Assignment 3: A Better Open

A Better Open is a multi-threaded file server that allows multiple clients and a server to establish communication and perform operations on files. Essentially, A Better Open allows multiple clients to perform open, read, write and close operations on files that are not within the clients file directory by communicating with a server, and the server will then perform the operations on the files.

Authors:

Ganapati Tirumalareddy
Netid: gt253
gt253@scarletmail.rutgers.edu
Section: 06

Alex Silva
Netid: ars366
ars366@scarletmail.rutgers.edu
Section: 06

Synopsis:

To run the server:

./netfileserver

To run the client:

./client <server name> <file name> <operation> <mode>

Example:

./client man.cs testfile.txt B u
(R – read, W- write, B – both)
(u – unrestricted, e – exclusive, t – transaction)

Protocol:

- Client:
  - open(): “open [mode] [file name length] [file name] [flags]\n”
  - read(): “read [file descriptor] [# of bytes to be read]\n”
  - write(): “write [file descriptor] [# of bytes to be written] [string to be written to file]\n”
  - close(): “close [file descriptor]”
- Server:

Description:

First the server is executed to be able to accept incoming connections with clients. This is achieved by binding the server to a host machine (ilab) and a port number. After this it will wait for clients to attempt to connect to it. Once a client attempts to connect to the server, both entities then establish a connection and a thread is spawned to further continue the transfer of data between one another. The client starts off by setting the operation it wants to perform on a file and the mode the client is in. Depending on the mode the client is in, other clients may have restrictions when it comes to server requests. The client will then send a request to open a particular file and another request to either read, write, or both (read and write) to that file. The server receives these requests and performs the operations from the client on the file. Once the server is done with its operation it will then

communicate with the client stating the results of the operation. When the client receives the results of the operation it will then either throw an error if the operation was unsuccessful or send the server another request if another operation is needed (for example read and write). When all read and write operations terminated the client will proceed send a close request to the server to close the file since the client is finished. When the server closes the file, it will communicate with the client the results of the close request and then go on to terminate the current thread as well as the client terminating its execution. As the spawned thread is created to maintain communication between server/client, the main process continues to wait for incoming client connections.

Design:

Server: The server does not take any arguments at input. It uses getaddrinfo to get the information of the current machine being and a particular port. It then goes on to search for a socket to be able to create and to bind too. Once the server is binded to a particular host and port number it will go into an infinite loop where it will wait for incoming connections and spawn worker threads for each operations the client requests the server to perform. The server will not terminate unless an error occurs that forces the server to shut down or a user exit command.

Client: The client takes 4 arguments at input that will be the server, a file, an operation and a mode. The client then goes on to initiate the connection with the server and requesting to open a file provided at inputs. The client will print the file descriptor for that particular file if open successfully as well as the number of bytes that were successfully read and written to the file by using the net operations. Once there are no more read or write requests to the server, the client will use the file descriptor earlier received to send a close request to the server to close the file being used.

Client operations: The client starts off by attempting to connect to the server by calling netserverinit() that uses getadrinfo to retrieve the server information and establish a connection. The client will then call netopen() and send a string to the server (see protocol) and wait and return a file descriptor from the server if successful and -1 if unsuccessful. Afterwards based on the user input the client will call either or both netread() and netwrite(). Netread() will send a string to the server (see protocol) and wait for a response containing the number of bytes the server was able to read from the file and return that value. Netwrite() will send a string to the server (see protocol) which also contains the string to be written to the file. The client will wait for a response from the server containing the number of bytes written to the file and return that value.

Server operations: The worker threads that are spawned by the server will run the functions netthreadopen(), netthreadread(), netthreadwrite() and netthreadclose(). Netthreadopen will read the request from client which will contain the filename and the flags. The filename and flags are then used to call the open function and retrieve a file descriptor which is then once again sent over the network to the client. If the client is in Transaction mode then the netthreadopen() will fail to open due to the clients lack of permission. Netthreadread() awaits the read request from the client which is then used to retrieve the number of bytes to be read from the file. The number of bytes requested is attempted to be read from the file and once it's done reading the server will send back to the client the number of bytes it was actually able to read. If the client is in transaction mode then netthreadread() will exit due to lack of permissions from the client. With netthreadread() finished the program will move on to netthreadwrite() where the server will await a write request from the client over the network. The message is stored in two variables which consists of the data and the number of bytes to be written to the file. After the data is written netthreadwrite() will send back to the client the number of bytes that were successfully written and exits. If the clients mode is exclusive then only one netthreadwrite() will be performed at a time as the other clients await their turn.

Argument parsing: Argument parsing consists of two helper functions parseInt() and parseString() that allow the program to parse the data that is being sent over the network in the format we assigned. The data is then stored in the respective variables so furthermore they can be utilized in the necessary operations.

Threading: The multi-threading process is achieved by using posix threads in our implementation where the server spawns threads for every operation that a client requests (netopen, netread, netwrite and netclose). To avoid data corruption and undefined behavior, mutual exclusion locks are used in the netthreadread() and netthreadwrite() functions before we perform operations on the files and then unlocked when terminated, to guarantee that multiple clients do not simultaneously attempt to read or write data when in exclusive mode. If the client is in unrestricted mode, no locks are utilized, and the file is accessed in a first come first served basis.

Extensions: We implemented extension A for this assignment. As specified, the user can enter which mode they want to open the file in through the command line as the last argument. The client file we use then sends that to netserverinit, which will save the mode. When opening a file, the client will send the mode as the first argument. The server then saves that with the file descriptor assigned to the client during netthreadopen(the server side of netopen) under the filename. The server uses that later during operations and frees it during netthreadclose.

Difficulties:

We had a massive problem with out-of-bounds memory accesses. This would not have been a problem without the address sanitizer, but our code would have been leakier and buggier. The bugs showed their face when we implemented extension A, which was very frustrating for debugging. Even more frustrating was the fact that the bugs hid when using gdb. Eventually, we went back to a previous version to test if the base code still worked. It now came out that these errors had been around since the base code. After much sweat, tears, and gdb, the bugs were slowly squashed, and the base code finally worked. After that we meticulously copied the tiny but numerous changes into the current version of the code, and it finally worked. Kinda.

Testing:

For testing, we utilized a server, multiple clients and a text file to establish a connection and have a file to perform operations on. With gdb we were able to step by step test the the data being sent and received by both client and server to then parse it and store it in the adequate variables. Then we analyzed and stored different data in our test file to check its content after every operation to test whether or not the operations were successful and the data was actually written or read from the file. Finally we used multiple clients to connect to the server at the same time to check whether or not our threading implementation did not cause conflicts or data corruption while all clients request different operations to the server but on the same file.