Name: Brian Jon Stout
Date: 17JUNE2017
Current Module: Networking in C
Project Name: FDR

Project Goals

The goal of this project is to write a UDP server that when prompted for certain services, and send a response back to the client

Considerations

- Server must run on three ports
- Server must be able to respond to requests

Initial Design

The main file is called fdr.c. Fdr.c is responsible for forking the server processes to children. The children call a function which runs the servers.

In the header file parsers, there are functions which are responsible for taking input from the client, and processing their meaning. Those parsers are the roman_numeral parser, the fibonacci parser, and the decimal to hex parser.

The hex parser calls the big_number module from openSSL to do it's processing. The fibonacci parser calls the fibonacci module, and the Roman_numeral parser calls the roman_numeral module.

The fibonacci module contains a function that processes the fibonacci number of a specific iteration.

The roman_numeral module, when given a string of valid roman_numerals, returns the value for those letters

Data Flow

The program starts in fdr.c. It grab the user's UID which will be used to populate the port. The main function then starts a fork and checks if the process is a child or a parent. If it is a child it creates a server using the create_server_fork function, which is responsible for setting up the sockets, binding them, and then running the infinite receive/send loop.

If the process is a parent it goes through and creates two more forks using different ports. A child process is given a bool value of true named is_child which is checked before the fork logic is ran preventing the children from creating their own children. At the end of that process it using the wait function to avoid an early exit, making orphans. The create_server_fork function starts with setting up a signal handler in the server. Hindsight, since forks duplicate the address

space for each process we could have done this in main, but the intent was to have each fork return with unique data to the parent. After this the program goes through the steps of creating, and binding a socket. Most of the code here is repurposed from the book's examples.

The program transitions into an infinite for loop which receives data from the client and sends it into a buffer. The program starts up a signal handler for each fork. The datagram is passed to the newline_terminator function, which loops through the datagram and replaces all newlines with a nullbyte. The purpose of this function is to make the inputs from netcat by itself, or netcat sending an echo'd response (like from the sample use on the assignment) more similar to each other so they both get responses from the server.

Following this, the program processes the data using a switch statement. It looks at the first letter of a string and determines which parser it should use. If none of the case statements match the first letter, it prints "Error!" and drops the packet. The three case statements are for the fibonacci parser, the roman_numeral parser, and the decimal to hex parser.

The fibonacci parser converts the string (after the initial letter using pointer arithmetic) into a usable number. It checks to make sure there wasn't any extra garbage, and then checks to make sure it wasn't bigger than 300 (the hard coded limit for the assignment.) Before that it checks to see if the string has a negative symbol in the front, preventing the value from ever being below 0. After this, it create a BIGNUM * and calls the BN_fibonacci function. The BN_fibonacci function is a fibonacci calculator that uses the BIGNUM data types so it can have a very very very very large number. After the function return the big number, the BN_bn2hex function from the openSSL/bn module is called on the big number. The output of the function is a hexadecimal string representation of the number. The string is then copied to a buffer which already has '0x' in it using strncat. The string output from BN_bn2hex requires the OPENSSL_free function is run on it to free the memory specifically so that's done immediately after it's not needed any more. The big_number also needs to be free'd before the function is closed. If the string fails any of the parsers checks it returns null instead of the buffer holding the return string.

The roman_numeral parser does similar things to the fib parser. It grabs the relevant data from the UDP DGRAM sent from the client and sends it to to the roman_numeral_converter function in the roman_numeral module. The roman_numeral_converter finds the length of the string being processed and loops through the string based on the length. There's a few checks making sure bigger numerals don't come after small ones, and that certain numerals aren't repeated multiple times, making the roman numerals additive only. The largest value being MMMM (4000 limit,) but the longest string being MMMDCCCCLXXXXVIIII (3999.)

Each value is sent to the roman_character_value function which uses a switch statement that returns a value based on the letter being passed. The value is added to the variable ret_num if each loop is successfully. If at any point the string breaks the additive numeral rules, or there's an illegal character, the loop set ret_num to -1 and breaks. The parser checks to see if it's a

negative value to determine if the string was good or not. The final number is converted to hexadecimal output using snprintf, which sends the results to buf. Buf is a char pointer with malloc'd memory based of an arbitrary power of 2.

The decimal to hex parser (named hex_convert_parter) grabs the decimal represented as string and sends it to the BN_dec2bn function which processes a decimal represented string and sends it to the passed BIGNUM address. The function returns length which is how many digits the the string had. The parser then uses this length value to determine if it's valid or invalid. If it's 0, nothing was done so it returns NULL, if the length 21, it has to run the string through a loop to determine if it exceeds 10^20, otherwise it's an invalid string. If it is greater than 21, then it definitely exceeds the max value and the function returns a NULL. After this the big number is passed to the BN_bn2hex function, which converts a big number to a hexadecimal string representation. The big number is freed being done with, the hex_string is formatted using snprintf so it's prefaced with a 0x, and then the hex_string is free'd using OPENSSL_free(). The snprintf outputs to buf, a char pointer with malloc'd memory based on the length of the hex string plus byes for the 0x, null and newline. The function returns the buffer.

Back at fdr.c (main), All the functions return the string to a char * called func_ret. The program checks to see if func_ret is null and if it is, it skips the loop. If it isn't, it gets the length of that func_ret, and copies it into the buffer that the server received data into, and then terminates it with a newline and a nullbyte. The func_ret string is free'd since all three functions allocate memory for it. After the buffer is prepared it's sent back to the client using the sendto function.

Communication Protocol

UDP

Potential Pitfalls

- SQL Injection vulnerability from string appends.
- Not verifying database integrity

Test Plan

User Test:

1. Send F10, D10, RMMII to each of the ports using the echo netcat method
2. send F10, D10, RMMII to each of the ports using regular netcat method
3. Send junk using other method
4. Send valid data but with spaces and alpha characters making it invalid
5. Send RMMMDCCCCLXXXXVIIII to server

6. Send 300 to the fib process
7. Send 10^20 to decimal process
8. Send D10^20+1 F300+1 RMMMM to their respective processes

Test Cases:

1. Expected output
2. Expected output
3. No server response
4. Responses containing alpha characters fail, responses containing just spaces do not for certain functions such as the DEC2HEX portion. This is partially because the dec2bn function does not return an error unless it encounters an alpha character. If I had additional time I would have implemented a method that scans the string for spaces before hand (a whitespace remover would have been nice for all these functions actually.)
5. Response is 0xf9f or 3999
6. Expected output
7. expected output
8. No server response, as expected

Conclusion

More time for this program would have been nice. While everything works as expected there's some small inconsistency issues that could be fixed and improved. These issues exist because something had to be implemented at the time and later on it became obvious that better methods were available. Call it lessons learned.

Examples are analyzing the data before passing it to the functions. The pointer arithmetic is fine but beforehand it probably would have been better to go through the data and remove extra bits, running is_digit and is_alpha on the strings before it even gets to the parsers. Having consistent format methods for the data (the 0x preface) would have been better to. There could have been a prebuilt string at the start of the loop containing the preface and each func could have just returned then we could use snprintf to the buffer for the data (since it was all hex) and formatted it there. It would have been a little bit of extra logic but maybe it would have made things more readable. As it is now certain functions get different input because the different methods were used when different problems showed up instead of getting a standard output from each function and then adding the preface.

There was also a lot of time wasted on getting the assembly fib function to work in this program. I attempted to do what Cuillo with his fib function, did but it was pretty unstable and the time spent fixing it would have been the same time spent making a more robust solution for the fibonacci in C. Even though I got it working there was some issues with calling other functions

like strlen, or strncpy after words. Just adding a printf gave an assembler error for mismatched operands ‘ror’ and I couldn’t figure out how to debug it there, I assume it had something to do with buffered input.

There’s certain formatting issues that I missed the first go around that I’ll probably fix after the write up is finished just for the sake of fixing it even though it’s not likely to count, small things like extra spaces. It’s also super jarring switching from different styles. I think it might be worth the time setting up a new indent style for the new style guide since using the indent program saves a massive amount of heartache from inconsistent styling.