Name:Daniel Ems
Date: 06/17/2018
Current Module: Networking in C
Project Name:fdr

Project Goals:

The goal of the project was to create a server that hosted functions, and could be communicated with via udp. The functions the server needed to host were F(), R(), and D(). F took a decimal value and returned that position in the Fibonacci sequence. R took a Roman Numeral string and converted it to Hex. Finally, D took a large decimal and converted it to hex. The server had to listen on three separate ports.

Considerations:

1. How will you set up the server?
2. What error handling do you need to consider with server?
3. How will you listen on multiple ports?
4. What requirements will each function depend on?
5. What error checking will be needed for each function?
6. How will large numbers be handled?
7. Will the Fibonacci sequence be rewritten or linked?
8. How will the server respond when invalid input is given?

Initial Design:

The program contains a Makefile, a fdr.c, roman.c and roman.h. The Makefile contains the appropriate links, flags, and warnings. Roman.h contains the headers for roman.c. Roman.c contains the functions and data structures to convert roman numeral into hex. Fdr.c contains the main for the program, server dependency functions, bignum(), and fib(). The Makefile links roman.o to fdr.c.

Data Flow:

The program begins by setting the server up. The children are set up to receive udp messages on UID + 0, + 1000, and + 2000. Once a child receives a packet, the string is sent to buf_strip,a switch case that calls the appropriate function based on the string received. The functions will perform their corresponding purpose and return a string to the buf_string, and buf_strip will return the string to the children where the string is sent back to client.

Communication Protocol:

UDP

Potential Pitfalls:

Their are several pitfalls that come along with this project. The primary source of which is the limited knowledge we posses on setting up sockets. However, that is a moderately easy thing to figure out. Completing the functions requested serves a separately equal task. The potential downfall from the functions is found in error handling, handling ridiculously large numbers, and properly getting that information back to the client. The most difficult of all was Roman numerals. This required creating a dictionary, performing a lot of error checking, and properly converting the int into a string.

Test Plan:
User Test:
A) In regards to the functions.

1. hard code all values.
2. where possible *implement a portion of a requirements.
    a. see if you can convert M to 1000 before trying to add.
3. try to compile.

4. check output.
5 if success → go back to 1 and repeat moving through requirements
6. if unsuccessful → debug and come back to 4.
7. Once requirements are met, try user passed info from cmd.
8. repeat steps 1-5.
9. once successful, try and use client messages.
10. repeat 1-4.
11. Once successful, error check. Repeating steps as necessary.

B) In regards to the server
1. hard code address and ports.
2. have printf for connection.
3. try to compile.
4. try to connect.
5. once connected, try to fork single process.
6. repeat 1-5 until successful and have three forks.

Test Cases:

The test cases for this program were a little sporadic due to the scope, and range of the programs elements. Each function will get a brief paragraph on test cases used while working towards completing them.

The Server:

The server was hard coded to 127.0.0.1 and port 1000. I set up the basic dependencies and set up a printf to the server (not the client) that would print if a connection was successful. I would then use nc to try and send a message. Once I knew I was able to access the server on that port. I would then try and access the message sent. Once I was able to do so, I began to work on the functions.

Fib:

Much of the Fibonacci had been done before. The only test case was switching to c, and using big numbers. The test cases implemented here were to see if I was able to get accurate numbers while using only big numbers.

Big numbers:

This was useful in getting fib to work in c. The bn man page offers an extensive list of functions that allows you to manipulate big functions. The test case here was first to see if I could create a bn object. Then put something in it. Then print it. Finally, it was a matter of seeing if I could manipulate it into hex. All of which there are built in functions to support.

Roman Numerals:

Roman numerals offered the most trouble due to the key value matching. The first attempt was to place the pairs in an array. This was abandoned. The next attempt was to build a dictionary with key value pairs. Once This was done, I was able to convert the Roman numeral into ints and store them in an array. This allowed for me to start error checking legitimate values. This was completed through a system of trial and error. A case would be sent and if it failed. I would try and figure out what exactly had went wrong.

Conclusion:

This project began as a nightmare, and then became a lot of fun. I traveled this weekend, and so my ability to work Saturday was limited. This led to a shorthanded project. No comments, error checking not complete, edge cases not evaluated fully. This is important because I was thoroughly disappointed that I was unable to implement a lot of the functionality that big numbers have to offer. It would have been nice to be able to write back to the client error messages as well.

I think networking is fun because it is challenging.

Note * John Haubrich helped me with setting up the architecture of my server, and the theoretical understanding of the bn’s.