CS214 Project 3: Multithreaded Bank System

Abstract
For this assignment, you will deviate from the codebase you've been maintaining throughout the semester in order to develop a system to exercise your knowledge of threads, synchronization, signals and sockets by having them all work together in a simulation of a text-based online banking system.

Introduction
For this project you will write a server program to emulate a bank and a client program to communicate with it. The server program will need to support multiple simultaneous client TCP/IP connections, which will require multithreading. Multiple clients accessing the same account database at the same time will require the use of mutexes to protect all shared data. The server will also globally lock the account data at regular intervals to print diagnostic information to its STDOUT, ostensibly for the bank's network operator to monitor server status. The server will not take any commands once it is running and should be able to shut down gracefully by catching the exit signal sent by Ctrl+C.

Your client will be fairly simple. It will connect to the server, read in user commands from STDIN, send them to your bank server and write the server's responses to STDOUT. The client side should additionally check the commands the user is entering to make sure they are correctly formed and formatted.

Methodology
a. Server Operation
Your server process should spawn a single session-acceptor thread. The session-acceptor thread will accept incoming client connections from separate client processes. For each new connection, the session-acceptor thread should spawn a separate client-service thread that communicates exclusively with a single client connection. You may have more than one client connecting to the server concurrently, so there may be any number of client-service threads running concurrently in the same server process.

The bank server process will maintain a simple bank with multiple accounts. There will be any number of accounts. Initially your bank will have no accounts, but clients may create accounts as needed. Information for each account will consist of:

- Account name:
  a string up to 255 characters long)
- Current balance:
  a double-precision number
- In-session flag:
  a boolean flag indicating whether or not the account is currently being serviced

The server will handle each client in a separate client-service thread. Keep in mind that any client can create a new account at any time, so adding accounts to your bank must be a mutex-protected operation.

b. Server Diagnostic Output

The bank server has to print out a complete list of all accounts every 15 seconds. The information printed for each account will include the account name, a tab, current balance, a tab, and "IN SERVICE" if there is an account session open for that particular account. New accounts can not be created while the bank is printing out the account information. Your implementation will need to use timers, signal handlers and semaphores to accomplish this.

In order to interrupt all current connection threads and print out the service information, you can set a special interrupt timer (with setitimer()) to cause a SIGALARM every 15 seconds and a signal handler to do the diagnostic printing. Since the signal handler needs to lock the database and is running alongside threads, it needs to use an asynchronous threadsafe synchronization mechanisms, so you'll need to use a semaphore to lock your account data while you're in the signal handler.

c. Client Operation

The client program requires the name of the machine running the server process and port as a command-line argument. The machine running the server may or may not be the same machine running the client processes. On invocation, the client process must make repeated attempts to connect to the server. Once connected, the client process will prompt for commands. The syntax and meaning of each command is specified below.

Command entry must be throttled. This means that a command can only be entered every two seconds. This deliberately slows down client interaction with the server and simulates many thousands of clients using the bank server. Your client implementation should have two threads: a command-input thread to read commands from the user and send them to the server, and a response-output thread to read messages from the server and send them to the user. Having two threads allows the client to immediately print out to the user all messages from the server while the client is either waiting for the user to input a new command, or keeping the user from entering a command.

d. Client/Server Command Syntax

The command syntax allows the user to; create accounts, start sessions to serve specific accounts, and to exit the client process altogether:

create <accountname (char) >
serve <accountname (char) >
deposit <amount (double) >
withdraw <amount (double) >
query
end
quit

The client process will send commands to the bank, and the bank will send responses back to the client. Every messages the client sends to the server should result in a response from the server; either data, an error or a confirmation message.

Create:
The create command creates a new account for the bank. It is an error if the bank already has an account with the specified name or can not create the account for some reason. A client in a service session (see below) cannot create new accounts, but another client who is not in a customer session can create new accounts. The name specified uniquely identifies the account. An account name will be at most 255 characters. The initial balance of a newly created account is zero. It is only possible to create one new account at a time, no matter how many clients are currently connected to the server.

Serve:
The serve command starts a service session for a specific account. The deposit, withdraw, query and end commands are only valid in a service session. It is not possible to start more than one service session in any single client window, although there can be concurrent service sessions for different accounts in different client windows. Under no circumstances can there be concurrent service sessions for the same account. Once a client ends (see below) a service session, they can start a new one for a different account or the same account, so it is possible to have any number of *sequential* service sessions from the same client.

Deposit/Withdraw:
The deposit and withdraw commands add and subtract amounts from an account balance. Amounts are specified as floating-point numbers. Both commands should result in an error if the client is not in a service session. There are no constraints on the size of a deposit, but a withdrawal is invalid if the requested amount exceeds the current balance for the account. Invalid withdrawal attempts leave the current balance unchanged.

Query:
The query command simply returns the current account balance.

End:
The end command ends the current service session. Once the service session is ended, it is possible to create new accounts or start a new service session.

Quit:
The quit command disconnects the client from the server and ends the client process. The server process should continue execution.

e. Client/Server Start-up
The client and server programs can be invoked in any order. Client processes that cannot find the server should repeatedly try to connect every 3 seconds until they find a server. The client must specify the name of the machine and port where the client expects to find the server process as a command-line argument. The server takes the port to listen on as the only argument:

(be sure to use a port number over 8K (8192) that you pick for your group)

./bankingServer 9999

./bankingClient cp.cs.rutgers.edu 9999

Implementation

a. Messages

Minimally, your code should produce the following messages:

Client announces completion of connection to server
Server announces acceptance of connection from client
Client disconnects (or is disconnected) from the server
Server disconnects from a client
Client displays all error messages received from the server
Client displays informational messages received from the server
Client displays successful command completion messages received from the server

b. Program Termination

The server can be shut down by SIGINT, caused by using Ctrl+C on the server's terminal.
The server should catch SIGINT and shut down gracefully. In particular, it should stop its timer, lock all accounts, disconnect all clients, send all clients a shutdown message, deallocate all memory, close all sockets and join() all threads.
Clients should shut down automatically when they receive the server's shutdown message.

Submit

A “Asst3.pdf” documenting your design and paying particular attention to the thread synchronization requirements of your application.
All source code including both implementation (.c) and interface(.h) files.
A “Makefile” that produces the executable program files:
“bankingServer”
“bankingClient”
.. with 'make' or 'make all'
Please submit all files compressed in to a “Asst3.tar.gz”