PA5: **MULTITHREADED BANK SYSTEM**

Tiago Costa and **Thanh Ngo**

(Extra Credit Implemented)

The Multithreaded bank system consists of 2 main programs : the client and the server programs, where the client program interacts with the server program using TCP/IP connection and continuously sends the requests and receives the responses to and from the server to accommodate the user transactions.

0 / The Communication

On the “application level”, the server and the client uses reqres.h interface to communicate with each other. Inside this file, there are 2 structures which are fundamental to this communication:

The request_t : abstracts the request from the clients, the instances of this struct are created every time the user asks a valid request and they are sent to the server.

The response_t : abstracts the response from the server, the instances of this struct are created when the server receives the request struct instance from the client. The server handles the request and form a response struct and send it back to the client.

1 / The Client Program

The client program is multithreaded oriented designed while the server program is multiprocess oriented.

The source code of the client is organized into the following source files:

client.c : contains the main program that will creates two threads : command thread and response thread which are stored in the following files respectively.

reqres.c and reqres.h as described above.

client-command.c (client-command.h interface) : contains the structure of the client command thread argument and the subroutine. This source file will handle user input on the command line.

client-response.c (client-response.h interface) : contains the structure of the response thread argument and the subroutine. This source file will handle the response received from the server program.

There is no locking mechanism in the design of the client.

2/ The Server Program

The server program ,through out it life cycle, creates multiple processes and also threads.

The processes:

The main process: this is the process created when we first execute the server binary. This process' main purpose is: to set up the server, to create a shared memory between the main process and the session process and to populate a session process (which is described below). Then it continuously prints out the account information every 20 seconds. When it receives the signal from the system, it stops printing out accounts, kills and waits for the session process.

(Extra credit)

The session process: this process proactively accepts incoming connection from new client and forks new processes, that we called the "client process", to handle the corresponding client's requests. Also, after accepting a new client, this process creates a new thread called the "client-collector-thread" which is described below and continue to accept new client.

(Extra credit)

The session process's client-collector thread: wait()s for the client process to be done and close the client socket. It also frees up any memory the session process allocated.

The client process: this process handles user incoming request

Note on the signal handling of the main process:

When SIGINT is sent to the main process: The main process stops the printing account loop. Then it sends another SIGINT signal to the session process to tells the session process to stop accepting incoming clients or whatever it is doing and exit. The main process then waits for its session process child, releases the shared memory, closes the server socket and exits.

(Extra-credit)

Processes' shared memory

The shared memory is created in the main process and is used throughout all the processes. This memory is mapped from a file in /tmp/.server_session.bin, which is deleted when the server system reboots.

The shared memory contains

+ 20 client accounts data
+ The current number of accounts in the bank
+ The new account lock mutex instance which locks the client process from creating new account.

They are abstracted by server_session_t struct in server-session.h.

The source code:

server.c : contains the main program, which will at first set up the shared memory of the main bank process and the session process.

reqres.c and reqres.h : abstracts the communication (as described in 0)).

server-session.c and server-session.h: contains the session process code.

server-client.c and server-client.h : has the client process code.

account.c and account.h: which contains the structure of the client accounts and an array of functions to handle the account. This structure is designed to be accessed by the server's processes (not the client's, for security purpose), which abstracts the client accounts.

Concurrency and locking mechanism :

New account mutex lock: In the shared memory, we have a mutex to lock one process from attempting to create a new account when another reads the number of accounts or creates a new account. The main process has to lock this mutex when printing out all accounts . Additionally, the client process also has to do this when receiving incoming open account request.

(Extra credit)

Account session mutex lock: this lock is used by the client process only. The lock is associated with each individual account, hence account_t struct contains an instance of the mutex. When a session is started on the account, the process try to lock it. If the locking fails, this means the account is in session. The process then sends a response to the client telling the user to wait, every 2 seconds, until the account is available to be in session.

3 / The efficiency:

The requirement of the project only allows at most 20 accounts in the server. Hence, the data structure used to store the user data is an array, which is efficient enough. And the searching algorithm when the user starts and opens an account is linear search, which has the complexity of O(n).

Hence, when the user performs open and start transaction the worst case time complexity of the server is O(n). Otherwise, the other transactions are O(1) in time complexity since they are straight forward when the client account is defined.

On the client side, every transaction is O(1) in time complexity.