**Project 1:**

**CMSC 621: Advanced Operating Systems**

**By: Syed Naqi Hussain**

**xk39135**

**Design Documentation**

**====================================================================**

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

**====================================================================**

**Index**

1. **What is this about?..............................................................................................1**
2. **How it works?......................................................................................................2**
3. **Design considerations and trade-offs………………………………………... 4**
4. **Scope of further development……………………………………………...… 4**
5. **Results……………………………………………………………………….….5**
6. **References………………………………………………………………………6**

**1. What this is about?**

This project tries to capture the intricacies of a multithreaded system by implementing a Socket based, one server multiple clients bank systems. A peek into a banking system would reveal how a single bank server ensures multiple clients are simultaneously interacting with it.

An example client can be an ATM which is trying to access a bank account. This ATM would query the bank server for the account details and then sends it transaction updates to the server. The server on its side will check and authenticate if the bank account is present in its records and then proceed with the transaction.

Imagine multiple ATM’s trying to contact the bank server with its transactions. Every ATM/client would except that their transactions are getting completed in the least time possible. In order to ensure this, we make our server a bit more robust by utilizing concepts of Multithreading. A server creates a thread for every client it connects with and the transactions are completed on the thread, instead of the server directly. This way the server distributes its load and makes sure every client is being served in the minimum time possible.

Come to think of it, you would realize that such a system would run into trouble when a bank account is simultaneously being updated with transactions from two different clients. In such a case **Client A** would go into a deadlock until the other **Client B** completes its transaction.

Consider the below example:

Let’s assume **Account no 100** has a balance of 50000 dollars in its account. Two clients are requesting to access this account as given below.

Client A: Account no 100, deposit 1000

Client B: Account no 100, withdraws 51000

Client A connects with the server and initiates its transaction, and even before this transaction ends, Client B initiates its transaction. As a result, server gives a wrong message that there are insufficient funds.

To ensure such cases do not occur, the system uses **mutex locks** which avoids scenarios described above, by locking the server until the whole transaction is completed.

**2. How it works?**

This system is built using C++ programming language. Given below is a high-level description of what the source code does.

The source code is divided into two into a Server and a client code. The server first opens a Records.txt file and stores the data, by segregating the values in it.

A model **Records.txt** file would look as shown below.

101 SyedHussain 50000

114 RahulK 45000

123 KalyanK 80000

The first column represents the account number, second has the username and the last column has the balance associated with each of those accounts.

Steps for server:

1. The server code first opens the file and saves the data in it, into arrays **id**[xxx], **name**[xxxx] and **balance**[xxxx].
2. It then creates a socket, binds it with address, and the port number. The code used command line argument to take the port number. I use a pre-defined function **htons()** to convert the **little endian** notation to network byte order(**big endian notation**).
3. After binding the server socket starts listening for Client connections.
4. Once a client connection is established the server code creates a thread for the client.
5. The thread then proceeds with creating a new socket and saving the socket data it receives from the main () function of server.
6. The new socket reads the data/transaction (being sent by the client) , compares the account ID from the transaction with the account ID’s in the array **id[xxx]** and accordingly proceeds.
7. If the account id from the transaction is found in the records, we lock the server process using a **mutex**, the thread then goes on to read the operation(withdrawal(w), deposit(d), others…) from the same transaction.
8. If the operation type is a ‘deposit’, the system deposits the amount present in the transaction in the corresponding account ID.
9. If the operation is a ‘withdrawal’, the system first checks if enough balance is present on the account, if yes, then proceeds with the withdrawal operation, otherwise sends a response back to the client, stating “Low Balance”.
10. Be it a deposit or a withdrawal, the code then goes on to update its records to show the correct balance.
11. The **mutex** lock is released once the records are updated, so that the next set of transactions can proceed.

**Updating the Records.txt for interest value:**

The server additionally runs a thread to update all the records present in the Records.txt with interests.

**How to achieve this?**

I create a thread which runs periodically to update the Records.txt with interest values.

Consider the threads: **ophandler** and **updateinterest**

In order to make the **updateinterest** thread run periodically, I make the server active and run 100 clients**(updateinterest)** on it using my script file “script.sh”. Once the thread for the 100th client runs, **a conditional variable** set on the transaction thread signals the ‘**updateinterest’** thread to run. The ‘**updateinterest’** then updates all the balances present in the record and writes back all the updated balance back to the **Records.txt**

**Steps for client:**

1. The client code takes the following arguments from the command line

* The object code for the program to run
* The IP address of the server
* The port number on which the server is present.
* The time interval with which each transaction executes
* The file “Transactions.txt” which contains the transaction details
  + A model **Transaction.txt** filewould look like below

20 110 d 500

25 115 w 800

30 116 d 5000

The first column shows the timestamp of the transaction, the second shows account number, the third one represents the operation type and last column, the amount to be withdrawn or deposited.

1. The client code first opens the Transactions.txt file, processes each transaction, and tokenizes the timestamp to generate proper delay between consecutive transactions.
2. The client then proceeds with connecting with the socket listening on the server side.
3. Once the connection is accepted, the client starts with sending the first transaction on the socket. Once the first transaction is completed, the server sends an acknowledgment to client.
4. The client then proceeds with sending the next transaction on the server.
5. Once all the transactions in the file “Transaction.txt” are processed the client closes the socket connection.

**How to run the code:**

Compilation:

The **server** code is compiled as shown below

-$root **g++ -o server qserver.cpp -lm -lpthread**

The **client** code is compiled as shown below

-$root **g++ -o client strcli.cpp -lm -lpthread**

Running the client and server:

-$root **./server 8787**

-$root **./client 127.0.0.1 8787 .2 Transactions.txt**

**3. Design trade-offs:**

1. This code saves all the records and transaction updates in a file. A file is easy to open and write to, hence the system takes minimal time to run multiple transactions. In a real-time environment, however, there are million transactions which take place. Banking industry measure a systems throughput using the term “TPS”. TPS stands for transactions per second. Average value for TPS for a bank running real time can vary from 1000-5000. In such a case it becomes necessary for the clients to send their request in a timely manner. Updating of the files with each transaction cannot be performed real time and hence should be bundled into batches and run periodically. Hence to maintain the state of the server system, we are trading-off real time data saving with increased system availability.
2. The system requires testing with various business use test cases. Given the restricted time-line for the project submission, a trade-off has been made to test the system for a confined set of test cases. This set would ensure usual transactions would be rendered without any glitches.

**4. Scope of further development:**

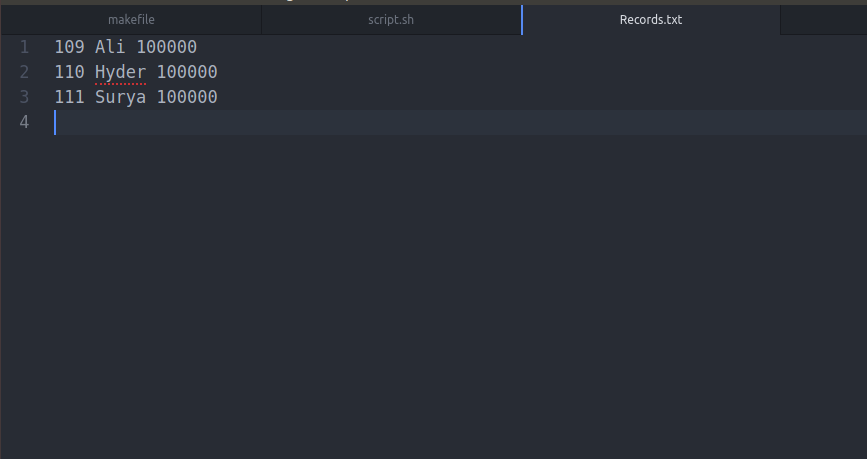
1. This system provides minimal operations which are absolutely necessary for a bank to function. Activities like periodically updating all the bank accounts with a fixed amount based on fixed rates, the ability to create a completely new record at the server side have been ignored.
2. Security: In order to maintain the authenticity of all the transactions, all the transactions can be sent after hashing them using the SHA2 algorithm. A SHA2 algorithm returns a 256-bit hash code for each transaction which can be then matched at the server side for correctness, using the same hash.
3. An Intrusion detection system can be built, which periodically measures the time taken for a standard set of instructions to run on the system and compares the values with the base values. Any large deviations in the results would suggest that there is a risk/threat to the bank server systems.

**5. Results:**

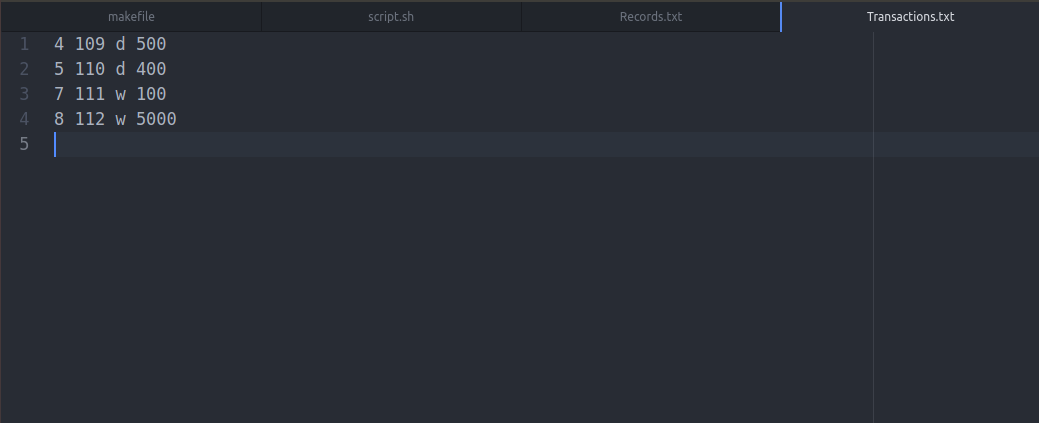
Below are few screenshots, showing some results from the testing of the system

**Test data used:**

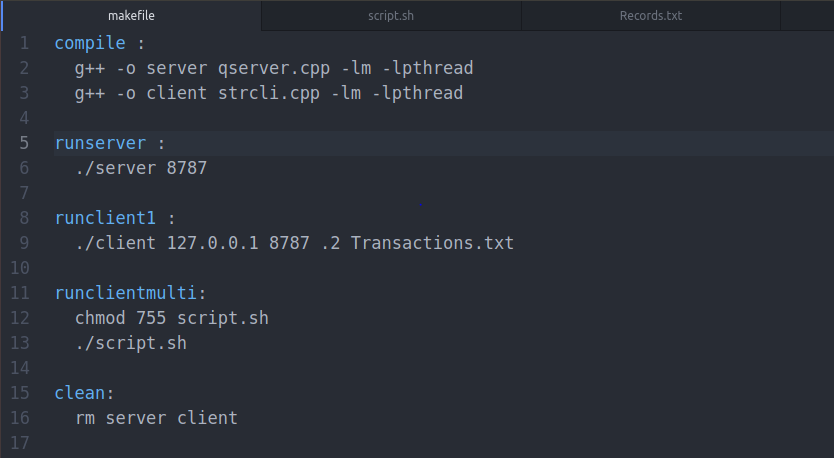
**Records.txt file:**



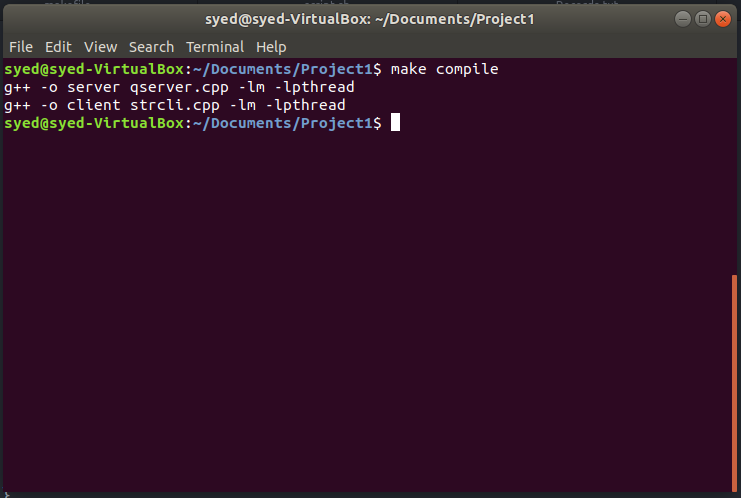
**Transaction.txt**



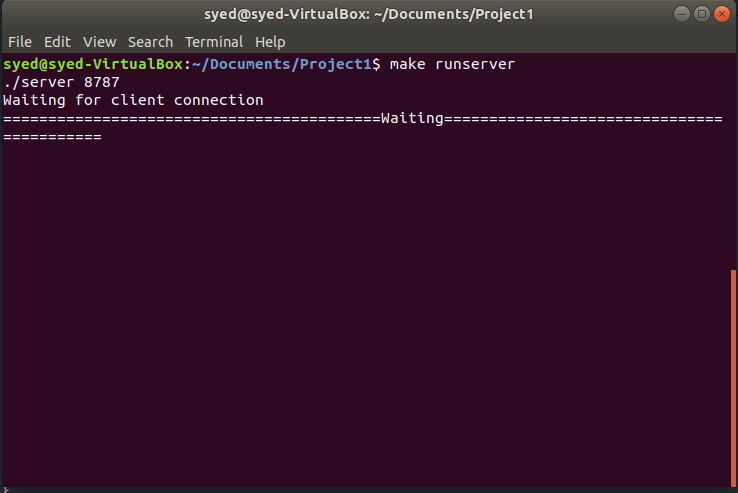
**Makefile:**

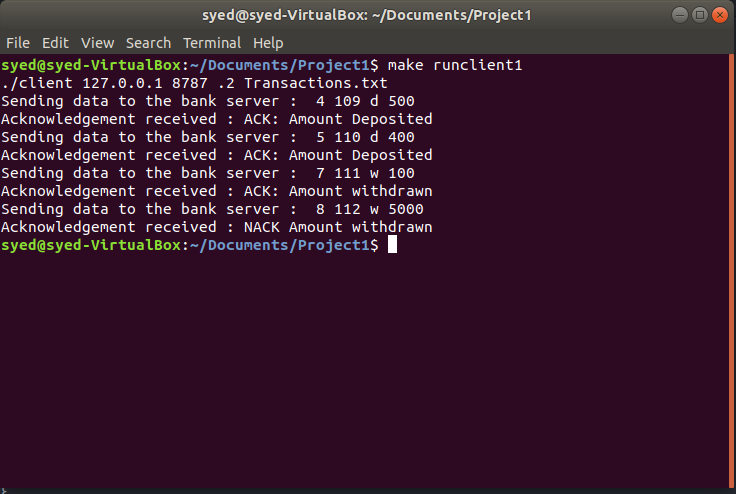


**Compiling the code:**

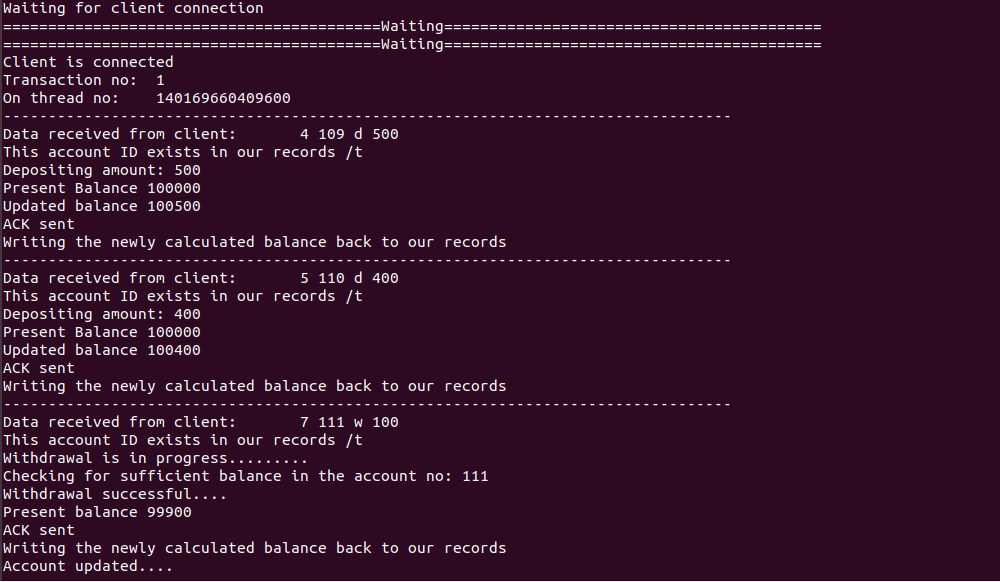


**Running the server:**

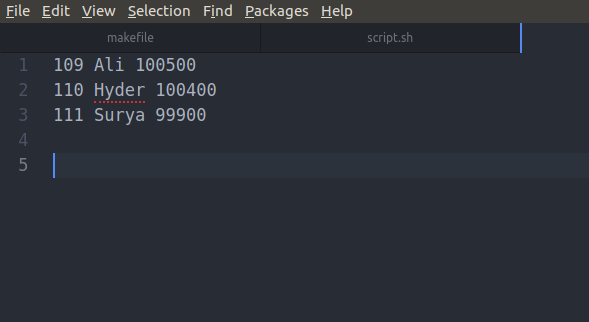


**Running the client:**

**Server screenshot after the transactions are processed:**

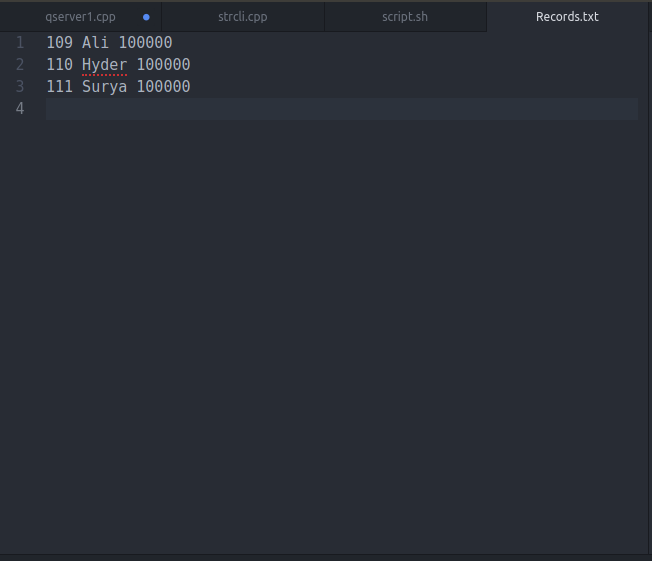


**Records.txt after running the transactions**

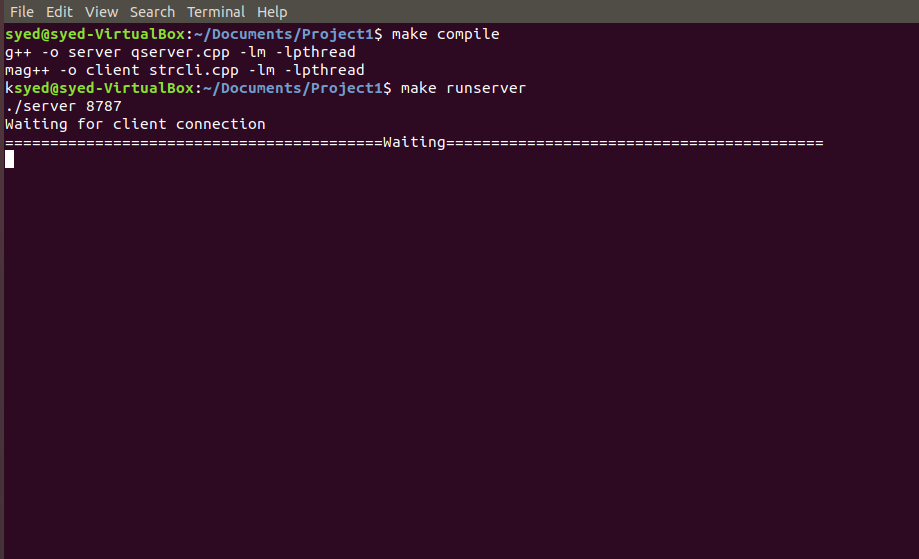


**Screenshots showing Updateinterest results:**

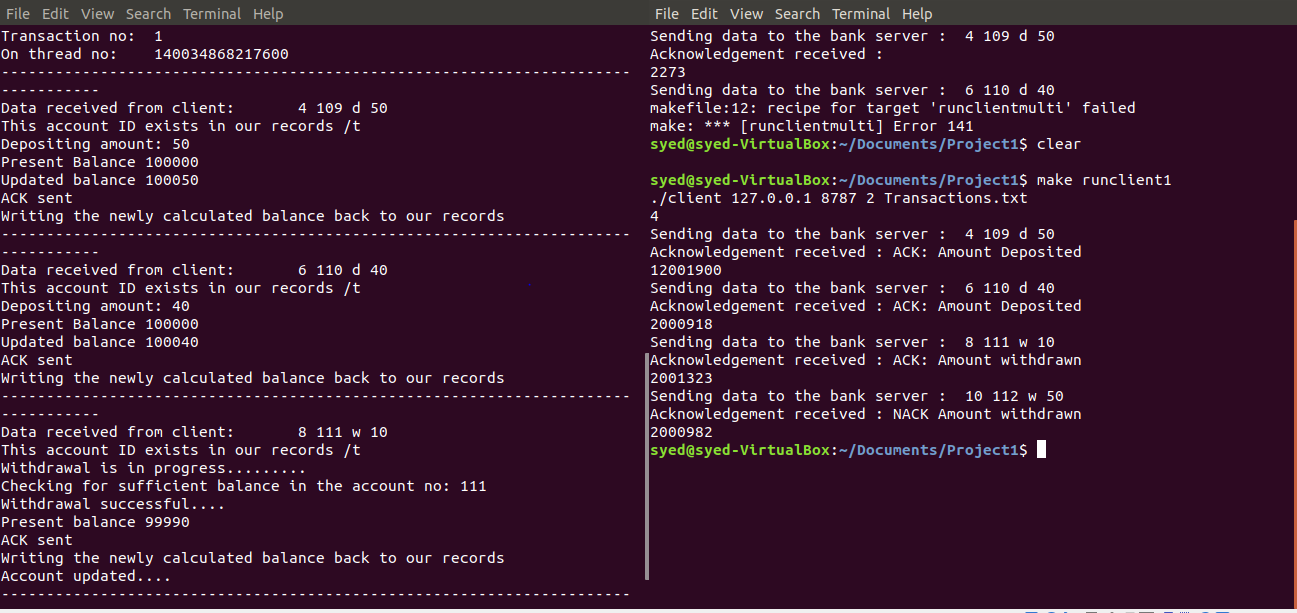
**Records.txt before running the system:**



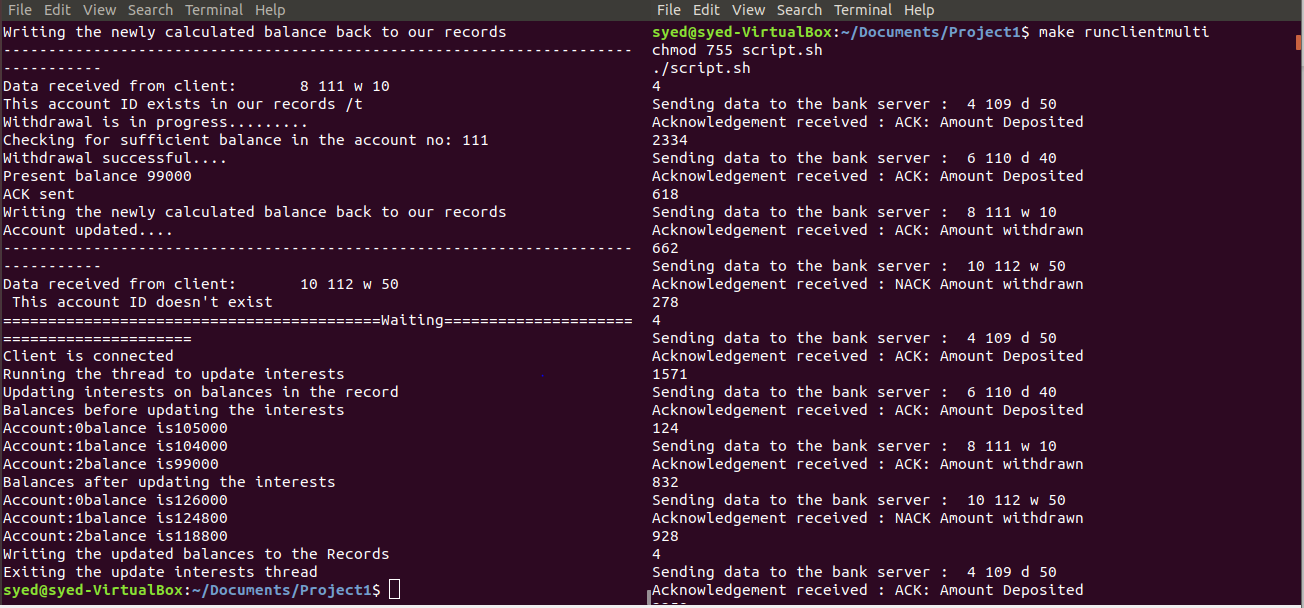
**Running the server:**



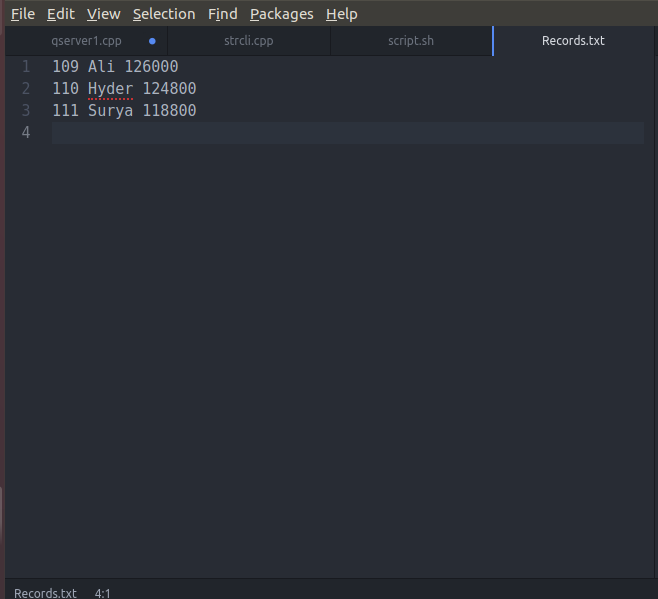
**Running the client once:**



**Running the client 100 times using the script file in order to run the updateinterest thread:**



**Records.txt after updating interests:**



**6. References:**

1. Distributed systems principles and paradigms, Tanenbaum
2. Abraham-Silberschatz-Operating-System-Concepts---9th2012.12
3. A lot of ideas have been derived from a similar implementation on Git hub whose link is provided: <https://github.com/pabitralenka/Multi-User-Bank-Account-Manager>