COEN 346 OPERATING SYSTEMS

Programming Assignment #2: A File Sharing Server[1]

Due date: Nov 13 2024 – 11:59pm

In this project, you will be developing a file-sharing server. A file share server is a computer or server that enables multiple users on a network to access and share files and folders in a central location. It allows users to access files on various devices. You will create a server file system simulator to store all the files created by clients.

This programming assignment **must** be completed in pairs. You will not be allowed to seek advice from other students or copy/paste someone else's code, including code generated by AI tools like Copilot, even if the code is open-source. However, you are allowed to look at online resources, tutorials, and Q&A websites to solve the problems. The entire code must be written by yourself.

Make sure to familiarize yourself with the academic code of conduct

This guide describes

- The file system to be used by the server
- The server/client architecture
- Your tasks
- Design report guidelines
- Deliverables

File System

The simulated file system is stored in a single file. There is only one directory in this file system, and there are limits on the number of files (MAXFILES) it can store and the amount of space (MAXBLOCKS) it can occupy. The key file system structures are already defined; your job is to add functions: the ability to create a file, delete a file, write to a file, read from a file and list all files.

The file system is an array of blocks. Each block is a contiguous chunk of BLOCKSIZE bytes. Metadata (described below) is stored at the beginning of the file in the first (or first few) blocks. The simulated file system contains two types of metadata: file entries (FEntry) and file nodes (FNode).

A FEntry contains:

file name: An array to store a name of maximum length 11.

[1] This assignment is based on the CSC209 programming assignment by prof. Kianoosh Abassi and Andreas Bergen from the University of Toronto

COEN 346 OPERATING SYSTEMS

size: An unsigned short integer giving the size of the actual file. Note that a file might not use all of the space in the blocks allocated to it.

firstblock: An index into the array of fnodes. The specified fnode knows where the first block of data in the file is stored and how to get information about the next (second) block of data.

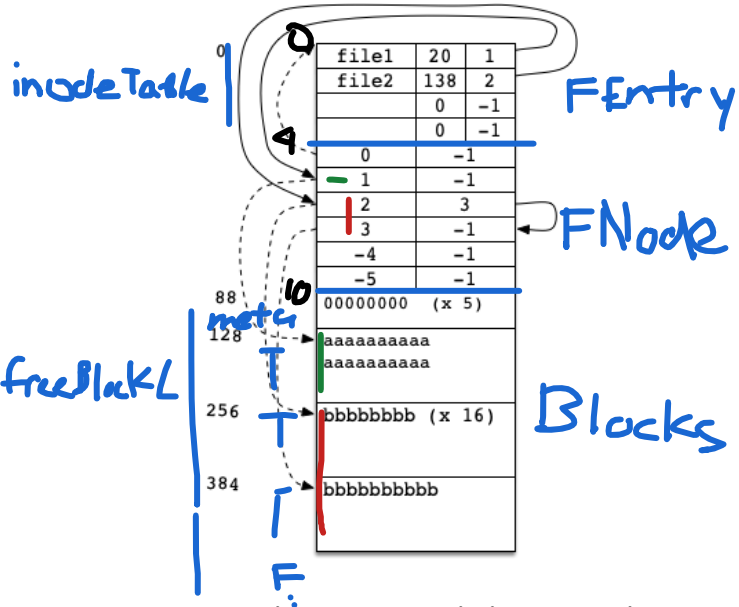
An FNode contains:

blockindex: The index of the data block storing the file data associated with this fnode. The magnitude of the index is always the index of the fnode in its array; it is negative if the data block is not in use.

nextblock: An index into the array of fnodes. The specified fnode contains info about the next block of data in the file. This value is -1 if there is no next block in the file.

There are a fixed number of fentries and a fixed number of fnodes, so it is possible to calculate how much space they will take. The array of fentries is stored at the very beginning of the file, followed immediately after by the array of fnodes. The number of blocks required to store the fentries and fnodes depends on how many of them there are (MAXFILES and MAXBLOCKS).

For example, suppose the maximum number of files is 4 and the maximum number of fnodes is 6. The block size is set to 128. The fentry and fnode arrays take 84 bytes ((15*4) + (4 * 6) = 84), so they can fit in a single block. As a result, the first fnode is in use (block 0 contains metadata), and file data begins in the second block (index 1). The real file containing the simulated file system would be arranged as shown in the following diagram. Please note that the boxes are not scaled according to the number of bytes they occupy.



Note that it is a good idea to set the maximum number of files and blocks so that they fit in exactly N blocks (where N is an integer), to avoid wasting space. However, like the example above, you may not assume this is the case; there may be wasted bytes between them and the start of the file data.

Also note that file data blocks for a particular file will not necessarily be next to each other, though the example above does have them together. It is possible for a file to be in blocks 1 and 3, for example, with another file at block 2.

Server/ Client Architecture and Sockets

A server is a software programs that handle requests from clients (such as web browsers) and sends back responses containing web pages, files, or other resources. Traditional clients and servers communicate using a predefined protocol. The client initiates a connection and sends a request for a specific action or resource, and the server processes this request and sends a response. The client then processes and presents the received information or takes further action.

Servers are identified by both the machine they run on and the port they use for communication. Ports are crucial because they allow multiple, independent network activities to happen simultaneously on a single machine. For example, a mail server might handle email on port 25, while a database server handles data requests on a different port.

While many protocols have a standard port, a different port may be used depending on the specific configuration.

The protocol in this project is simple. The client sends a command along with the parameters of that command, and the server sends a response.

The commands are the following

- CREATE <filename>: Creates a new empty file in the filesystem. Filename must be 11 characters of less, otherwise the server sends the message “ERROR: filename too large”
- WRITE <filename> <content>: Writes the contents in the specified file. Contents may be text or bytes. If the file with the specified name has not been created the server sends the message “ERROR: file <filename> does not exist”. If there are not enough free blocks in the filesystem, the server sends the message “ERROR: file too large”
- READ <filename> : Reads the file with the specified filename and sends it to the client. If the file with the specified name has not been created the server sends the message “ERROR: file <filename> does not exist”
- DELETE <filename> : Deletes the file with the specified filename. If the file with the specified name has not been created the server sends the message “ERROR: file <filename> does not exist”
- LIST : Sends a list of all files existing in the server.

Your tasks:

File System Operations:

COEN 346 OPERATING SYSTEMS

You should implement the file system operations. These operations should work for any positive, valid values of BLOCKSIZE, MAXFILES, or MAXBLOCKS. (Valid values will not cause the short values in the structs to overflow.)

- void createFile(String filename) throws Exception: The createfile command takes a single argument: a simulated file name. It will create an empty file of that name on the file system if it is possible to do so, using the first available fentry. If there are not enough resources on the file system, an error should be emitted.

- void deleteFile(String filename) throws Exception The deletefile command also takes a single argument: a simulated file name. It should remove the file from the file system, including freeing any blocks used to store the file. **To avoid malicious use of old data, your operation should overwrite the file data with zeroes**.
- void writeFile(String filename, byte[] contents) throws Exception: Writes the contents in the file, overwriting any existing data. If the file does not exist or if the file system doesn't have enough free blocks available for the write, then an error should be emitted. In any error case, it is imperative that no changes be made to the filesystem. The file system cannot be left in an inconsistent state; the whole operation must be completed, or none of it.

- byte[] readFile(String filename) throws Exception: Returns the contents of the file – up to the size of the file (no additional bytes should be read). If the file does not exist, an error is emitted.
- String[] listFiles() : returns a list of all filenames existing on the server.

Multithreading:

The current implementation of the server is single-threaded. This means that each new client needs to wait until the previous client has finished for its request to be served. To help you see this, we have provided a SimpleWebClient class that waits for one minute before sending a request to the server. Run the server, then run the client and then open the webpage and see what happens.

To solve this problem, implement a multithreaded approach to allow the server to handle multiple client connections concurrently. You will need to detail your strategy (classes used, when a new thread is created, when it is started, what task does each thread handle) in the report.

To evaluate you, we might create thousands of clients, your server should support them.

Synchronization and Deadlock Prevention

As you implement multithreading, it's crucial to be aware of synchronization issues. When multiple threads access shared data concurrently, it can lead to race conditions and data

corruption. Multiple clients should be able to read files concurrently, but there should only be one writer at any time. If there is a writer, there should be no readers.

You may use Java's synchronization tools.

Error Handling

Your server should display errors to the client, but it should continue to receive requests from other clients. The client should also be able to send another request in the event of an error.

Design Report: Offer a report detailing your multithreading strategy, your synchronization strategies implemented and how they address potential issues. Include any testing or scenarios you used to validate the correctness of your implementation.

You must explain the following 3 aspects of your proposed design. Create sections for each of these aspects.

- **Data structures and functions** – Describe any struct/class definitions, global (or static) variables, typedefs, or enumerations that you will be adding or modifying (if it already exists). You should write this using UML diagram. Include a brief explanation the purpose of each modification. Your explanations should be as concise as possible.
- **Algorithms** – This is where you explain how your code works. Your description should be at a level below the high-level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. Instead, you should try to convince us that your design satisfies all the requirements, including any uncommon edge cases.
- The length of this section depends on the complexity of your design. Simple explanations are preferred, but if your explanation is vague or does not provide enough details, you will be penalized. We recommend that you split the explanation into parts. Describe your algorithm for each part in a separate section. Start with the simplest component and build up your design, one piece at a time.
- **Rationale** – Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

COEN 346 OPERATING SYSTEMS

Deliverables:

You should submit your code via the github repository. To submit, simply commit and push to your repository.
In addition to the repository, submit to Moodle:

- a zip file of the code
- a pdf file of your design report.

File names should be in the format <SID1>_<SID2>_assignment2_<type> Where SID corresponds to the student id of each member of the group and type is either code or report.

Grading Criteria

[25%] File System
[25%] Server commands and Multithreading – handling clients
[25%] Synchronization
[10%] Design Report
[10%] Coding style
[5%] **Github** practices

Your program will be evaluated not simply on correctness (i.e., whether it follows the assignment specification), but also on good design and coding style (documentation, naming, consistency, modularity, error checking and handling).
Follow the coding style guide for specific advice on style.

For GitHub practices, both students should use the same repository and commit regularly. Contributions are roughly equal.

Additional resources

Thread documentation
https://download.java.net/java/early_access/panama/docs/api/java.base/java/lang/Thread.html

Concurrency tutorial (for java version 8, note java is currently version 21). Main concepts remain https://docs.oracle.com/javase/tutorial/essential/concurrency/

Virtual threads
https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-DC4306FC-D6C1-4BCC-AECE-48C32C1A8DAA