online banking management system

About

The project aims to develop a banking system that is user-friendly and multi-functional.

Database design

The database must have accounts, transactions and customer details. These details are to be stored in a file based database example hands-on 1 ticket question (17).

Necessary details for accounts:

1. Account number (should be unique, can be character array or an int)
2. Balance (must be a double, 2 decimal precision)

Necessary details for customer:

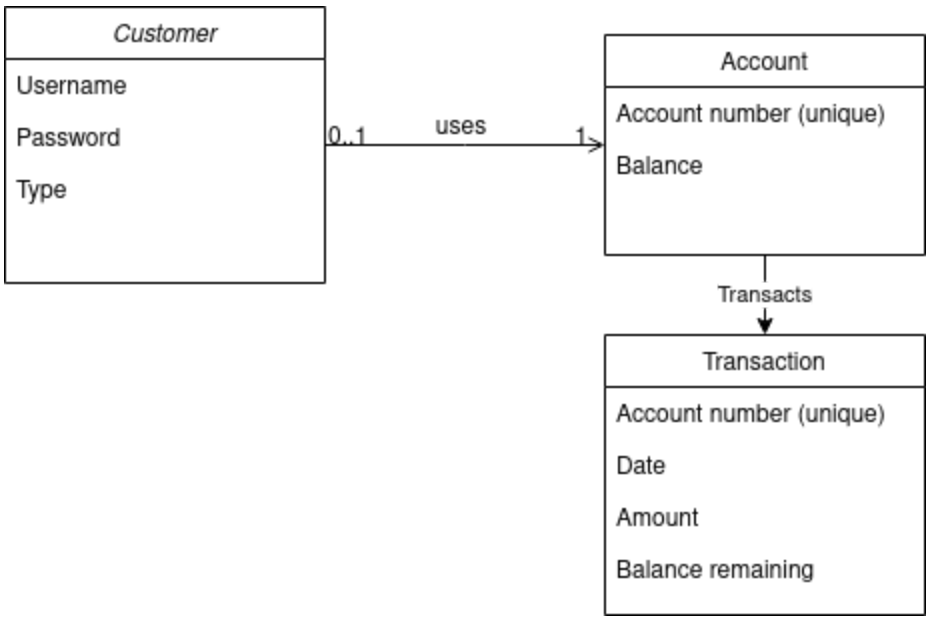
1. User name (should be unique, character array with respectable length)
2. Password (character array with respectable length)
3. Type (can be bool or character array, different bool value for admin and normal user)

Necessary details for transaction:

1. Date (should be a character array, no epoch values)
2. Account number (should be one you added in accounts)
3. Amount (double, credit or debit)
4. Balance remaining (double, remaining amount for that account)

*These are only necessary details one must have in their database, rest you can twist things based upon your requirement.
All the major operations are to be done purely using system calls.

Things to be taken care while making database:



So how would you connect Customer with Account?

1. Well one way could be you can scrap Account all together and put account number under Customer table and balance too.

```
struct Customer{
    char username[1024];
    char password[1024];
    char account_number[1024];
    double balance;
    bool type;
}
```

2. Second approach could be making another account structure (advised) and connect two tables using common entity account number, why account number (thoughts?).

```
struct Customer{
    char username[1024];
    char password[1024];
    char account_number[1024];
    bool type;
}
struct Account{
    char account_number[1024];
    double balance;
}
```

Well this should be enough for task admin access and normal user access but what about a joint user account?

1. You may assume an account can have max of two users, then how would things change here in the database. Example

Username Light Yagami accesses account number KIRA457248624 and username Ryuk can access the same account number KIRA457248624.
*This account number should be unique and different from other customers account number and same with the username.

Now coming to the implementation, how would things be changed in the database.

```
struct Customer{
    char username1[1024];
    char password1[1024];
    char username2[1024];
    char password2[1024];
    char account_number[1024];
    bool type;
}
struct Account{
    char account_number[1024];
    double balance;
}
```

But here you see even if an user doesn't have a joint user, you are literally wasting about 2048 bytes on your database and don’t advise this one. (Gotta respect the harddisk).
Any ideas how can you make things better? Think.
Can we do better? wink wink.
Well this now settles normal users, joint user and administrator.

Administrator (With great power comes great responsibility)

Well an admin can add, delete, modify and search for specific account details.

1. Admin can create a username and password for any customer and relatively creates an account for the same user.
2. Admin can delete an account or customer (a customer cannot exist without an entry of account). If a specific account gets deleted you will have to delete the customer too and vice-versa. Delete from the database. (deleting is little tricky, so think out of the box)
3. Why I say deletion is tricky: well if you delete a particular record then you have to move every other record, so think of an alternate way.
4. When an admin chooses to delete an account, you can go forward to delete that account and its related customer or customers too.
5. Example: If you delete account number KIRA457248624, then Light Yagami and Ryuk should both be deleted.
6. If the admin chooses to delete a particular customer and if its related account is not a joint account then you can delete the account and customer straightforwardly. But if it's a joint account then the joint user can still persist that account.
7. Example: If you delete username Light Yagami for account number KIRA457248624 and Ryuk is joint user to that account, then Ryuk should still be able to access the given account.
8. Always prompt for a reconfirmation if a deleted account does have some balance standing.

9. Coming to the modification part, admin can modify users password and if a current user wants to be a joint account. Here you can always prompt the admin to enter a joint user username and password.
10. Finally the search part, look for a particular account number or username in your database file and print the relative details.
11. Remember all the admin operations are to be done from the client side only. You can differentiate between admin and normal user based on type.

Here's an incomplete example when you are on the client side.

```
shubham@shubham:~/Documents/IIITB/2020-SystemSoftware/MiniProject_Banking$ gcc client.c
shubham@shubham:~/Documents/IIITB/2020-SystemSoftware/MiniProject_Banking$ ./a.out
-----------------------WELCOME TO BANKING SYSTEM-----------------------

Kindly Enter your credentials

Username: light
Password: 1677
-----------------------Welcome light-----------------------

Connection Closed, Good Bye
shubham@shubham:~/Documents/IIITB/2020-SystemSoftware/MiniProject_Banking$
```

Figure 1: Non admin user logs in.

```
shubham@shubham:~/Documents/IIITB/2020-SystemSoftware/MiniProject_Banking$ ./a.out
-------------------------WELCOME TO BANKING SYSTEM-------------------------

Kindly Enter your credentials

Username: admin
Password: 1677
-------------------------Welcome admin-------------------------

Press 1 to add an account
Press 2 to delete an account
Press 3 to modify an account
Press 4 to search
Press any other key to Exit
Enter your choice: 1
Enter following details

Username (UNIQUE): barryallen
Username already taken :( Please try again

Username (UNIQUE):
```

Figure 2: Admin logs in and creates a user but damn not unique.

```
shubham@shubham:~/Documents/IIITB/2020-SystemSoftware/MiniProject_Banking$ ./a.out
------------------------WELCOME TO BANKING SYSTEM------------------------

Kindly Enter your credentials

Username: admin
Password: 1677
------------------------Welcome admin------------------------

Press 1 to add an account
Press 2 to delete an account
Press 3 to modify an account
Press 4 to search
Press any other key to Exit
Enter your choice: 1
Enter following details

Username (UNIQUE): barryallen
Password: 1677
Welcome, barryallen. Your account number is barr1603016969 and your balance is Rs 0.00

Press 1 to add an account
Press 2 to delete an account
Press 3 to modify an account
Press 4 to search
Press any other key to Exit
Enter your choice:
```

Figure 3: Admin user logs in and creates a user.

Similarly you can implement delete, modify and search segments for the admin. Do remember only admin be able to see these options.

One can try to implement password which is hidden, example
Username: clarkkent
Password: *****
Or you can go the linux way where the password client enters the user cannot see but surely your server processes it.
(some extra points to gryffindor)

Customer - Paisa hi paisa hoga

After a customer successfully logs in, lets credit debit some money.

1. Deposit - For the logged in user credit some money to the balance, the same entry should be produced in the transaction database. You can credit money upto two decimal places. Example credit 420.69 Rs
2. Withdraw - For the logged in user, the user can debit some money to the balance, the same entry should be produced in the transaction database. While debiting, always look if that kind of money is available in the account. You can debit money upto two decimal places. Example: debit 666.99 Rs only if balance is more than that.
3. Balance Enquiry - For the logged in user, print balance for the user on the client side.
4. Password change - A logged in user can change the password from the client side.
5. View details - let's modify this terminology, for this please print the account passbook or transaction statement for the logged in user. This should print the transaction table for the user account. This should include date, account number, balance and debit/credit amount.
6. Exit - damn can’t explain this.

Implementation

Implementation is the time consuming portion over here. I found two solutions to this problem, so let's discuss the easier one.
But before that lets come to the socket programming, you got to implement the similar to what we implemented in 34a (hands-on two, yes concurrent programming)
Here's a snippet of the server code.

```
main(){
    struct sockaddr_in server, client;
    int socket_desc, size_client, client_desc;
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);
    if(bind(socket_desc, (void *)(&server), sizeof(server)) < 0)
    {
        perror("Error on binding:");
        exit(EXIT_FAILURE);
    }
    listen(socket_desc, 5);
    while(1){
        size_client = sizeof(client);
        if((client_desc = accept(socket_desc, (struct sockaddr*)&client,
&size_client)) < 0) {
            perror("Error on accept:");
            exit(EXIT_FAILURE);
        }
        if(fork() == 0){
```

```
            // verify credentials for the connected user and further
stuff here
            close(client_desc);
            exit(EXIT_SUCCESS);
        }else{
            close(client_desc);
        }
    }
}
```

Here's the snippet of the client code

```
main(){
    struct sockaddr_in server;
    int socket_desc;
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);
    connect(sock_desc, (void *)(&server), sizeof(server));
    while(1){
        // do you client stuff here
    }
}
```

One important note: all the transaction handling and user verification is to be done on the server side. Strict no to client side operations.

So what's the stuff, how to approach?

The easy way -

1. So can we write all our if else ladder or switch cases on the client side.
2. So for every input from the client we can mirror the code written on client side to the server side. For every write on the client you can mirror a read on the server side.

```
// client
write(1,"Choose an option : ",sizeof("Choose an option : ")); Step 1
scanf("%d",&choice); // I choose to credit some money, Step 2
write(sd,&choice,sizeof(int)); // write to the socket Step 3
// now we know the server is in apt function, we can perform operations
write(1,"How much money: ",sizeof("How much money: ")); Step 7
scanf("%d",&credit); // Step 8
write(sd,&credit,sizeof(int)); // write to the socket Step 9
// read what server sent and notify the client // Step 14

// server
read(sd,&user_choice,sizeof(int)); //Step 4
// check what user chose, oh user chose to enter credit money Step 5
// jump to the appropriate function on account, Step 6
read(sd,&user_credit,sizeof(int)); //Step 10
// apply record locking if joint account Step 11
// update the balance after searching appropriate account Step 12
write(sd, "SUCCESS", sizeof("SUCCESS") // Step 13
```

So this particular pattern we have to follow throughout the mini project and this is one such approach.

The tricky way -

Well I won't tell you that, but here's the hint - you can write the same code on the server and share the stuff on the socket to the client but real challenge comes when you have to tell the client that this message from the server is asking for user input or is simply for the display on the client side.

Benefits of this style is that your client side would be abstract and really the client can't see any flow and of course the number of lines of your entire project will be really less, saving some time.

But these implementation documentation should not restrict you from other implementations. You're free to try, experiment with the flow as far as it suits our project requirements.

But I see some file locking, what's with that?

1. Well for any operation on databases, you got to implement file locking.
2. So say Light Yagami from the above example tries to credit some amount to the account. While Ryuk tries to debit some amount from the same account because Light and Ryuk are joint account holders.
3. So during this you have to lock the record of that account in the database while Light is making some transactions and Ryuk will wait meanwhile.
4. So such cases you can keep in mind while implementing.

Verify credentials -

1. You can use the brute method to check a password for the unique username in the file. If found you can go forward with the operations else throw an error and quit the connection.

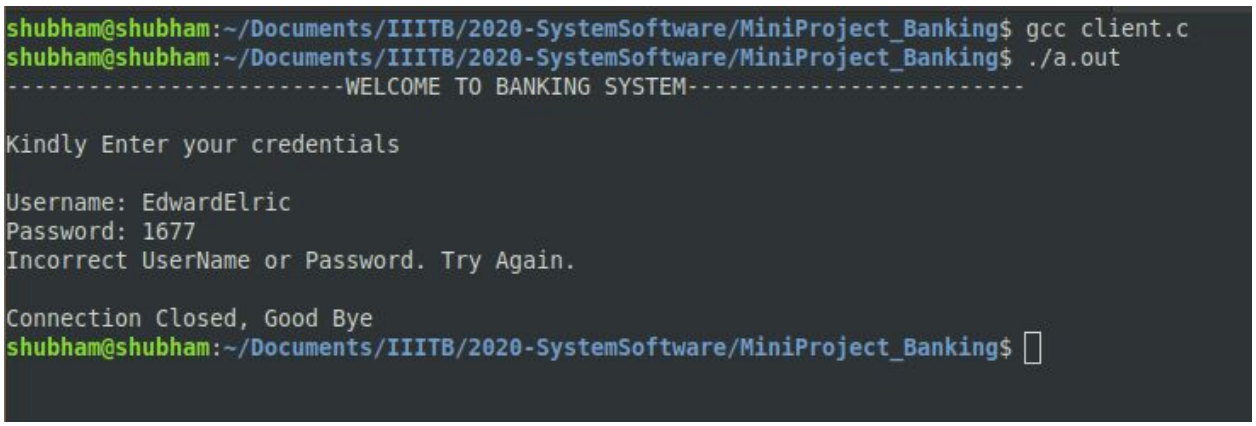


Figure 4: For an incorrect username and password

So this is it, this should give you a head start, but if I missed out on something you can always contact me,
and again I repeat this guide should nowhere stop you to perceive things differently, another approach is always welcome.

This project would test your learnings in socket programming and file locking, rest are simply basic read write system calls.
If you want to add different functionalities like :

1. the same user cannot login if there's already a session active for him/her then you can add that too.
2. What if the server fails how would you handle a connected client and
3. say the client was in the middle of a transaction so rollover mechanism, you can work on that too
4. And if the user is not active for say 10 minutes you can logout/exit for the user, but these are optional and you can implement if you feel like.