Requirement

To implement a **Centralized Multi-User Concurrent Bank Account Manager**. The system has two important components:

Bank Server: The server program that services online requests for account manipulations and maintains all customer records correctly.

Clients: Customers are clients of the bank server and use its services to update bank accounts. The operations that can be performed on an account are: **withdrawal** of an amount from an account and **deposit** of an amount into an account. Additionally, the bank server can have its own service that periodically deposits an interest amount to each account based on some fixed rate.

Program Design

The program has been developed in C++ on Ubuntu Linux. The communication between server with multiple clients has been implemented using stream sockets.

Working/Handling multiple threads & synchronization:

Server: For server to receive queries from customers, the server reads a records file, that works as server database in which it has customer records: customer's account number, name and current balance. The socket system calls are performed on server to create socket, bind it with host(server) address and the server is made ready to listen to its clients. As soon as the server accepts a request from the client, it sends the acknowledgement that the connection has been established. For each request received by client, a new thread is created using pthread. A function to perform operations of withdraw and deposit is called in that thread for each client. For transactions to be performed server receives another transaction file from client that lists down all transaction in following space separated format:
<timestamp> <account number> <transaction type(withdrawal(w)/deposit(d))> <amount>
If the transaction type is d or deposit, the server adds the mentioned amount to the customer's balance amount, and if the transaction type is w or withdraw, the server subtracts the mentioned amount from the customer's balance amount. If the withdrawal request specifies amount more than the available amount in records, the server gives out a message saying insufficient balance. To allow multiple user work on multiple transactions, a mutex lock is incorporated to avoid race condition. Once the connection is established and thread is created, the socket gets closed.

Client: Client is executed with command line arguments specifying hostname, port number, timestamp and transaction filename to connect to respective host/server. For client to send queries to bank server, the client reads a transaction file. A new socket is created on client side and a connection request is sent to server for connection. When server accepts the client request, the client reads the transaction file and sends queries line by line ordered based on the timestamps associated with each transaction to avoid race condition. The process will sleep until the previous transaction process releases the mutex lock. The lock will be released when the time taken to execute the process plus the specified timestamp mentioned is over. Once the connection is established and thread is created, the socket gets closed. A clock has been incorporated at the beginning and end of each transaction to collect reading for system's performance evaluation.

How to compile and run the code

A makefile has been created that lists down commands as follows:

- compile: make compile

writing "*make compile*" in command line will create the server and client objects.

- serverRun: make serverRun

writing "*make serverRun*" in command line will run the server.

- clientRun: make clientRun

writing "*make clientRun*" in command line will run the server.

- clean: make clean

writing "make clean" will clean all the objects created by the makefile.

The client request rate is by default set to 0.5 in makefile clientRun command. This can be changed by editing in makefile. Also, changing hostname and port number can be changed accordingly by editing in makefile.

Design tradeoffs

Although the system accurately handles synchronization using the timestamps specified in transactions file, however, one of this adds overhead of sending timestamps manually with each transaction. Also, transaction with same timestamp is not handled by the system.

Possible improvements and extensions

Instead of using timestamps mentioned manually with each transaction, one of the possible improvements could be to record the time at which the request is made using clock synchronization. Using select() function for the processes with same timestamps can be one of the other extensions to the project.