README - Systems Programming: Assignment #4

Due on May 1, 2017 at 11:59pm

Professor Fransisco Section #1

Srihari Shankar, Douglas Rudolph

1 Design

First Approach: When writing the program the first time around, my partner and I tried to write it in such a way that we allowed the client to cache file data. The way we achieved this was by splitting up the process of sending the data to the server into two sub-processes:

Sub-process 1:	If the client has never seen the file that is being read and or written to on the server, then the client caches the file name and the permissions of said file on the client, and continues with running the process on the server.
Sub-process 2:	If the client has seen the file before, and the client doesn't have the correct permissions on said file for the queued up operation, we halt the operation and never let the server see the incoming operation. This minimizes the amount of incoming operations on the server.

Even though this approach technically works, we were told to deprecate this build and make *a* server handle all the permission checking. So

Second Approach: On approach number two, we got rid of the caching on the client side and theorized the idea of caching all the data on another server. The reason for this is because we felt that if we are going to make this program modular, we would technically have a database that we can query a look up inside of, rather than having to call upon a horrific data structure that would be considered the opposite of modular.

Even though this idea worked on paper, we were told that everything had to be centralized *one* server.

Final Approach: Our final approach involved putting everything on one server, as well storing a global data structure that would manage all incoming processes. The structure of our server was split into the following structure:

void handle_conn():	The purpose to this function was to take in a single character sent from the client that corresponds to a specific from of functionality that the server will handle with the next incoming piece of data - $chars = \{o, c, r, w\}$: o for open a connection with a client, c for close a connection with a client, r for read a file specified by the client, w write to a file specified by the client.
void _handle_open():	This function is a private function on the server who's purpose to this function is to open a connection with a specific client when called by *void handle_conn()*
void _handle_close():	This function is a private function on the server who's purpose to this function is to close a connection with a specific client when called by *void handle_conn()*
void _handle_read():	This function is a private function on the server who's purpose is to handle any operations related to reading a file from any client given that client has the right permissions and mode. This function is called by *void handle_conn()*.
void _handle_write():	This function is a private function on the server who's purpose is to handle any operations related to writing to a file from any client given that client has the right permissions and mode. This function is called by *void handle_conn()*.

It is worth noting that an once error is thrown, **no further netcall will work from said client**.

2 Extensions

Extension A: When implementing extension A, we went about implementing this component by adding *booleans* to check to see if a given file is in *unrestricted*, *exclusive*, and or *transaction* mode when opening up. On top of this, we added a precondition for when looping through the data structure to make sure that server won't open a file when it is already open in a conflicting mode.

Extension C: When implementing extension C, we decided to use a semaphore to queue up the instructions on the server for every file. Whats nice about this approach is that the semaphore takes in the next operation, and the next operation will run regardless of the current state of a file after the first client closes.

3 Time Complexity

The actual time complexity of our program without simplifying it down is$O3n$. This is because in the struct that stores all of our information related to a file also stores a linked list of file descriptors that all correspond said file. Also, the same struct queues up the operations. So the wort possible scenario is when a set of clients keeping queuing up different processes on the same file with different file descriptors. This means that the server would have to loop through the file linked list, then loop through the file descriptor linked list, then dequeue n operations.Thus, the worst case simplifies down to $O(n)$.