Multithreaded Banking System

(with Multiprocessing Server and Shared Memory)

Phanindra Cherukuri

Rutgers University: New Brunswick

phani.cheruk@gmail.com

December 6, 2015

---

1 Description of Executables

This program simulates a bank system where users can create a bank account and deposit or withdraw money. It does this with two executables:

client.exe

The client.exe should be executed with the command " `./client <hostname>` " (in my test cases, the hostname was cd.cs.rutgers.edu as my project was stored on my iLabs account, so my command was `./client cd.cs.rutgers.edu`).

Upon entering this command, the client will attempt to connect to the server every three seconds. Once it successfully makes a connection, it automatically launches a reply_output_thread and a operation_input_thread.

In the main function of my client code, the operation_input_thread is joined so that the program continues to run as long as input is available. The reply_output_thread is **not** joined because the server should be able to send a message at any given time.

The functionality of the client thread is solely to read and write input options after these two threads are launched.

Commands that the user can execute:

- `open <accountname>` - open new account
- `start <accountname>` - start account session
- `finish` – close session
- `exit` – disconnect from server
- `balance` – view account balance
- `credit <amount>` - for deposits
- `debit <amount>` - for withdrawals

server.exe

I made the server.exe multiprocessing, and it should be executed with the command "`./server`".

Upon doing so, the server will automatically boot up and bind to a port. The main function of my server code also creates shared memory, and a message is shown in the terminal when it successfully does so. The main function also sets up signals and timers.

I set up a socket descriptor to accept any incoming connections in the parent function. After accepting a specific connection and getting a file descriptor, it forks to check if it is in the parent or child process.

- If in the parent process – the file closes and the loop continues.
- If in the child process – the alarm signal handler is set to `sig_IGN`, and moves on to the client handling function.

I created two functions to parse input, take actions, and display any needed outputs (`parse_request()` and `client_session_actions()`). **One issue here that I did not have time to solve is that the client side output can sometimes be cluttered with pieces of previous output statements due to some delay from `stdout`.**

2 Functionality Details

Memory and Semaphores

I created a struct called `BankServer` in the shared memory that contains a semaphore with its value initialized to 1. It also contains an array that is meant to hold a maximum of 20 bank accounts, and an integer called `numberOfAcc` that represents the current number of accounts registered in the system.

For each account field, another semaphore is stored, along with an account name, balance, and a flag that shows whether the given account is currently in-session ('`true`' if it is in-session/'`false`' if it is not).

Signal Handlers

There are three signal handlers which catch these signals:

1) `SIGCHLD`: If sigchld is caught, a wait is performed in the parent process to reap child processes.
2) `SIGAlrm`: If sigalrm is called, the current registry of bank accounts is printed, and the alrm is reset.
3) `SIGINT`: My sigint handler will send a message to all client processes to shut them down if you perform a Ctrl+C to disconnect the server. It will also clear all shared memory.

Functions

I have implemented a number of checks that make sure the user inputted command is valid before the particular function is called.

A global variable int flag called `insession` will tell whether a process is currently in a customer session. This flag is set to a value of 0 when the account is first opened, but once a session is started with said account, the value is set to 1. It will again be set to 0 when the user inputs a '`finish`' or '`exit`' command. `insession` should be true for clients to perform certain operations.

To open a new account in the bank system, the bank semaphore is locked so that the system update can be performed without interruption. After the update successfully occurs, the semaphore is unlocked again. Some checks in place include:

- Making sure the bank system is not at its maximum of 20 accounts
- Making sure that the name for the new account hasn't already been used

In order to print the current bank account information every 20 seconds, I catch a sigalrm and call a function called `printBankAccInfo()` from the signal handler, and reset the alarm. The function locks the bank semaphore, prints the information, and then unlocks it again.

There is no way a new account can be opened during the printing of the information as the bank semaphore is secured in the `printBankAccInfo()` and the `open_acc()` functions.

Customer sessions are controlled through the `start_acc()` and `finish()` functions. Each time a client attempts to start a session with a certain account, the account semaphore must first be locked for that account. If the client cannot immediately lock the semaphore (if that account is in session already from another client) it continuously tries with the function `sem_try_waits` until it can successfully do so and start the session. During this time period, a message is sent to to the client over a continuous time interval while they wait for the lock to occur ("Waiting to lock account and start customer session for <`accountname`>, please be patient…"). I put a delay in the try_wait loop to conserve processing power. Once the client successfully locks the account semaphore, the client can perform all operations (`balance`, `credit`, `debit`) until ending the session.

All sessions will stay active until terminated through a finish or exit command by the client. In either of these cases, the semaphore is unlocked, and picked up by any waiting process. The shared memory is completely freed and the socket descriptor in the parent function is closed.

3 Issues

One issue that I did not have time to solve is that the client side output can sometimes be cluttered with pieces of previous output statements due to some delay from `stdout`**.**

4 Other Implementations

- If a customer session is already in place for an account, the bank continually attempts to lock, and the server sends a message to the client over a continuous time interval while they wait for the lock to occur ("Waiting to lock account and start customer session for <`accountname`>, please be patient...").
- Implemented shared memory that uses a fork to spawn child processes and a wait() to clean up.
- The server (using separate processes) prints out messages (i.e. "created/ending/killing child processes)