Aditya Geria (aag177, section 3)

Tanya Balaraju (tb463, section 3)

PA5: Multithreaded Bank Server

==========

SECTION 1: Overview:

==========

This implementation of the bank server and client module uses a multithreaded, multiprocessed and memory mapped solution to the problem.
The information for any given account is stored in struct account, which is memory mapped to a file "bank". Also, a new process is spawned for each client that joins the session, while the parent continues to accept new sessions.

==========

SECTION 2: Some Notes on Executions

==========

Server – the server will print debugging messages in the middle at some points. These include things like the PID, the name of the process, starting thread messages, "Token" messages to check whether the token was read in correctly, command completion/execution messages. All of these are left in for a few purposes. The grader can easily determine where the program is going wrong, and deduct the appropriate points (and not assume that the code entirely doesn't work); also, we can determine where/how the program failed. Also, the server may tell the client that they need to re-connect after an unrecognized command. This is left in to simulate actual log-in log-out conditions. For example, after you attempt to start a session that is already in progress, you will need to restart the client (just as you would need to re-log in). The server messages must also be monitored to ensure optimal, intended execution.

Client – The client will display the messages retrieved from the server (may be split up due to length). Also, the client MUST be responsible and enter a "finish" or "exit" command after starting a session. This is again, to simulate actual log-in conditions. For example, if you are logged into your email, close the tab and reopen it, you will already be logged in.

Makefile – use "make bank" to create an empty bank file. Use "make cleanmmap" to delete the current bank. Use "make" to clear the client and server executable, then create a bank file, and the client and server executable.

==========

SECTION 3: Server Algorithm

==========

First, the client-server communication is set up using a mixture of the method **getaddrinfo**(), and the system calls: **socket**(), **setsockopt**(), **bind**(), **listen**() and **accept**(). At this stage, the "**bank**" file is memory mapped to contain an array of accounts. Also, a new thread is spawned to handle printing the entire bank at 20 second intervals. After we have acquired a valid socket descriptor, we use **fork**() to allow the parent process to continue accepting sessions, while the child enters the **client_service** method with the accepted connection.

Additionally, in the print function, the bank_locks **mutexattrs** are set to allow it to work with shared memory (pthread_mutexattr_setpshared(…)) The print function first trylocks() the bank_lock, then prints the info, then unlocks the **bank_lock**, and then sleeps for 20 seconds. The print function prints out the entire bank (all 20 accounts) at once. As a result of a new tax rebate granted by the government, all new accounts start with $60!

For any credit/debit/balance commands, a mutex lock with initialized attributes **accounts[i].acc_lock** is locked, and unlocked after changes have been made or info has been retrieved.

In **client_service**, a command is read in and properly formatted. First, we check for open/start/exit which are the only valid "opening" calls one can make. Once those are found, and the appropriate action completed (will be explained in depth), it moves to look for the next set of commands: balance/credit/debit/exit/finish. Once those are found, it completes the corresponding action.

In the case of SIGINT, a signal handler is set up in main such that it writes "SIGINT" to the client, which immediately seizes operation with a "thank you for using" note. In the case of SIGSEGV, we close up the socket and file descriptors, unmap and exit properly to avoid unexpected behavior.

==========

SECTION 4: Server Methods

==========

Open – checks if account numbers is at 20 and denies access if so. Otherwise, mutex lock the bank_lock, add the necessary account info at index accnum (account number), and then unlock the bank_lock.

Start – First, check if the account is in the bank at all through bank_lookup – which takes in a name and returns the account number (index). Then, if found, check if its in session. If not, mutex lock the bank, start the session, unlock the bank, and return.

Balance – Look up the account by using bank_lookup(). Check if the index exists. If it exists, then place a mutex on that account. (accounts[index].acc_lock). The mutexattrs have already been initialized for this by this point for acc_lock. Then, write the balance to the client using sprintf and changing it to a char[] buffer. Lastly, unlock the lock and return.

Credit/Debit – these work the same way, and very similar to balance. Retrieve the index, check if valid, lock the account (acc_lock), make changes to account balance, use sprint to turn into char[] and write to client. Then unlock and return. The debit method will block attempts to withdraw more than you currently have as a balance.

Finish – this will nullify the name char[] and set the IN_SESSION flag to 0. Allowing you to input more open/start commands

Exit – Sends a SIGINT code (as a char[], not a signal) to the client telling it to stop execution, ends session for any in session by the client.

==========

SECTION 5: Client Algorithm
==========

The client uses a multithreaded approach, as described in the pdf. One thread is used to constantly read any responses from the server (server_response), and the other thread is used to send commands (every 1-2 seconds) to the server.
The server response thread will cause the client to exit when "SIGINT" is received from the server to the buffer in the client.
The client response thread will also close at this point.

The client response thread will send commands to the server using write() every 1-2 seconds and allow the server_response to catch the response.

The client does not use mutex locks or thread synchronization.

==========

SECTION 6: Extra credit

==========

We have successfully attempted both the **multiprocess** and the **mmap**() portion of the extra credit. The accounts are memory mapped to a file called "bank", and info is retained between executions of the bank. This is done by first opening the file with an open() call, then lseeking to a point that gives us enough room to write account info, then writing an empty string to that point. Lastly, the accounts are mmapped at offset 0.

The multiprocess portion of this comes from the **fork**() in client-service and session acceptor. The child process will handle the client service for any incoming clients. The parent process will continue to accept sessions with **accept**(). The PID for both is shown upon their **fork**().

The bank_print function is a result of **pthread_create**. However, this could also be worked in as a process.