Here is how our program works (we did the XC multiprocessed version):

Client:

Every three seconds, the client tries to connect to a server.
Once it connects, it launches a response_output_thread and command_input_thread.
In our main function, we join on the command_input_thread to ensure that our program runs as long as there is input. We do NOT join on the response_output_thread as we want to server to be able to write messages at anytime.

After these two threads are launched there is not much to be done in the client thread but read and write inptut.

Server:

Our main method sets up our shared memory and signals and timers.
We set up a socket descriptor to accept all incoming connections in our parent function.

After we accept a connection and get a file descriptor, we fork and check if we are in the parent or in the child.

If we are the parent we close the file and continue the loop.
If we are the child we set the alrm signal handler to sig_IGN, and move on to our client handling method.

We use two parsing methods to parse the input, take action, and display the appropriate outputs (client_session_actions and parse_request). Note that the output can sometimes get messy on the client side due to stdout delay

Memory/Semaphores/Signal Handlers:
In our shared memory we have a bank struct. This bank struct contains a semaphore with initial value of 1, an array of 20 accounts, and int num which is the number of current accounts.

Within each account field we store another semaphore, balance, name, and flag which tells if it is currently in session.

The bank struct's semaphore is usually unlocked. The only methods that try to lock it are create_account and printinfo, and serve_account. Each of these methods are guaranteed to only hold the semaphore for brief period of time.

We have three signal handlers. We catch the following signals:
SIGCHLD: When sigchld is caught we simply do a wait in the parent process to reap the children.
SIGAlrm: When sigalrm is called we print our bank information and reset the alrm
SIGINT: We thought it was logical to have a sigint handler in our function. If you ctrl c out of the server we can just send a message to the client processes to shut down. In our sigint handler we also clear out all of our shared memory.

Functions:
For each of our function calls there are a series of checks to ensure that you are allowed to do the command you inputted.

Whether a process is currently engaged in a customer service session or not is shown by a global variable int flag that we call inservice. This inservice flag is set to 0 at the creation of the

account, but once it successfully starts serving an account it gets set to 1. It gets reset to 0 during an end or quit call.
Thus at the beginning at most of our functions we have a check to see whether inservice is true so we can confirm the client allowed to execute a certain command.

In order to create a new account for our bank, we lock the bank semaphore, perform the update and then unlock our semaphore. In our account creation we initialize the semaphore for the account, as well as the account name . We do some checks to see if the bank has 20 accounts already or if the account already exists.

To printinfo every twenty seconds, we catch a sigalrm signal and call a printinfo method from the signal handler and reset the alarm. In the printinfo method we lock down the bank semaphore , print our info and then unlock it.

Note that there is no risk of new accounts being created during the print we need to secure the bank semaphore in both the print and create_account function (and the semaphore only has a count of 1)

Our next functions utilize the individual account semaphores.

We control our customer sessions in our serve_account and end methods. Every time a client tries to serve an account it first has to lock the account semaphore corresponding to that account. The client continously sem_try_waits for this semaphore until it can lock it down. During the try-wait period messages are sent to the client about how he is currently trying to lock the account (XC part). We put a delay in our try_wait loop as to not sink too much processing power into the task. Once the client locks down the account semaphore the client is free to withdraw, deposit and query until the client wants to stop.

Customer sessions last until prompted by end or quit. In either case, end or quit releases the semaphore which is then picked up by a waiting process or kept unlocked and waiting.

At the end we free everything clean up our shared memory and close the socket descriptor in the parent function.

The End:
We had a lot of fun with this assignment. This assignment really opened our eyes to how thoughtful and careful you have to be with multiprocessing and shared memory. We spent hours debugging some simple errors that could've been solved with a bit of forethought on conceptually what needed to happen. We tried fiddling with the stdout a lot but it is really laggy to print detailed messages.

We hope this readme made it a lot easier to understand our code.

Thanks :)