CLIENT:

For the client implementation, we went with a straightforward approach. We made space for two pthreads, and created the threads to handle the read and write respectively. After thread creation, we just made the main thread join the pthreads. In order to exit the client program, the user types in exit. When the user enters ‘exit’ the write thread changes the global variable ‘end’ to 1. Once ‘end’ is set to one, both the write and read threads return and the program executes safely. Another safeguard we put in place was created two global variables: ‘thread0Active’ , ‘thread1Active’ which are set to 1 when the write and read threads are active.

In order to ensure that a valid socket is passed to connect, we call a function buildSocket() which returns a new socket. This is because a failed connect puts the socket into an undefined state. We modified the professor’s error function to join on the read and write threads if they are active before exiting the program.

SERVER:

MAIN:
-sets up the bankthread and session_acceptor thread
-creates the simple list to store the client acceptor thread ids
-sets up the SIGINT signal handlers
-starts the bankthread and sesion_acceptor thread
-destroy the simple list

SIGHANDLER:
-handles SIGINT
-sets the global variable thread_exit to TRUE
    - thread_exit tells all the other threads to safely return
-closes the server_sock which is the socket used by the sessionAcceptor function to accept client connections.
    -this is done so that accept() fails and the sessionAcceptor thread can proceed with variable cleanup

SESSIONACCEPTOR:
-handles client acceptance
-sets up the server socket
-listens on port then enters loop which breaks when thread_exit != TRUE
-accepts a connection then creates a pthread which is used to communicate with the accepted client
-appends client_thread id to a simple list in order to join for later
-when finished, it joins all the client_threads spawned

BANKTHREAD:
-thread solely dedicated to printing out the account information
-if an account is active then the account variable active = TRUE
    -prints out in session if active

CONNECTIONHANDLER:
-handles actual client server interaction
-pretty basic implementation
-the thread is joined in the sessionAcceptor thread

Bank :

The account creating and handling , is done in a bank class , which allows us to keep the solution simple.

The various commands as per the requirement are implemented

Bank structure simple contains a 20 elem wide array , with each elem having a balance , name and an inuse boolean. The inuse boolean is used to ensure that the account is

Open account

Sets up an account and create a session with that account.

In order to fulfill the instruction of the simultaneous account not being opened , we use a mutex lock to ensure that the accounts are opened in a thread safe manner.

The main reason we do this is to ensure that the #accountsUsed varible is accessed in a thread safe manner. We do not want to create situations where multiple threads open the account and the numAccountsUsed variable is not properly updated.

Start account

This is not protected by mutex and there is no change for a race condition. Each thread was a particular bank account number and only access that particular account. If a thread is using an account, no other thread can start accessing that particular account , since we keep a boolean associated with each account. This boolean helps ensure that the accounts are accessed by one one thread at a time.

Credit amount :

This adds or subtracts the amount based on the sign. There are no limits to the amount that can be added or deduced.

Debit amount :

This adds or subtracts the amount based on the sign.

Better Explanation :

Credit always happens, regardless of balance:
This means that credit 50 causes balance to increase by 50.
credit -50 causes balance to decrease by 50.
So credit can make balance negative..

Debit:
debit 50 causes balance to increase by 50

debit -50 causes balance to decrease by 50 ONLY IF BALANCE >=50