Ariel Ahdoot (aa1046)
Ashni Mehta (am1531)

PA5: MULTITHREADED & MULTIPROCESSED BANK SYSTEM
(with shared memory!!)

Server

server.h
Within our header file for our server, we have two separate struct definitions: Bank and Account. Each Account is comprised of a char array of capacity 100 (as the maximum length of an account name is 100 characters), a float balance, an in session flag (that we call isf), and a semaphore.

Within the bank struct, there is another semaphore (that is used when printing out bank information), an Account array of capacity 20 (because the maximum number of accounts is 20), and an int to hold the number of accounts that the bank currently holds.

We have a variety of function definitions within our header file as well, but will go into an in-depth explanation below.

server.c

- **main** sets up an alarm of 20 seconds, that we use to **print bank info** if there exists at least one account within the bank. We use the function **claim_port** to bind to our chosen port (36963 – a palindrome made of all odd numbers). We listen for incoming client connections, and once one is received, we set up shared memory (with a call to **shm_setup**) and we fork (multiprocessing!!!) and send the client commands to **detrequest.**
- **claim_port** accepts a char* port, and after creating an addrinfo and addrinfo*, we bind to the port specified using the socket(), setsockopt(), and bind() system calls.
- **shm_setup** (shared memory!!) generates a key based on the character ‘s’ and creates a Bank within shared memory. We initialize the semaphore for the bank struct (with only two options – locked or unlocked), and set the value of current accounts = 0 (as there should be no accounts currently in the bank).
- **detrequest** parses the client input for commands, and based on the number of arguments determines what the client would like to do. Depending on the client input, we have functions dedicated to each of the bank commands: **exit, balance, finish, open (makeAccount in our case), start, credit, and debit**. After a call to the appropriate function, **detrequest** returns.
- **exit** doesn’t exist as its own function, but rather exists within **detrequest**. If we are within a client-server session (which we should be) we unlock the semaphore for the account we’re currently serving and set the in-sesion flag to false. We then exit(0).
- **balance** returns the current balance of the account (a float, but in traditional xx.xx decimal form).
- **finish** first checks to see if we are currently in a session, and if yes, unlocks the semaphore for the associated account and sets the in-session flag to 0.

- **makeAccount** first checks to see if the current process is already serving an existing account, then makes sure that there is enough room within the bank to add the new account. If the name specified by the client is already in use by an existing account within the bank, the account cannot be created. Otherwise, we create a new account with the name specified by the client, and initialize the balance and in-session flag. While creating the account, we lock the bank's semaphore.
- **start** uses our method **findaccount** to find the index of the account specified by the client, then locks the semaphore associated with that account (as only one client can access a given account at a time). We also set the busy flag for this process, as it is now currently in a client-server session.
- **credit** makes sure that the amount specified by the client is a valid amount, then credits the appropriate amount to the index specified by the client in **start**.
- **debit** uses the same logic as **credit**, and debits the appropriate amount to the index specified by **start**.
- **findaccount** uses a linear search to find the account specified by the name sent in. Although on large data sets linear search is certainly not the most efficient way to search, we decided that because the maximum number of accounts is twenty, linear search vs. a more optimal searching algorithm wouldn't create much of a difference.

Client

client.c

- **main** checks to make sure that the number of arguments is 2 (the name of the machine running the server process must be specified), then tries to connect to the server through **serverconnect** every three seconds. Once the client has successfully connected to the server, we **spawn_threads** (multithreading!!) and close the socket descriptor. We then disconnect from the server and return 0 upon success.
- **serverconnect** takes in the server and port names, and after creating an addrinfo and addrinfo*, attempts a getaddrinfo. Based on the value of ai_flag returned, we have a variety of informational error statements to help the client better understand what is happening incorrectly (these were taken from **netdb.h**). We attempt to create a socket descriptor using the socket() system call, and if we are able to connect to the server using the connect() system call, we are successful! Otherwise, we have the client wait for three seconds (**sleep**(3)) and then attempt to connect again (this is within a while loop).
- **spawn_threads** creates two threads, one to take in input from the client and send it to the server, and one to receive output from the server and return it to the client. We use two functions, **c_input** and **s_output** to handle these two threads. We join() on the client input thread rather than the server output thread, as we need to keep taking in client input.
- **c_input** takes the client input and sends the request to the server. We include a call to **sleep**(2) so that the client input is throttled by two seconds.

- **s_output** receives server output and uses a pthread_detach from the current thread that's running.

TL;DR

Please be lenient.

All jokes aside, this bank system uses **multithreading**, **multiprocessing**, and **shared memory**. We have a single server that can support multiple clients, all communicating through TCP/IP network connections. Each account within the bank system has a semaphore, and the bank has its own semaphore, bringing our semaphore count to 21 semaphores for this assignment. We chose semaphores instead of mutexes because they sound cooler, and in this case, really just are glorified mutexes.