KHURANA NITIKA (BH42137)

Project 1 Description

Program design

The Bank server application implements the banking service whereby one server can connect with multiple clients who ask for services from the server. The server can deposit or withdraw an amount, checks if the balance is enough to perform an operation and update the records. We specify the port number over which we want to establish the client-server connection while running the server. And the server is open to connect with clients using internet sockets. Once, the server is run, it opens the Records.txt file and starts reading the records and saves the account records in data structures by filtering the account ids, account holder's names and the balance amount in the accounts.

The client instances are run by specifying the server's IP address, the port number (same as the one on which the server is listening), the time delay between each transaction and the file from which transactions are to be read. Once the client instances are run, the server accepts the connections from multiple clients and establishes a connection for providing services. The clients read the transactions that need processing from the Transactions.txt file, and start sending the transactions one by one over the socket.

The server accepts these transactions and creates a **separate thread for each client** to allow these clients to run concurrently. The server calls the handler(connhandler) for each client that is connected to the server socket. On calling the handler, the socket starts reading the transactions from each client one by one.

The handler then filters the values from the transaction records by separating the account id of the holder, the type of transaction to be performed and the amount that is to be deducted/added to the specified account id. Once that is done, it checks whether this account id exists in the records database and if it does exist, it then implements a **lock** (mutex) over the processing that needs to be followed. Having done so, we check the type of transaction to be performed. If the transaction is of "d" (deposit) type, we increment the balance for that account id by the specified amount and write the result back into the Records.txt file. If the transaction is of "w" withdrawal type, we deduct the balance amount with the specified amount and write the results back into the file. In case if any of these transactions, on successful completion of file write command, we send an acknowledgement back to the client specifying that the transaction is complete. In case of any failure, a negative acknowledgement is sent to the client. On successful send to the client, we take off the lock from that account id. This is how multiple clients or threads are **synchronized** using mutexes. Once, all the transactions for all clients are completed, the file and the server socket are closed.

The client sends the transactions at an interval which is equal to the timestamp mentioned in the Transactions.txt file multiplied by the timestep that is given while running the client program. The client periodically sends the transactions for processing by going to sleep for this interval between multiple sends.

Design trade-offs:

The current design handles multi-threading and synchronization. This improves system performance, prevents the program from infinite wait, and ensures correctness. However, when we are creating new threads, we are not implementing measures for memory deallocation so the memory that is actually allocated to the program is not entirely released. We need to take into consideration the memory leaks and release the memory once the threads are complete.

The program does not consider the creation of new accounts, deletion of existing accounts and calculate interests.

Also, when the user checks that the current balance in the account is less than what is needed to perform the withdrawal, it just sends a negative acknowledgement and does not take any further action. It picks up the next transaction from the client and starts executing that.

For the purposes of reliability, we are implementing TCP sockets. These, however, do not provide better performance than UPDP sockets and can't be used for transfer of information involving live audio, video etc.

Improvements or extensions:

The program can be further improvised to include further functionalities associated with the banking application. The server can be modified to calculate interest, create new records in the Records.txt file, delete existing records, etc.

We can further extend the program to perform some action once the transaction fails. We can calculate some fine of some sorts for that account if the account balance is less than what is needed for a withdrawal. The system can also be modified to define a minimum balance that needs to be present for each account.

We are currently implementing one thread for each client. All the transactions for every client form a single thread. This can be modified to further divide these transactions among multiple threads. This would provide better usage of cache storage by utilization of resources and better utilization of CPU and decrease maintenance costs.

Running the program:

We first execute the makefile to compile our Bankserver and Client programs. We issue the command:

make compile

If there are previous output files present, we first delete all the .o files using **make clean** and then issue the make compile command. Then after compiling both the programs, we run the server program by executing:

make runserver

Then after running the server we need to run our script, like the one I have written in the manner:

./script.sh

After running this command, our script will start running and display the messages being sent across from the client to the server and vice-versa. We then calculate the time taken to complete each transaction by each client for the two cases:

1. When the clients are increased
2. When the transaction rate is increased marginally