Anthony Wong
Allen Tung

1.1 Introduction

In this assignment our intention was to create a Serve-Client program that would emulate a bank. We followed the C and Posix standard socket programming to accomplish our goals. Utilizing the IPv4 standard, sockets, and multi-processing we were able to meet the project specifications.

1.2 Discussion of Shared Memory

Since clients are required to communicate to a central bank and we opted to used fork() instead of the thread operation, we needed a way to allow the processes to access the central bank of information. To this end we used mmap and its family of functions to created a shared memory block to store account information.

1.3 Multiprocessing and Synchronization

Since we used multiprocessing we needed to synchronize the critical sections of our code; in particular accesses to shared memory. We used the Readers-Writer philosophy which follows three rules: if there was a writer in shared memory there could not be any readers, if there are readers we cannot have any writers and we cannot have more than one writer. These conditions were kept track of by three named semaphores and a global reader count; "Readо" and "Writeo" as their names suggest locked reading and writing in shared memory. Global variable Readers was used to keep track of the total readers in shared memory. Only when readers equal zero can the write semaphore be waited on. The third semaphore, "Welcome" was used only to synchronize changes to the global reader count. To synchronize parent-child processes, we used a signal handler that would handle SIGCHLD and properly wait and exit ending child processes

1.4 Client

The client program used multi-threading to asynchronously read and write to the server. This made is so that the client would not be blocked by either the read or write functions. In the write thread, the client actively checks if the input is valid before sending. This check is done locally so that the server does not need to waste time sending packets back and forth with invalid input.

1.5 Signal Handling and Termination

Beside our child signal handler discussed in 1.3, signal handlers in our server and client are used to handle sudden termination of either the server or client. If SIGINT is raised in the server, the SIGQUIT is raised to all processes, promptly terminating them. If the client is raised

SIGINT, we have client send the server the command "quit" before exiting acting as though it exited by quit. This allows for the server to free any semaphores that were held by that process and allow it to exit the client without complaint.

1.6 Efficiency

We opted for server multi-processing for ease of synchronization and manipulation of data. Of course, this cam with an overhead of slower inter-process communication and inefficient shared memory.