malicious use of data.

listFiles( ): Lists the names of all the files currently on the disk.

saveMetadata( ): Writes out the contents of inodeTable, fnodeTable and freeBlockList into the disk such that it persists even when the server is closed and reopened.

loadMetadata( ): Reads the data back into memory when the server starts up.

freeFileBlocks(int firstBlock): Frees up all the blocks used by a file, by starting from block **firstBlock**.

writeFixedString(String s, int length): Ensures the filename fits exactly 11 bytes when saved to avoid metadata to be misaligned and potentially make loadMetadata read garbage values.

readFixedString(int length): Reconstructs the 11 byte strings back into Java. Along with writeFixedString, it ensures that persistence exists across restarts.

freshFileSystem(): Clears the inodeTable, it marks all blocks as free except metadata, resets every FNode in fnodeTable and write into disk using saveMetadata. It enables a new filesystem to be initialized if there isn't an instance yet created.

FileServer

fsManager: The fileServer holds a FileSystemManager object so that it can call File System methods for command execution.

port: A variable that holds the localhost port for the ServerSocket that client with communicate through.

start() method: Handles the Server initialization and client socket allocation for concurrent access to the File Server.

clientHandler() method: Handles the client inputs and parses them into arrays to enable operations on the File Server.

The UML file shown in Figure 1 shows the relationship between classes used in this assignment along with the attributes and methods for each of them.

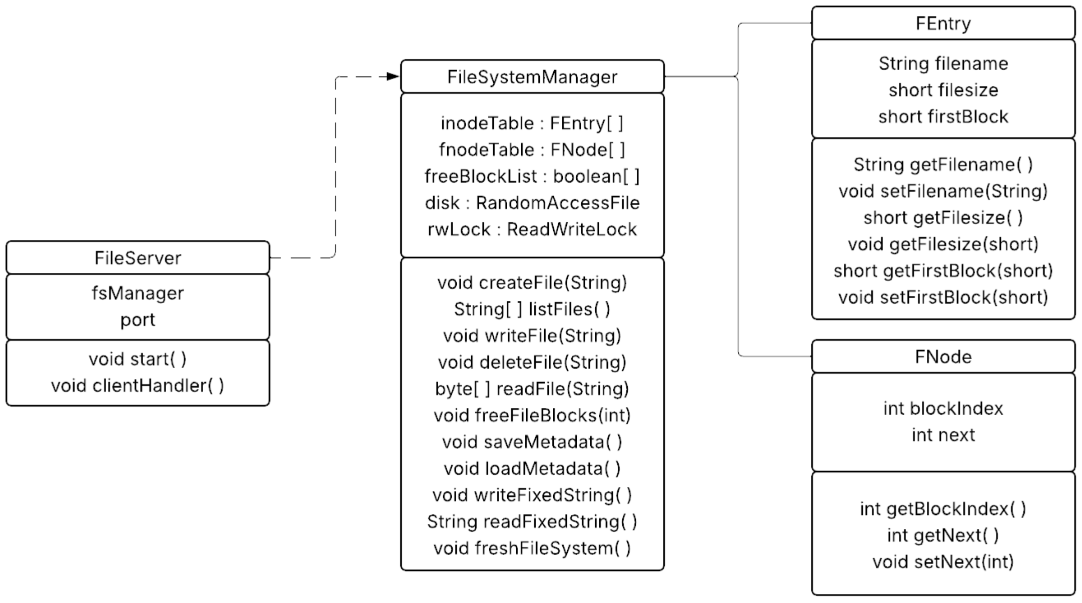

Figure 1: UML Diagram for File System

Algorithms:

FileSystemManager:

createFile(String filename): It acquires a write lock and tries to create a new file entry by using a String argument **filename**. The inodeTable is studied to determine if an existing file already has the same name, in which case no file is created and a message is sent to the user specifying that it already exists. Next, the method finds the first empty slot in the inodeTable; if no entry is available in the inodeTable, an exception is thrown stating that the file limit was reached. Next, a new FEntry is created with **filename**, a size of 0 and **firstBlock** = -1, before being saved in the free slot. The metadata is finally stored in the disk to save the change.

writeFile(String filename, byte[] content): It acquires a write lock before finding the FEntry of the target file in the inodeTable. It then calculates the the number of blocks necessary to store the new content to write. It scans freeBlockList to see if enough blocks are available for storing **content**. If there is not enough space available, the method stops and throws an exception stating that there is not enough space available. If enough space is available, writeFile frees up the blocks currently used by the file by calling freeFileBlock( ). The new content is then written at the correct position in the RandomAccessFile and marking the block as "used" in freeBlockList[]. The method updates the FNode at that index such that it points to the next block in the sequence. The FEntry size is also updated by using **content**'s length and **firstBlock** is updated to the index of the first allocated block. The metadata is stored in the disk to save the change.

readFile(String filename): It acquires a read lock to allow concurrent users to read a file's contents. The FEntry is found by name, and throws an exception if it does not exist. Reading an empty file returns an empty array of bytes. Otherwise, it traverses the linked list of FNodes starting from the file's firstBlock by using the **next** pointers of FNodes until it reaches -1; signifying the end of the file. By traversing the linked list, it appends each of them to a buffer. The array of bytes is returned at the end.

deleteFile(String filename): It acquires a write lock to delete a specified file if it exists. The inodeTable is scanned to find the file which holds the **filename**. If found, the file's first block index, **firstBlock**, is used to invoke freeFileBlocks(**firstBlock**) to release all the blocks used for the file. The corresponding FEntry in the inodeTable is set to **null.** The file's metadata on disk is overwritten by calling saveMetadata( ). Additionally, the file data is zero-filled.

listFiles( ): It acquires a read lock to allow concurrent users to see the filenames currently stored. The method searches for non-null entries in the inodeTable and appends them into a String array. The array is returned to the caller to be displayed. If no files are found an empty array is returned.

Utility Methods:

- freeFileBlocks(int startBlock): It iteratively frees up a data block chain starting from the given index; **startBlock**. It reads the **next** pointer of each block in the fnodeTable to traverse the list.

- saveMetadata( ): Seeks to the beginning of the disk file and writes the the contents of inodeTable, fnodeTable and freeBlockList. It writes a 11-byte filename and two short values, fileSize and firstBlock, for each FEntry in inodeTable.

- loadMetadata( ): Reads the data in the disk file back into memory when the server starts up. Each non-empty FEntry slot is reconstructed by using **next** pointers of each FNode according to the stored values.

- writeFixedString(String s, int length): It converts the string into a byte array and truncates or pads it to fit the length required. It is then written to the metadata area by using **disk**. It ensures that each FEntry uses a predictable space in the metadata block, which is important to make sure offset calculations are correct when calling loadMetadata( ).

- readFixedString(int length): It reads a fixed number of bytes, **length**, from the metadata region and constructs a String from them. It then truncates unnecessary bytes (null or padding) to get the filename back.

- freshFileSystem( ): It creates an empty filesystem in memory and saves it to **disk**. It clears all data structures in memory; the inodeTable, the fnodeTable and sets all data blocks as free in the freeBlockList. Every FNode's **next** pointer is set to -1. After creating this fresh layout in memory, it is serialized in the **disk** by using saveMetadata( ). It is only called when the persistent metadata is absent or invalid.

Synchronization:

A ReentrantReadWritelock (rwLock) is used to enable synchronization between threads. It enables threads that perform write operations (create, write, delete) to acquire a write lock, preventing other threads from writing into the same file or deleting a file that is being written into, for example. Read locks are acquired by threads that perform read operations, like readFile and listFiles, enabling other threads to perform these read operations concurrently.

Persistent Storage:

Java's RandomAccessFile provides low-level direct access to a file the same way as an actual disk would. Instead of only being able to write sequentially, RandomAccessibleFile makes it possible to seek any position in the file and read or write bytes at that exact offset. This is needed for implementing a block-based file system because blocks need to be read or written independently without having to modify the whole file. Also, it allows to serialize structures like the inode table or FNode table.

FileServer

Multithreading:

- start( ) method: It creates the listening socket through which client sockets will communicate to send fileServer requests. A while loop is used such that the server socket waits for clients to connect. When a client accesses port 12345, the server socket starts handling the client socket and creates a virtual thread for it. The server socket stays busy-waiting for new clients to access the port. Each client is then handled by clientHandler( ).
- clientHandler( ) method: It sets up I/O wrappers that are necessary to handle client requests. A while loop is used to keep the handler in a busy-wait state while it waits for a client to send a command. The command is parsed for the handler and a switch case is used to determine which operation to do. Each case contains a try-catch to determine if the right number of elements were parsed to fulfill each command. For example, if the create command line contains less than 2 elements, it must mean that no filename was specified by the user. The FileSystemManager's method that corresponds to each case is then called to execute the intended command. Error handling is used to check if client handling fails or if commands do not have the right number of parsed elements.

Rationale

For the server design, we chose virtual threads because they offer a more scalable and practical concurrency model compared to other models. For example, the thread-per-client method which relies on OS threads, requires a large stack and becomes subject to a higher amount of context switch. Thread-per-client becomes less sustainable as the number of clients grows. Or also, the thread pool can become saturated and clients must wait in queue which also is not sustainable in scalable situations. This is precisely where virtual threads become relevant. Virtual threads are lightweight, managed by the JVM instead of the operating systems, and can be very scalable without exhausting the resources. This makes it possible to follow the one thread per client model without worrying about thread starvation. Another advantage is that it preserves a blocking programming model, while the JVM manages the thread efficiently by performing a blocking I/O operation. The JVM temporarily blocks that virtual thread but frees the other threads to serve other tasks. This results in excellent utilization of CPU time with minimal complexity. Although we considered using a thread pool or the newVirtualThreadPerTaskExecutor(), directly generating a virtual thread for each incoming connection turned out to be the simplest and most scalable solution for this assignment.

The FNode class is implemented like a linked-list, where each node points to the node that follows it; making a chain of nodes that holds the content of a file. Using a linked-list makes it so the data can be stored non-contiguously in the disk. An alternative design would have been to store file data in contiguous blocks. However, this method comes with major limitations, it requires large consecutives free regions that make file growth more difficult. The linked-list fits better the small and dynamic nature of the assignment.

Implementing the freeBlockList as a boolean array provides the simplest and most efficient way to track which blocks are free. Each index corresponds directly to a block number so checking/updating availability has a time complexity of O(1) which makes it fast and predictable.

The inodeTable is a fixed size array because the maximum filesystem's maximum count is known in advance. Using a fixed array guarantees predictable memory usage.

For a small storage like the one used in this assignment, we chose to go with a first-fit memory allocation strategy. First-fit is ideal in this case since the storage space is limited and fast to traverse, enabling rapid allocation. An alternative that was considered was best-fit but we found that the overhead it would create would outweigh the advantages of this strategy, thus making first-fit the better option.

For Dynamic File System *Size*, the current design uses fixed-size structures (MAXFILES and MAXBLOCKS are compile-time constants). This limits the number of files and total storage. A possible improvement would be to allow the file system to grow dynamically. For example, by increasing the backing file size and extending the freeBlockList and fnodeTable when needed, or by implementing multiple "extent" files. Even without dynamic resizing, simply choosing larger

Our filesystem uses a fixed-layout metadata serialization scheme, which stores the inode table, FNode table, and free-block bitmap in a predictable, predefined order. This makes saving and loading the filesystem state straightforward and minimizes the risk of corruption caused by partial writes or inconsistent formats.

An important design decision was to reserve the first two blocks of the disk exclusively for metadata. During testing, we noticed that only the first file stored in the filesystem would occasionally contain corrupted characters, even though the write logic itself was correct. The corruption appeared sometimes, especially after restarting the server or performing multiple writes. This led us to investigate the metadata layout, and we discovered that the serialized metadata occupied 165 bytes which is more than the 128 bytes available in a single block. As a result, the metadata overflowed into block 1, which the filesystem incorrectly treated as a data block. Any file whose first block was allocated at index 1 would therefore have part of its content overwritten every time the metadata was saved. A possible fix would be to increase BLOCK_SIZE to 256 so that all metadata could fit in a single block. However, it would also make the system less flexible and more fragile if metadata grows in the future. Instead, we adopted a more scalable solution: reserving blocks 0 and 1 exclusively for metadata, and starting file data allocation at block 2. This preserves the original block size and avoids structural changes

When files are deleted, all associated blocks are explicitly zeroed out. This prevents old data from leaking into newly allocated files and ensures that recovered free blocks start from a clean, uniform state.