CS 214: Systems Programming, Spring 2015
Programming Assignment 4: Multithreaded Bank System

1 Introduction

For this assignment, you will write programs for a multithreaded banking system simulation. This will give you an opportunity to exercise mutexes and thread coordination. You will write client/server programs with a single server that supports multiple clients communicating through TCP/IP network connections. Having to support multiple concurrent client-service threads in the server will require the use of mutexes to protect and manage shared data structures.

2 Server: Multithreaded Bank Server program

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 the connected client. You may have more than one client connecting to the server concurrently, so there may be multiple 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 a maximum of 20 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 100 characters long)`
- `Current balance (a floating-point 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.

3 Server: Printing Out Account Information

The bank server has to print out a complete list of all accounts every 20 seconds. The information printed for each account will include the account name, balance and "IN SERVICE" if there is an account session for that particular account. New accounts cannot be created while the bank is printing out the account information. Your implementation will uses timers, signal handlers and sempaphores.

4 Client: Connecting to the server

The client program requires the name of the machine running the server process 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 in the next section.

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 would 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 server to proactively and asynchronously send messages to the client even while the client is waiting for commands from the user.

5 Command Syntax

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

- **create** `accountname`
- **serve** `accountname`
- **deposit** `amount`
- **withdraw** `amount`
- **query**
- **end**
- **quit**

The client process will send commands to the bank, and the bank will send responses back to the client. The bank will send back error or confirmation messages for each command.

The **create** command creates a new account for the bank. It is an error if the bank already has a full list of accounts, or if an account with the specified name already exists. A client in a customer session 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 100 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.

The **serve** command starts a customer session for a specific account. The **deposit**, **withdraw**, **query** and **end** commands are only valid in a customer session. It is not possible to start more than one customer session in any single client window, although there can be concurrent customer sessions for different accounts in different client windows. Under no circumstances can there be

concurrent customer sessions for the same account. It is possible to have any number of sequential client sessions.

The **deposit** and **withdraw** commands add and subtract amounts from an account balance. Amounts are specified as floating-point numbers. Either command complains if the client is not in a customer 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.

The **query** command simply returns the current account balance.

The **end** command ends the customer session. Once the customer session is ended, it is possible to create new accounts or start a new customer session.

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

6 Deadlocks and Race Conditions

There should be NO DEADLOCKS and NO RACE CONDITIONS in your code.

7 Program 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. The client must specify the name of the machine where the client expects to find the server process as a command-line argument.

The server takes no command line arguments.

8 Implementation

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 error messages generated by the server.
- Client displays informational messages from the server.
- Client displays successful command completion messages generated by the server.

9 Program Termination

The server can be shut down by SIGINT, no signal handler necessary. The client(s) should shut down when the server shuts down.

10 Extra Credit

It should not be possible to have concurrent customer sessions for the same account, which requires a mutex lock for each account. That's not the extra credit part. This is the extra credit part: Instead of silently blocking while trying to start a customer session, the bank could try to lock the mutex for the account every 2 seconds and if the locking attempt fails, the server could send a "waiting to start customer session for account so-and-so" message to the appropriate client process.

Normally, the session-acceptor and client-service threads run in the same process. For extra credit, you can make the session-acceptor and client-service algorithms run as separate processes that share account information residing in `shared memory`. Your server might set up the bank in shared memory and then use `fork()` to spawn client-service child processes. As a responsible parent process, the session-acceptor process would use `wait()` to clean up the child client-service process(es) and clean up the shared memory.

If you choose to use separate processes, it would be useful (and look cool) to have the server process print out messages about the client-service process(es) is creates–something like "Created child service process <PID>. You can also add similar messages about how the server parent wait()s for each of its children.

You could also add messages about set up and removal of shared memory.

11 What to turn in

- A writeup documenting your design **paying particular attention to the thread synchronization requirements of your application**.
- All source code including both implementation (.c) and interface(.h) files.
- A makefile for producing the executable program files.

Your grade will be based on:

- Correctness (how well your code is working, including avoidance of deadlocks.
- Efficiency (avoidance of recomputation of the same results).
- Good design (how well written your design document and code are, including modularity and comments).