**OS Project 1 CMSC 621 Centralized Multi-User Concurrent Bank Account Manager**

**Submitted By: KAUSHIK VELUSAMY**

**“How it works”**

The system has two components.

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

2. Clients Customers: Clients are used to update bank accounts. You can withdraw and deposit to an account. More than 1 client can be used.

**The Server program has the below functionalities:**

1. Accept multiple concurrent customer requests (i.e., must be multi-threaded)
2. Provide locking/protection for access to an account records during shared access.
3. Maintains correctness of records at each record, (i.e., allow withdrawal from an account only if it has sufficient funds etc.)
4. The Server initially has the Records.txt database to start with.
5. The format for each line being: account number, name, balance amount (space separated)
6. The Server receives the status of transaction from client and print to the console.

**The Client program has the below functionalities:**

1. Issues withdrawal or deposit requests.
2. The client issues requests at fixed time intervals. A client reads the input file for transaction information (Transactions.txt) and perform those tasks accordingly.
3. Transactions.txt has the following format: timestamp, account number, transaction type (withdrawal /deposit) , amount (space separated)
4. The client receives the status of transaction from server and print to the console.
5. Multiple clients can be started at the same time to communicate with the server

**Evaluation Checklist:**

1. Client and Server is implement in C++
2. The Program is developed and tested in a linux Machine.
3. Makefile handles the compiling the server and the client and starting the server. Make and Make clean
4. The server handles multiple transaction requests at the same time.
5. The server provides protection for shared simultaneous access to the same record using Mutex locks.
6. Correctness: server preserves correctness of transactions by deducting money only when available
7. The program is multi-threaded
8. The Server accepts requests concurrently
9. This program print messages for the data being sent by the server and for the data being received by the client.
10. **Demonstrated in the Attached Video: The use of locks that protects simultaneous access to the same account.**

**To Run the Program:**

*Open new terminal for server*

Run make clean

Run make

Or

To compile: g++ -o server server.cpp -std=c++11 -pthread

To Run: ./$server 8005 Records.txt

*Open any number of terminals for Clients*

Run

./Client 1 127.0.0.1 8005 0.1

./$(ClientObject) $(CLIENTID) $(IP) $(PORTNUM) $(TIMESTEP) $(TRANSACTIONFILE)

**Mention Different ClientID on every client that you try to connect to the server for clarity of the Design**

**Design:**

1. The server program uses Records.txt for the initial Database.
2. The Client program uses Transactions.txt for the transactions to perform.
3. The Server gets Each line from the Transaction line from the client and performs the specified operation on its Records Database.
4. Server creates the socket, binds and listens for the incoming client connections.
5. Client creates a socket and connects to the server.
6. Each connection to the server from the clients is handled by multithreading \*create\_thread .
7. Separate thread for each clients.
8. After the client has established the connection with the server. The clientID is sent to the server.
9. Each line from the transactions.txt is sent from the client to the Server and Acknowledgment is Received at both the ends.
10. Each line in the transaction is sent after timestep steps mentioned in the client argument to the server.
11. The server loads the Records.txt to its struct array
12. The server checks if the account number from the transaction line exist in the Records struct array, only then it proceeds further
13. For deposit d in the transaction line, the deposited amount is added to the account.
14. For withdraw w in the transaction line, the withdraw amount is withdrawn if it already has sufficient amount to withdraw, else the transaction line is cancelled and reported to the client.
15. For Each transaction, the old status: Account Name and Balance and the new Status : Withdraw/Deposited Amount or Failure Reason and the New Balance is reported to the client
16. A mutex lock is implement before every transaction is performed on the account. This ensures synchronization and correctness.
17. The server ends the client connection after completing all the transaction process from the client.

**Future Works:**

1. The Design could be optimized to improve the performance
2. Condition Variables with semaphores can be implemented on the locking part.
3. Security Features can be implemented on the server for each client.
4. Transactions from clients can be sent in bulk and can be handled to avoid Network latency for each Transaction.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*