Design Report of Programming Assignment #2: A File Sharing Server

Operating Systems, COEN 346

Presented by

Jenny Tan 40281924,

Jing-Ning Chen 40281215

Submitted to

Professor Paula Lago

“We certify that this submission is our original work and meets the

Faculty’s Expectations of Originality”

Jenny Tan, Jing-Ning Chen

November 17th, 2025

DESIGN REPORT

This report regards the design methodology of our File Sharing Server, which includes our File System Manager, File Server, Multithreading and Synchronization tools. This report is separated into 3 sections: Data structures and functions, Algorithms and Rationale.

Section 1: Data structures and functions

1.1 Server and File System UML Diagram

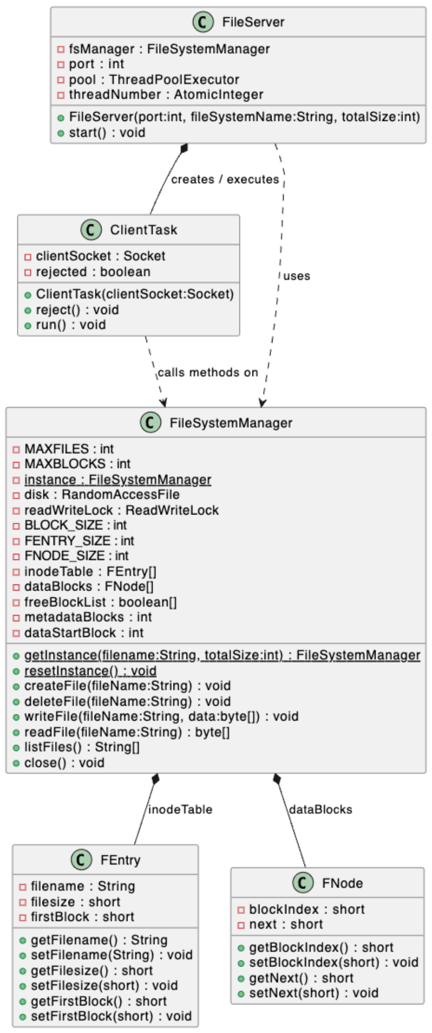

Figure 1: UML Diagram

1.2 File Server:
Table 1 provide a summary of the modifications done on the FileServer file. Only the one we found relevant are mentioned.

Table 1: Summary of Modifications for FileServer

Modification	Why this modification?
ThreadPoolExecutor pool	To serve multiple clients concurrently, allowing multithreading. It limits the number of active threads to prevent resource exhaustion and reduces the overhead of creating a new thread for every client connection.
AtomicInteger threadNumber	To safely generate unique sequential IDs for worker threads. This is used by the ThreadFactory to provide meaningful names which are helpful for debugging.
ClientTask	Encapsulates the entire I/O session and client request loop. This ensures blocking operations occur only on the worker thread, preventing the main server thread (the acceptor) from being blocked, allowing continuous acceptance of new connections.
RejectedExecutionHandler rejectedHandler()	This prevents the server from resource exhaustion by sending a message to the client when the thread pool and bounded queue are full.
addShutdownHook	This ensures that fsManager.close is called on JVM exit to persist metadata and close disk file descriptor on shutdown.

1.3 FileSystemManager:

getInstance(filename,totalSize): Returns singleton instance, allows only one FileSystemManager.

resetInstance(): Closes current instance and clears singleton.

initializeMemoryStructures(): Creates in-memory FEntry and FNode arrays.

formatFileSystem(): Writes empty filesystem structure to disk.

loadFileSystem(): Reads existing filesystem from disk into memory.
writeContentsToBlocks(firstBlockIndex, data): Writes data to allocated block chain.
allocatedBlocks(data): Allocates and links blocks for data and returns first FNode index.
writeFEntrytoDisk(index): Writes single FEntry to disk at calculated offset.
writeFNodetoDisk(index): Writes single FNode to disk at calculated offset.
findFileIndex(fileName): Linear search for filename in inodeTable.
countFreeBlocks(): Counts available data blocks using freeBlockList.
getDataBlockOffset(blockIndex): Calculates disk position or data block.
freeFileBlocks(entry): Releases all blocks used by a file.
persistMetadata(): Writes all metadata to disk.
close(): Persists metadata and closes disk file
The main functions will be mentioned in part 2 of Algorithms.

Section 2: Algorithms

2.2.1 File Server

This section details the operational flow of the server, with a focus on the main thread and the worker threads in a multithreaded architecture:

1. **Initialization (constructor)**

- *FileSystemManager.getInstance(fileSystemName, totalSize)* is called to safely initialize the Singleton instance, which opens/creates the disk file and loads the initial file system metadata into memory.
- *ThreadPoolExecutor* is constructed with the configured minimum (core) and maximum threads,the *boundedArrayBlockingQueue for task queuing*, a *ThreadFactory* and a *RejectedExecutionHandler* to manage overload.
- On exit, the *shutdownHook* calls *fsManager.close()* to persist metadata.

2. **Accept loop on the main server thread: fileServer.Start()**

- In fileServer.start(), the ServerSocket is created and bound to the port:12345.
- The main server thread enters the loop and repeatedly calls serverSocekt.accept() to return a Socket to a client connection request
- For each accepted Socket, a ClientTask(clientSocket) object is created to handle the client request by encapsulating it.

- The ClientTask is submitted to the ThreadPoolExecutor using pool.execute(task) who will use an existing worker thread in the pool of thread to execute the client request. The main thread remains available to accept new connections while worker threads from the pool handle the client, allowing high concurrency and multithreading.

3. **In WorkerThread: ClientTask.run()**
   The Client Task is executed by a worker thread from the pool

- If the task is rejected by the executor's RejectedExecutionHandler, the task exits after notifying the client.
- The client socket input and output are wrapped in a BufferedReader and BufferedWriter
- The worker thread repeatedly reads command lines from the client, parses, validates and calls the corresponding fileSystemManager functions.
  - CREATE <filename> -> fsManager.createFile()
  - WRITE <filename> <data? -> fsManager.writeFile()
  - READ <filename> -> fsManager.readFile(), where they bytes are converted to string using UTF-8 encoding.
  - DELETE<filename> -> fsManager.deleteFile()
  - LIST -> fsManager.listFiles()
  - QUIT -> close client connection to server

Multithreading strategy:

The described design below allows multithreading because it separates the continuous task of accepting new connections from slow and blocking task of processing the client request.

Classes involved:

- FileServer: This is the main thread responsible for continuously blocking on ServerSocket.accept(), accepting new client connections.
- ClientTask: This class implements Runnable and encapsulated the entire I/O session and processing loop for a single client connection.

- ThreadPoolExecutor: Manages the lifecyle and scheduling of the worker threads, and maintains the bounded task queue for handling the submitted ClientTask objects.

When is a new thread created?

- A new worker thread is created by the ThreadPoolExecutor class when submitted task cannot be immediately handled by core threads, and the bounded queue is not yet full. Threads are created up to the maximum pool size.

When is a new thread started?

- The worker thread being's execution upon its creation or when an existing idle thread is pulled from the pool to execute a submitted ClientTask.

What task does each thread handle?

- Each worker threads executes one ClientTask at a time. The thread runs the client's command processing loopuntil the client sends, an error occurs, or an error occurs.

How are Threads reuse?

- After ClientTask.run() returns, the thread goes back to the pool of threads and remains available to handle the next submitted ClientTask, minimizing thread creation overhead.

2.2.2 FileSystemManager:

All concurrent access to the shared FileSystemManager metadata is secured by ReentrantReadWriteLock protocol to meet the synchronization requirement.

The shared read lock is acquired for read-only operations by:

- ReadFile(), listFiles(), findFileIndex()

Whereas the shared write-lock is acquired for write-only operations by:

- createFile(), deleteFile() and writeFile(), close()

The file system is based on three main variables: inodeTable, dataBlocks and FreeblockList, which are all arrays that represent a metadata table, the nodes that link to data blocks and data blocks that are available or in use, respectively.

We are using a singleton implementation for the FileSystemManager because we want to assure that there is only one instance to avoid conflicting requests. All the functions used such as: createFile, deleteFile, writeFile, readFile and listFiles follow a similar pattern. We are using error handling to handle different cases.

In the **createFile** function, we are making sure that we are not creating a file that already exists, as well as making sure that there is enough space for our FEntry, only then do we update out FEntry (inodeTable) and persist to the disk.

In the **writeFile** function, we check if there is enough free space to allocate a new block and then write data to the block. If the allocation fails then we will restore the original chain, this ensures that there is no partial allocation.

In the **deleteFile** function, we overwrite all data blocks of a file with zeros and update the FreeBlockList and FNode (dataBlocks) so that there is no more data from that file.

In the **readFile** function, we are searching for the filename and reading through the file based on its offset and the bytes that are to be read from their position in the disk.

In the **listFiles** function, we only list the filenames that are valid, which means files that were created and not deleted.
The disk layout contains the FEntry array followed by the FNode array and the data block.

Some helper functions are also implemented such as persistMetadata(). This function ensures to write all in memory the Fentry records and the FNode

Section 3: Rationale

3.1 FileSystemManager:

We choose a singleton design for the FilesystemManager over a multiple instances approach, because we want to prevent multiple instances from corrupting the disk, we don’t want to have duplicates in the memory, we want a single point of control for the inputs, outputs and synchronization. One of the possible shortcomings could be that there cannot be multiple file systems per process.

We choose fixed sized arrays for FEntry and FNode over a dynamic method because it has a time complexity of O(1) for direct indexing, and it is bounded. It is a more straightforward approach and simpler to track. One the shortcomings would be that the capacity of the array is not as easily changeable.

For the FNodes we used a linked list of blocks rather than using the possible alternative of contiguous block allocation. This approach is advantageous as it is flexible as files can be removed and added without moving other files.

3.2 Multithreading and Synchronization

Recap of Design:

Our design consists of having a ThreadPoolExecutor with worker threads handling client request encapsulated by the ClientTask class. Each ClientTask object is created in the accept() loop of the main thread and submitted to the ThreadPoolExecutor so a worker thread executes it. This prevents blocking the acceptor and allows many clients to be served in parallel. The executor is configured with corePoolsize=50, maximumPoolSize=500 and a bounded work queue of capacity 1000 (as assignment mentioned that we must be able to test for 1000 clients). The executor creates up to 50 threads immediately for first tasks, tasks are queued up to 1000. When the queue is full, the pool creates additional threads up to 500 in total. After 1500 concurrent submissions, the RejectedExecutionHandler runs to handle overloading. The RejectedExecutionHandler calls ClientTask.reject() which writes a message and closes the socket, preventing resource exhaustion.

Each worker thread runs ClientTask.run() which read lines from the socket, parses command, invokes FileSystemManager, writes SUCCESS/ERROR responses and closes the socket on QUIT or error. The worker threads are created by the ThreadFactory and started immediately by the executor when needed. When threads are not in used, they are allowed to time out after keepAliveSeconds to shrink the pool size.

In terms of synchronization, the FileSystemManager is a singleton and guards its critical section with a ReentrantReadWriteLock. readFile() and listFiles() acquire the readLock allowing concurrent readers. Whereas createFile(), writeFile(), deleteFile() and persistMetadata() acquire the write lock, where only one thread can execute a write operation at a time.

Design Advantages:

1. Single responsibility roles: The main thread is only responsible of accepting client requests and submitting them to the ClientTask, whereas the worker threads are the one responsible of executing the request. FileSystemManager handles the disk and ensure synchronization. This separation of concerns produces a modular design that is easy to understand and enables multithreading.
2. The use of ReentrantReadWriteLock enables high concurrency reading. Multiple clients can read simultaneously, but only one can perform a write operation at a time.
3. Bounded Queue and Rejection handler limits the number of tasks and rejects any excess connections. This prevents unlimited requests from crashing the server by providing an immediate message “Server is busy” to the client.

Alternative considered:

We have considered implementing a Thread-Per-Client design at the beginning. This would mean that a new Thread is created for every serverSocket.accept(), leading to high overhead. Creating and destroying threads is expensive and consume excessive memory. It also leads to excessive context-switch, increasing the CPU time. Using the ThreadPool reduces thread creation and limits resource creation, providing a better performance for short-lived tasks.

Shortcomings:

We have identified the following shortcomings in our design that could be improve.

1. Locking:
   The ReentrantReadWriteLock is applied for the entire FileSystemManager, thus all files shared the same lock. If one client is modifying FileA, all other readers and writers accessing FileB are blocked. We could implement a File-Specific Locking system using a HashMap to map each active file to its own lock. This would allow simultaneous and independent read/write access to different files. For example, Client 1 could write to FileA while Client 2 reads from FileB.
2. Metadata Persistence Overhead:
   persistMetaData writes all FEntries and FNodes, while holding the write lock, even if only one file was modified, increasing client wait time. We could implement a Dirty Bit for the Fentry and FNode structures, such that only the modified structures, marked as ‘dirty’, are written to the disk. This would decrease the client wait time and the critical section length.
3. Linear Search Complexity:
   The time complexity for findFileIndex is O(f), where f is the number of MAXFILES. Given that MAXFILES is small in our design, it does not produce a problem. However, for a realistic system with many files, this linear scan algorithm would become very expensive. We could replace the linear array for the Fentry with a Hash Table such that the average lookup is O(1).

4. We have tested all the provided tests where all passed except for the test: ThreadManagementTests.testReadersAndWritersSynchronization. Threads did not complete properly which makes deadlock possible. We have all tested the primary functions of the file server such as write, read, delete, list and create that came back successfull.