Bank System
Note: Extra Credit 0 was attempted.

The server and client both use port number 9709. The client must connect to the IP address where the server process is currently running. For testing purposes, this will be localhost.

Server process:

On startup, the server process will simply appear blank with no status messages as at this time it will currently be searching for incoming client connection requests.

Upon receiving its first connection request, and all following, it will print, if successful, that it accepted and connected to this new client

Every 20 seconds the Bank Server State will be printed to the server's terminal window displaying information on every account currently stored in the bank (Note: There may only be a maximum of 20 accounts at any given time).

The server stores bank data in system memory using two different structures

AccountInfo structs:

These structs store information on the account name, the account balance, and a flag to indicate whether or not the account is currently in session.
As per the assignment instructions, we wanted to store only this pertinent data in their own, self-contanied, data structures to maintain data integrity.

BankEntry struct:

As mentioned above, we wanted to keep the pertinent data contained in their own structures as per the assignment instructions.
So, to keep the AccountInfo structs themselves organized within a larger bank structure, we made BankEntry structs to store information about an AccountInfo struct (the account entry that corresponds to this entry within the bank) and whether or not the BankEntry is currently taken (that is, whether or not an account has been created in that particular space in the bank).
Note that the Bank structure, as per assignment specifications, is only capable of storing information on 20 accounts
So, to store these accounts, we simply allocated heap memory for an array of the size of 20 BankEntry structs, with each of these BankEntrys containing information on their corresponding accounts.

A pointer to the Bank data structure itself is stored as a BankEntry pointer variable with global scope so that it may be accessed at any location within the server process.

The server stores information about each client session's active account (as each client session is allowed one active account at a time – although each account may only be considered active in one client at a time – implying that the server may only recognize 20 accounts which are c onsidered active in a client session at any given time) in ActiveAccountInfo structs.

ActiveAccountInfo structs:

These structs store information on an account, and its entry number (that is its BankEntry position within the Bank array).

Each client session will have an ActiveAccountInfo variable called ActiveAccount which can either hold the value NULL (no account active) or an ActiveAccountInfo struct representing the account, and its position within the Bank data structure.

This information is useful because it allows us to use pthread_mutex_lock and pthread_mutex_unlock commands on pthread_mutex_t variables specifically locking a specific account withihn the Bank data structure.

For example pthread_mutex_lock(mutex_account[ActiveAccount->entryno]), or some similar call, could lock an account based on its position in the bank data structures

The last structure implemented is the ClientSessionInfo structure

ClientSessionInfo structs:

These structs store information on a client session's pthread_t handle and the socket file descriptor used to communicate with this client

These are stored in an array ClientSessionInfoList of size numclients (increases with each new client connected) which will prove invaluable upon server process termination wherein we must join each ClientSession thread to ensure every client has been disconnected properly and is notified of server shutdown.

Overall structure of server process:

The server process contains three main thread types:

After socket creation and initialization, a SessionAcceptor thread and a DisplayAlarm thread will be created.

The SessionAcceptor thread will continually search for new client connection requests and will put each successful client connected into its own ClientSession thread. This SessionAcceptor thread will continue until the server received the SIGINT signal sent by Ctrl-C. This thread is joined as one of the final steps of server shutdown after all individual threads have been joined.

The DisplayAlarm thread is responsible for alerting the server to print the Bank Server State every 20 seconds. This thread is detached as we are not concerned with joining it upon server shutdown.

Each ClientSession thread is responsible for handling the session of each client that has connected to the server. These threads by definition run concurrently and thus implement various mutexes to ensure that no ClientSessions “step on each others toes” while adjusting bank data. Each ClientSession thread is joined when SIGINT is sent to the server process by Ctrl-C.

Client process:

The client takes one command line argument: ipaddress. This is the ip address that the client will use to create a socket to attempt to connect to the bank server process. The port number is always set to 9709.

If the client cannot immediately find a server process located at this address/port number it will re-attempt the connection attempt every 3 seconds and prompt the user that it is doing so.

Upon successful connection 2 threads are spawned: ReadFromServer and WriteToServer

The Read thread is responsible for reading server messages (success, error, and informational) as they are sent.
The Write thread is responsible for sending the server messages whenever they are available from stdin.

Both of these threads will terminate upon 2 eventualities:

- The server has shutdown:
  - In this case, the client will be alerted of server shutdown and will proceed to shutdown itself.
- The user used the “exit” command or pressed Ctrl-C in the client window:
  - In the case the user entered exit, the client will receive a confirmation message from the server that it has been successfully disconnected, prompt the user that it is shutting down and then shut itself down. The server will close any active accounts held by the client.

    In the case the user pressed Ctrl-C, the server will recognize that the client unexpectedly shut down without using the exit command, end the client's ClientSession thread, and close it's active accounts.

After these threads terminate, the client process successfully shuts down.

This concludes the readme of the Bank System executables, client and server.