CS214: Assignment 3 Readme

netfileserver.c:

This is the server program that multiple clients can connect to and then open, read, write, and/or close files that are saved on the same machine that the server program is running on. The server has a main and two helper functions, sendAndRec() and error().

-main():

The server takes in a port number as an argument and attempts to create a socket. If the server is run with an incorrect number of arguments or the socket is not created the program exits with an error. The server's address structure is then set up and the socket is bound to the local port number given at runtime. At this point the server can begin to accept connections from clients so listen is called. When a client tries to connect the while loop in main is entered, which handles connecting to the client and any errors that may occur during that process. Our server uses threads to allow multiple connections at once. An array of size 10 is created to store up to 10 different thread IDs. We decided to only allow a maximum of 10 clients to connect to our server at once, hence the array of size 10. We use a while loop when the first client attempts to connect in order to allow multiple connections at once. After the first client connects a thread is created, storing the thread ID in our array and passing the helper function sendAndRec() as the function that each thread will execute. The while loop iterates and upon having another client attempt to connect will repeat this process. Threads are needed in order for our server to accept multiple connections at once because the server needs to be able to simultaneously listen and connect to any clients attempting to connect, while also executing commands from any clients already connected. The while loop continuously waits for attempts to connect to the server, then runs the code to make the connection and create the thread for the client, then iterates, which is how our server can make multiple connections to clients at once. The while loop also checks the thread ID for each client, if it is negative (meaning there was an error when trying to create the thread) it prints an error and exits, if it is equal to zero (which means the thread has finished running) it closes the connection to that client and calls exit(0), indicating success.

-error():

Anytime the server gets a request to open, read, write, or close a file, but is unable to, error() is called. It takes a string as an argument which describes the specific error the server had when trying to execute a command sent by a client. The string containing the message describing the error is passed to perror(), which displays the error mesage, and then exit(1) is called, which ends the server program due to an error (0 is used for exit() when the program ends successfully, and non-zero values are used for errors, so here when server stops running, the exit(1) call indicates that the program had an error causing it to end.

-sendAndRec():

This method is what each thread/client is using. It uses a continuous while loop in order to receive and run commands sent by the client, one at a time. Several character arrays are used to store info received from the client and info to be sent back to the client. A file pointer is also created in order to execute the commands the client sends on the file specified. First, a single character is read from a client which is used to determine what command the client is sending. For open, close, read, and write the client would send “o”, “c”, “r”, and “w” respectively. If statements are used to determine which one of these four was sent by the client (or if the client sent an invalid command, in which case it calls error()) and then runs the code for each.

For open, the server reads from the client again, this time receiving the file path of the file to be opened. If the file path is not valid error() is called. The server then reads from the client again, this time receiving “0”, “1”, or “2” which indicate the access mode to be used when opening the file, they correlate to read-only, write-only, and read-write respectively. Upon success, the file pointer is set after fopen() and the file descriptor is sent back to the client, upon failure the client is sent back -1.

For read, the server reads from the client again, this time receiving the number of bytes the client wants to read. If the commands were successful the server goes into the specified file, reads the number of bytes given to it by the client, then writes back the data it read and the number of bytes it successfully read.

For write, the server reads from the client twice, receiving the data to be written and the number of bytes the client wants to write. As long as there are no errors, the server goes into the correct file and overwrites it with whatever data the client sent it. Upon success it writes back to the client the number of bytes it wrote into the file. Upon error the server writes back 0 to the client, indicating failure to write any data, and then calls error().

Close simply attempts to close the current file pointer, which should correlate to an open file. If the current file pointer is not valid, or does not correlate to an open file, then error() is called and -1 is sent back to the client, indicating a failure to close the file. If fclose() is sucessful, the file is closed and the server writes 0 to the client indicating success.

It is important to note that the clients commands have to be called in a correct way in order to be successful. A client can only open one file at a time, so calling netopen twice would simply set the file pointer in the server to the second file the client opened. If netclose is then called, the second file is closed. Nonsensical flow of commands will result in errors i.e. trying to read or write after closing a file, trying to read or write without opening a file, etc.

libnetfiles.c:

This is the code for the client, it attempts to connect to a server, and once connected, can send commands to open, close, read, or write to specified, valid, files, on the server. If the client successfully connects to the server, the user is prompted with instructions on how to call the commands listed, which state that the user must first enter either “o”, “c”, “w”, or “r”. This input is used to determine the command the client is sending to the server. Once this is known it is sent to the server. The main() function in the client handles up to here, once the client is connected and the user is prompted for a command and inputs it, main() then calls a helper function that will handle the rest of the command, as well as prompt for any commands after so long as the last command called was successful. The user is then prompted again with instructions for each of the commands. Each one prompts the client for its necessary inputs (the file path and access mode for netopen, the number of bytes to read for netread, etc.) and writes them over to the server. The server then attempts to execute the command send with the arguments the user inputted and then writes back to the client, indicating success or failure as well as writing back data in some cases. We use a continuous while loop that exits on error or by indication by the user (which would be a command that disconnects the client from the server and ends the program). Each iteration of the while loop handles a single command from the user, which allows for as many commands the user wants to send the server as he would like.

-**netopen():**

Takes a file path and an access mode, sends both to the server, which attempts to open the file indicated by the file path in the correct access mode. It then reads from the server a file descriptor if it was successful in opening the file, or -1 if it failed to open, and returns that value.

-netclose():

Takes a file descriptor that correlates to a file the client wants to close. It just writes “c” to the server which indicates that the client wants to attempt to close a file. The way our implementation works, the client does not need to send the file descriptor to the server in order to close the file, because the server saves a singular file pointer correlating to either a singular open file, or a singular closed file. If the file pointer in the server correlates to a closed file, or netclose() was called twice in a row, etc. the server sends back -1 and the client prints the error. If the file pointer in the server correlates to a valid, open file, it closes it and sends back 0.

-netread():

Takes in a file descriptor, void * buffer, and a size_t indicating the file the client wants to read from, and how many bytes it wants to read. It sends the server “r” indicating a read command from the client, and then the number of bytes it wants. The server writes back the data it read from the file and the number of bytes it read upon success, and sends back -1 upon failure. Reasons read can fail include attempting to read from a file that hasn’t been opened yet, attempting to read from a file that has already been closed, etc.

-netwrite():

Takes in the same arguments as netread() except buf is a const void * instead of just a void *. The client sends the server a “w”, indicating a write command, and then sends the server the const void *buf, which holds the data the client is trying to write, and then sends the number of bytes it wants to write. The server responds with the number of bytes it was successfully able to write, or -1 upon error. This function will replace the data in the specified file with the data the user enters. Different implementations can be done here that would allow for the user to specify an overwrite or an append to a file but our implementation only overwrites the data.

The client will stop running if an invalid command is sent, if the server stops running while it is connected, or if the user stops it himself. So if no errors occur and the server stays running, the client program will continuously run, allowing any number of valid commands to be sent by the client, until the user disconnects from the server.

Problems we encountered while writing the code for this project were almost entirely based around sending and receiving the correct data between the client and server. Because only strings can be sent between a client and a server, anytime any data that isn’t a string needs to be sent/received it first must be translated into a string in order to send it, and translated back to its correct data type after being received. These translations being done incorrectly was the majority of the source of the bugs we encountered.