Multi-Process, Multi-Threaded Server-Client Banking System README

Joseph DeVita and Anhelina Mahdzyar

April 23, 2015

Server, Session Acceptor

The server program holds the data structures and controls the logic for the overall system. Upon initialization two sets of named semaphores are created and detached from the process. Two areas of shared memory are created via mmap() to hold necessary data structures:

```
typedef struct {

        int avail;
        char account[100];

} Repo;


typedef struct {

    char account[100];
    double balance;
    short in_session;

} Bank;
```

A function: void client_session (void * arg, Repo * repo, Bank * bank) is activated via a wrapper forking function: pid_t forkChild(void (* function) (void *, Repo *, Bank * b), void * ptr, Repo * repo, Bank * accounts) every-time the server reads a valid socket descriptor from the associated port (42968). The function: void print_bank(Repo * repo, Bank * accounts) uses the same function to create a separate process which handles printing the bank once per 20/sec. A signal handler for SIGCHILD is used to reap child processes which terminate before the main thread terminates. Another for SIGINT is used to shut down the program. A call to SIGINT will clean up the shared memory before exiting the main processes.

Server, Client Service Process

When the session acceptor thread calls: pid_t forkChild(void (* function) (void *, Repo *, Bank *), void * ptr, Repo * repo, Bank * accounts), a new child process is created to exclusively handle communication between a client and shared memory. The server will read from the socket descriptor from the client until

the client communication ceases or client quits. Commands are passed via characters appended to the end of any output originating from the client. Depending on the command, the server may also append special characters to put the client in session and/or let the client know the data structure it is trying to access is currently in use.

Client

The client program requires a host name passed via command line to connect. It will also need to have a matching port to properly establish communication to the server. Upon successful creation of a socket descriptor, the client thread will spawn a thread in which it uses for I/O with the server. Within this thread, the client can be in a general session or a client session. The general state allows for the following functionality:

General Session

```
int general_client_session()
{
        switch(option){

                case '1':
                        write(1, "Create: ", 8);
                case '2':
                        write(1, "Serve: ", 7);
                case '3':
                        close_client();
                default:
                        write(1, "ERROR: invalid input\n\n", 22);
        }

}
```

1. Create: Pass a string to the server to represent a means of identification for a new account. The server will respond whether the account creation was successful or not.

2. Serve: Accepts an account name in the form of a string and requests that the server give exclusive access to this client until the client submits an exit command or terminates.

3. Quit: Terminates the connection with the bank and the client process itself closes its socket descriptor and terminates.

Client Session

```
int account_client_session( )
{
        switch(option){
                case '1':
                        write(1, "Deposit: ", 9);
                case '2':
```

```
                write(1, "Withdraw: ", 10);
            case '3':
                write(1, "Query...\n", 9);
            case '4':
                write(1, "End...\n", 7);
            default:
                write(1, "ERROR: invalid input\n\n", 22);
        }

}
```

1. Depost: Accept a string and convert into a double and add it to the account-in-session's balance. If the value is negative it will add nothing.

2. Withdraw: Lower the account-in-session's balance by the amount converted by atof() in the sent string. If the amount < the current balance the user will be notified of their error.

3. Query: Returns the account-in-session's current balance.

4. End: Requests that the server put the account-in-session out of session.

Testing: Summary

During development most of the potential pitfalls had to be thoroughly handled otherwise the program simply wouldn't run in a state where test cases could be implemented. Two main areas consumed most of development time during development (aside from dying family members).

1. Handling synchronization between different processes as opposed to threads.

2. Creating shared memory in a unix environment. Understand shared memory enough to anticipate synchronization problems.

Testing: Synchronization Mechanism Selection

A lot of time went into choosing how to handle locking individual accounts and the entire bank. If this were the multi-threaded version, mutexes would have been used, but the literature online (stack-overflow) offered a different method to attack the problem given the multi process nature of the program. After a familiarization with: POSIX unnamed semaphores, POSIX named semaphores, and System V Semaphores. Named semaphores were deemed best for two reasons. One their similarity to unnamed semaphores, thus being more straight-forward than System V Semaphores. They also could be unlinked from the parent process upon creation, so in the event of improper termination, once the parent process died (The server application) the semaphores themselves will essentially auto-destruct. They also had a convenient flag: O_CREAT — O_EXCL which allowed them to be shared easily between forked client-service processes. As the bank was an array of structs and it can only be filled sequentially, an array of named semaphores initialized with the bank was all that was needed to handle mutual exclusion ownership of an account.

```
sem_t * sem[20];
sem_t * rep;
```

Testing: Shared Memory Implementation

shmget() and mmap() were two choices that were applicable to the problem. The child client-service process is the only process, along with the bank timer, which manipulates shared memory. For the same reason the of the un-needed complexity of System V Semaphores vs. named semaphores, mmap() was more than sufficient for this program. mmap() shared memory doesn't need nearly any extra overhead once the objects are initialized inside it to share between child processes. Simply creating a new pointer exclusive to each client-service process to access the memory, and it functions nearly the same as standard heap memory.

Testing: Use-Case: General Commands/Communcation

```
TC 1: ./client <valid host name>
: <prompt of successful connection>
TC 2: 1 <account name that isn't in use>
:  <confirmation account was added to the bank> [bank has avail spots]
TC 3: 2 <account name that isn't in bank>
: <error message>
TC 4: 2 <account name that is in the bank and not in use>
: <confirmation + flag to put the user in session>
TC 5: 2 <account name that is in the bank and IN USE>
: <server message its in use x 5 @ sec/per >  <timeout + return control>
TC 6: 3 <close SD and quit>
: <socket is closed and can be reused by another program requesting it.
TC 9: 1 <account name that is in use>
: <error message>
TC 10: 1 <account name that isn't in use>
: <error mesage> [bank is full]
```

Testing: Use-Case: General Commands/Communication

```
TC 1: 1 <string with valid positive numbers>
: <confirmation of deposit>
TC 2: 1 <string with no numbers>
: <error message>
TC 3: 1 <string with negative numbers>
: <error message>
TC 4: 2 <string with valid positive numbers and is < balance>
: <confirmation of withdraw
TC 5: 2 <string with valid positive numbers and is > balance>
: <error message>
TC 6: 2 <string with no numbers>
: <error message>
TC 7: 3 <>
: <account balance>
TC 8 : 4 <>
: <confirmation of termination of account session>
```

Testing: Use-Case: Synchronization

TC 0: <client−service can access shared memory>
: <pointer can reference variables declared by parents>
TC 1: 1 <valid account> + (access REPO during bank print)
: <Bank prints as single snapshot> + (NO RACE CONDITION)
TC 2: 2 <valid account> + (valid account in session by different client)
: <Pending message x5> + <time out>
: <Pending message ∗ n (n < 5) + <session confirmation>
TC 3: 1 <valid account> + (BANK is at max capacity)
: <error message>
TC 4: ./client ∗ 20 + ∗ 20 commands
: <proper output> (no race conditions)
TC 5: 2 <valid account name> + (n ./clients executing command concurrently)
: <proper output> + (NO DEADLOCK)
TC 6 <reference pointer in server> + <client has used shared mem>
:<references do corrupt each others>