**Project 1 -** **Centralized Multi-User Concurrent Bank Account Manager**

**Design and Implementation of Server:**

Server has 2 structures which are used for storing the contents of “Records.txt” file and each transaction that are sent from client. It accepts the connection from the multiple clients. First of all, it extracts all the data from the records file and saves it in a structure. Meanwhile, it gets the connection from the client and acknowledges back that the data has been received. It checks whether the account number that the client is trying to access is present in the records file. If it is not present it sends checks with the client and creates a new account. Once it is done, it makes the necessary changes to the account number, as sent by the client and updates the records file so that other transactions can easily use the same file. Moreover, the server program adds an interest rate of 5% on all the amounts present in the records file for every 30 seconds. In order to have effective data communication server uses below concepts:

**Sockets:** Sockets are used for inter-process communication between the server and client. The server program is the bank server which handles different transactions from multiple clients and ensures that the customer records are updated after each transaction.

**Synchronization:** Mutex has been implemented in the server program. This refers to the mutual exclusion. Whenever a thread tries to make changes to the records file, it acquires a lock until it completes its action. During that time, no other threads would be able to acquire the lock. To illustrate, when the client connects to server and tries to execute multiple transactions, it will first acquire the lock on the records file present on the server, then performs necessary actions and finally releases the lock. Similarly, when multiple clients connect to the server and try to execute multiple transactions, each transaction would acquire a lock one by one. Once the transaction is complete, it releases the lock. Only then, other transactions would be able to access the file.

**Multithreaded:** Multithreading concept is implemented in the server program. It performs multitasking and allows more than one program to execute concurrently. Whenever the client connects to the server a thread is assigned to the client. Vector threads are used when there is no awareness of the number of clients that would connect to the server. Each thread is assigned a particular task. For example, a client connects to the server and tries to execute multiple transactions. In the client program, it tries to extract each line from the transaction document and sends each of them to the server. When the server receives the connection, it assigns a thread to each transaction. Each of them performs necessary actions as mentioned in the transaction type (deposit or withdraw the amount).

**Files:**

“Records.txt” (account number, name, balance amount (space delimited))

“Transactions.txt” (timestamp, account number, transaction type(withdrawal/deposit), amount

(space delimited)) – where amount is of float datatype.

**Test cases:** Below test cases have been performed:

* CASE1: Funds not available: test case to show the client can't withdraw money from the bank if the money is not enough.
* CASE2: Protect simultaneous access: test case to show that semaphore/locks are properly used to lock a record so that no more than one transaction can process on the same record at the same time.
* CASE3: Scalability of the client number:  test case to try different number of clients to see how the number of clients impact the average transaction time.
* CASE4: Scalability of the client's request rate:  test case to try different request rate to see how request rate impacts the average transaction time.

**Syntax for compiling:** ./server <portnumber>

**Design and Implementation of Client:**

Client has a structure for storing the contents of each transaction. It uses TCP socket to connect to the server and exchange the transaction details. Once the connection is established, server sends a message to the client to notify that the client can start sending the transaction details. Once the client receives the message, it reads the “Transaction.txt” file line by line, which contains multiple transaction details for different accounts, and extracts the details from each line and then sends them to the server. Each line has a timestamp which means that the transaction must be executed after the particular timestamp mentioned. The amount present in transaction file has a float data type. For each transaction, client gets a success message from the server. Furthermore, when the client tries to access an account which is not present in the records files located on server, it gets a notification from the server about the new account creation. Depending on the client’s choice an account is created. Once the account is created, server sends a message to the client and provides the information about the new account. In addition, client receives an error message as well. For instance, when a client wants to access a particular account to withdraw funds and there is no sufficient balance in that account. In this case server sends an error message to the client. So, an effective data communication has been established between the server and the client.

**Syntax for compiling:** ./client <IP address> <portnumber>

**Compilation and Execution of Makefile:**

Makefile is used to compile the source codes and executes the commands as mentioned in the targets:

* compile: “make compile” compiles both the server and client source codes. It creates two executable files, server and client.
* clean: “make clean” removes all the existing object files.

**Possible Improvements:**

* New accounts for the existing account name can be created.
* Minimum balance could be maintained for each account and client should receive a notification incase the balance reduces below the threshold amount.
* Mutex has been implemented in the program which locks the entire file while processing a request. Instead, vector of mutexes could be implemented to lock only the record which needs to be updated rather than locking the entire file. This would definitely increase the processing time of other clients.

**Rashmi Prava Patro**

**AK14498**