**Title: Centralized Multi – User Concurrent Bank Account Manager**

Ask : To design and implement a multithreaded bank server which handles multiple and concurrent client requests.

Design:

* We implement this server using sockets for establishing connections and mutex locks for maintaining correct information on the record files.
* On the server, all the data that is present in the Records file is stored in an array of structures which have the variables to store the account number, name and balance of the user.
* Based the request these values are updated and each time a new request comes we load these values into the array.
* Server needs to handle multiple requests at the same time and this is done through a pool of threads
* On the client side we read each line from the file and send it to the server through multiple threads and their respective connections with the server
* Once we get the request, from the client ,we need to handle transactions like cash withdrawal and deposit on the server. This is decided through tokenizing the request and taking out the appropriate values.
* If no record of the user is found or if it’s an invalid transaction, same will be intimated to the client.
* Upon a successful transaction, a success message is sent to the client as an acknowledgement.

How it Works:

* So, to start with we establish a server by using sockets and bind it with port #8989 (can be changed, if needed but for now it is hardcoded in the application).
* Now, this server is ready to listen to the requests that come along the way.
* On the client side on the application, we ready the Transactions.txt file and each line in the file is a request to the server i.e. a transaction that needs to be executed on the server.
* We created a thread of arrays on the client side, whose maximum size is decided by the number of lines in our txt file.
* For every line of request that is read, a thread is created. This thread will connect to the server(done through sockets) and the request is passed as a message to the server. This keeps on happening until we reach the end of the file.
* Now, when the server receives a connection request and if its able to successfully connect with the client, the immediate step that we do is to push that connection -id(socket-id) into a queue.
* This queue is continuously monitored by a pool of threads, as soon as a thread is freed up it just checks if there any connections that are to be handled are present in the queue. If yes, then the thread picks up the connection and performs the required operations to get the data from the message that was sent. If no, it just waits.
* This enqueuing and dequeuing of the connections needs to be handled through mutex locks , as we don’t want to risk the possibility of a deleted pointer getting picked up by another thread at the same time or valid pointer being deleted while its still yet to be picked
* We handle this through mutex locks and condition variables. If there is a connection that is pushed to the queue, it signals the thread, which has been waiting for a connection to appear.
* Now, after determining the operations that are to be performed, we search the file for the appropriate records and if no record is found of the user, we intimate the same to the client.
* As long as we are searching, we just rewrite the content that is read back to the file and if the record is found we just update the balance and write the data back to the file
* As both the read and write operations that we are doing here needs to have data, that is not being manipulated or updated parallelly, we acquire a lock before the start of the search and unlock it once when we are doing with it.
* Finally, we close all the socket-ids that were created.

Compiling and Running the code:

* To compile the code, navigate to the src and folder and type the following command :
  + make compile (This compiles the files and creates object files)
* To Run the code,
  + ./server Records.txt ( This starts the server, here the first argument being the executable file and the second one being the records file).
  + ./client Transactions.txt (This starts the client side of the application and with the Transactions.txt file being the file of requests).
  + ./client Transactions.txt 0.2( This start the client side , here 0.2 is the timestamp and this is optional)

Multithreading and synchronization are handled through the creation of a pool of threads which picks up connections as soon as we get one and through acquiring locks over the critical sections of the application so that no threads are trying to write at the same time.

Possible Improvements:

* Instead of blocking the whole file maybe we can block only the record on the file. And I believe that this can be handled by declaring a lock to each record in structure that we have defined.
* Maybe we can try to run the server on multiple ports and then navigate the clients based on the traffic.