Problem Statement 3

[Online File Editor with Collaboration]

In this assignment, you will design an online file editor with online collaboration features. There will be a single server, which can handle multiple clients at the same time (max. 5 concurrent client connections).

Connection Phase:

The server creates a socket and listens for client connections on a specific port. The client should send a request for connection to the server, and an appropriate reply should be returned based on success or failure.

- ***Successful Connection:*** If successful, the server generates a unique 5-digit ID w.r.t. client socket id, and stores the client details in a *client records* file/data structure. A welcome message is returned to the client.
- ***Unsuccessful Connection:*** If unsuccessful, either because more than 5 clients are currently active or some other reason, an error message is returned to the client.

Server Functionalities:

After successful connection, a client can now access different functionalities by sending queries to the server.

– – ***Viewing active clients***

A client can request the server about the details of all active (currently connected) clients. The server returns a list of client IDs.

– – ***Uploading a file***

A client can upload a local file to the server. The server should create its own copy of the same file, and return success / failure messages.

The client successfully uploading a file is called the **owner** of the given file. The server should maintain a track of all files and owners in a *permission records* file/data structure. Apart from owners, there can be collaborators (see below).

For specific program requirements (as you will understand below), it is better to ensure that files with the same name are not allowed. Thus, if a client *C1* uploads a file A.txt, then if a client (*C1 or other)* wants to upload another file named A.txt, then the server should reject the operation and send back an appropriate error message.

– – ***Providing access to another client***

A client can allow access to other clients to *any file it owns.* If a client (say C1) asks the server to provide access to another client (say C2) to a particular file (say *f)*, the server dispatches an invite to C2. If C2 accepts the invitation, C2 becomes a **collaborator** of *f*. This invite (and subsequently on acceptance, the collaborator C2 itself) can be of two types, based on the type of access C1 wishes to give to C2:

- Viewer (V): Read-only privilege, C2 will not be able to modify *f*
- Editor (E): Read+write privilege, C2 can modify *f*

  – – – ***Accepting invites from other clients***

  As discussed above, if C1 invites C2 as a collaborator for *f,* an invite message should automatically be sent by the server to C2 with all details. If C2 accepts the invite, the server should perform the necessary actions as described above.

- ***Successful Invite:*** If the invite is successful, the server should send success messages to both C1 and C2, and maintain track of collaborators (and their access privilege V/E) along with owners in the *permission records* file.
- ***Unsuccessful Invite:*** An invite can fail in many scenarios (file does not exist, C1 is not the owner, C2 declines the invite, etc.). The server should send appropriate failure messages to C1 (and depending on the situation, to C2 also).

– – ***Reading lines from a file***

A client can request to read lines from a file by sending a query to the server. A client may be allowed to access the file (if it exists) *only if it is the owner or a collaborator (either viewer or editor) of the requested file*.

The client must also indicate in its query the range of line numbers it wishes to read, by specifying a starting and ending index. These indices may be in the range [-N, N) where N is the total no. of lines in the file (similar to what was discussed in Assignment 1). If only a single index is specified, only that single line should be returned. If no index is specified, the entire file should be read (start=0, end=-1 or end=N-1).

- ***Successful Read:*** If the read is successful, the requested line(s) are returned to the client.

- ***Unsuccessful Read:*** Appropriate error message to be returned to client based on situation (file does not exist / client does not have access / invalid line numbers)

– – ***Inserting lines to a file***

A client can request to insert line(s) into a file by sending a query to the server. A client may be allowed to modify the file (if it exists) *only if it is the owner or a collaborator (editor only) of the requested file*.

The client must indicate in its query the message (unlike Assignment 1, this message can contain newline characters) it wants to insert, and the line number it wishes to insert at, by specifying an index as above. If no index is specified, the contents of the message should be added at the end of the file.

- ***Successful Insert:*** Return the entire contents of the modified file
- ***Unsuccessful Insert:*** Appropriate error message to be returned to client based on situation (file does not exist / client does not have access / invalid line number)

– – ***Deleting lines from a file***

A client can request to delete line(s) into a file by sending a query to the server. A client may be allowed to modify the file (if it exists) *only if it is the owner or a collaborator (editor only) of the requested file*.

The client must also indicate in its query the range of line numbers it wishes to delete, by specifying a starting and ending index. These indices may be in the range [-N, N) where N is the total no. of lines in the file. If only a single index is specified, only that single line should be deleted. If no index is specified, the entire file contents (not the file itself) should be deleted (start=0, end=-1 or end=N-1).

- ***Successful Delete:*** Return the entire contents of the modified file
- ***Unsuccessful Insert:*** Appropriate error message to be returned to client based on situation (file does not exist / client does not have access / invalid line number)

Server Side:

- Create a socket and wait for client connections; once a client connects, generate a unique 5-digit client ID and store *client record* details (in file/data structure)

- Accommodate upto 5 concurrent client connections at the same time (refuse further connections if limit is reached, until some client leaves)
- Service multiple clients at the same time using select() / fork()
- If new file is uploaded / access privilege is granted for existing file, update the *permission record* details (in file/data structure)

Client Side:

- Connect to server
- Take command input from user, parse the command, and send appropriate query message to the server
- Retrieve server outputs and display to user

User Commands: To be entered at client-side input, parsed by client and appropriate communication (passing queries and collecting outputs) with server

1. `/users`: View all active clients
2. `/files`: View all files that have been uploaded to the server, along with all details (owners, collaborators, permissions), **and the no. of lines in the file.**
3. `/upload <filename>`: Upload the local file *filename* to the server
4. `/download <filename>`: Download the server file *filename* to the client, if given client has permission to access that file
5. `/invite <filename> <client_id> <permission>`: Invite client *client_id* to join as collaborator for file *filename* (should already be present on server). *permission* can be either of V/E signifying viewer/editor.
6. `/read <filename> <start_idx> <end_idx>`: Read from file *filename* starting from line index *start_idx* to *end_idx*. If only one index is specified, read that line. If none are specified, read the entire file.
7. `/insert <filename> <idx> "<message>"`: Write *message* at the line number specified by *idx* into file *filename*. If *idx* is not specified, insert at the end. Quotes around the message to demarcate it from the other fields.
8. `/delete <filename> <start_idx> <end_idx>`: Delete lines starting from line index *start_idx* to *end_idx* from file filename. If only one index is specified, delete that line. If none are specified, delete the entire contents of the file.
9. `/exit`: Disconnects from the server, and then all files which this client owned should be deleted at the server, and update the permission file.

To avoid any confusion regarding the file in question for commands 5-8, we asked you to reject multiple files on the server with the same name.

Submission Instructions:

- You should have one file for server **“server.c”** and one for the client, named as and **“client.c”**.
- You should include a user manual / ReadME **“readme.txt”** describing your design choices in brief. Also, you may include instructions for the evaluators to follow, so that your program is evaluated the way you have designed it.
- Save this in a folder named in the format: **<Roll No.>_CL2_A3**. Compress this folder to tar.gz format, creating a compressed file **<Roll No.>_CL2_A3.tar.gz**. Upload this compressed file to moodle. *Example: If your roll no. is 21CS60R05, the folder should be* 20CS60R05_CL2_A2*, and the compressed file should be* 21CS60R05_CL2_A2.zip. Please use zip only, since some you have reported issues with tar.
- Not adhering to these instructions can incur a penalty.

Marking Scheme:

Setting up and maintaining server-client connections	**30**
Collaboration and Maintaining permission record	**25**
Uploading and downloading files	**25**
Read	**30**
Insert	**30**
Delete	**30**
Design Decisions and Error Handling	**30**

Design Guide:

- From Assignment 1 and 2, you know how to transfer messages between server and client
- From Assignment 2, you also know how to serve multiple clients using fork()/select() and non-blocking sockets.
- There are two separate aspects to handle for this assignment:
    - Requests for file operations, such as upload, download, read, insert and delete are exactly similar to that in Assignment 2.
    - However, the collaboration invite sending and acceptance/rejection handling is arbitrary.
    - *A particular client **C1** might be interested in performing any of the above commands (client-side is at any point of handling them), when a request might be generated by multiple clients **C2**, **C3**, etc. (imagining the worst case scenario for you to handle). How should you handle this (both server and client side)?*
- Well, you can handle it like the way we describe below (there may be multiple ways to do this, realistically)

Client-side:

- The client establishes connection with the server, and then forks into two processes that run on infinite loops — one for sending messages to the server, and the other for receiving messages from the server. This will help declutter most of the steps below.
- So, the process involved with sending server messages would normally flash a prompt at the terminal and wait for user input. After obtaining user input, it parses, creates message to send to server, and then sends in non-blocking fashion, thereby looping back to flash the prompt again
- On the other hand, the process involved in receiving messages would normally just wait for server messages and display them (and perform post-processing actions, such as saving to the physical file for the /download command, etc.)
    - This would ensure that any requested operation from the client side (that was sent by the sender process) completes, via the reply from the server that is handled by the receiver process.
    - The only issue is, since the sender process does not wait for the server reply, multiple requests could be issued by the client before the first server reply comes in (theoretically possible). This might become very difficult to manage.
    - You can ensure that the client cannot issue another command when one is still pending. This can be done via semaphores. After the sender sends a message to the client, it updates a shared variable that indicates that current operation is pending. It then waits until the value of this variable is changed. On the other side, when the receiver process receives the server reply, it updates the shared

variable indicating that now further user commands can be issued. This ensures synchronization between the processes.

- So far, so good. But what about the join requests? The receiver process can receive a message from the server that indicates a collaboration request, which would require an immediate Yes/No response from the client. How to take this input at the client-side terminal without getting confused with the sender process, which also waits for user input?
    - Again, semaphores are the key. Another shared variable is maintained to keep track of which process is requesting user input from the terminal. When the receiver receives a collaboration request, it checks via semaphore if the terminal is free. If so, it locks the terminal, and asks for user confirmation to the collab request (Yes/No)
    - The sender process would get blocked by the above semaphore, and would wait till the terminal is free to use again. Basically, it implies that the sender process would flash the prompt and ask for user input only if the previous command has been completed, and if the terminal is not currently being used by the other process.
    - After the user confirms Yes/No, the semaphore controlling this aspect is released.
    - On the sender side, the entire process of flashing the prompt and waiting for user input should be guarded by the semaphore.
- So, you have to operate two semaphores at the client side. One for checking if the pending operation is complete, the other for controlling that only one process should be accessing the terminal at any given time. Make all your sends non-blocking.

Server-side:

- At the server side, you do not need any critical handling like this. It is advisable to use fork() instead of select(), to make it easier to manage. A new server process is initiated for every new client, and can be used for handling like you have normally done so far.
    - For all other commands, the server process waits for client message. On receiving, it does necessary processing and sends output/error back to the client using non-blocking send.
    - Only if it is a collaboration request command, then an extra handling is required. It must send the invite request to the desired client, wait for its response Yes/No, and then send successful/failure messages to both involved clients, before resuming.
    - Use semaphores to guard server-side operations like updating the connections and permission records.
- Use infinite loops on both sides. Diligently use semaphores to ensure that conflicts are avoided.

Options and Requirements:

- You are free to try out any approach you like. There might be several other ways to handle the above situations, such as:
    - using two parallel server-client connections for handling file operations and collaboration requests separately
    - using a single client-side process, which somehow checks for collaboration requests in between the normal loop (receiving user input, sending to server, receiving server reply, printing to client)
- You can use any technique you used in Assignment 2 to transfer the files. However, sending it in one go is advised to avoid multiple send()/recv() calls.

Beyond design decisions, there are some things which you absolutely have to implement.

- User commands should be strictly in the format mentioned.
    - Remember to reject uploads having the same filename, apart from all the other possible errors.
- You should be performing read, insert and delete operations strictly at the server-side
- It is absolutely crucial that you handle the collaboration requests in real-time, i.e., the client should be notified as early as possible from the time of generation of such a request
    - You might at most wait upto completing the ongoing client operation, and then notify the client and handle the response
    - This should be a notification, i.e., it should be handled automatically from the end-user's point of view. No command or input should be required at the terminal for the notification to be flashed (it will subsequently be required to handle the user's choice Yes/No).
- Since we are testing the server and multiple clients on the same physical device, it is imperative to use some methods to distinguish all these files
    - You may store them in different folders. It is then fine, to require the end user to place their files to upload in that specific folder itself.
    - You may use different filename prefixes / suffixes to demarcate server and different clients
- For displaying client and file details, the format is up to you. However, the following information should be displayed:
    - For client records, both client ID and socket ID
    - For file records, information regarding the owner and all collaborators (with their requisite permissions), and the no. of lines in each file.
- Both server and client should run on infinite loops and not exit unless the user closes the terminal or the server gets disconnected.
- **ALL** CLIENT-SIDE COMMANDS SHOULD HAVE SOME KIND OF RESPONSE FROM THE SERVER
- The ReadMe is absolutely crucial for this assignment.
    - No need for paragraphs, however, include the necessary details and instructions.