Documentation Assignment 3: A Better Open

Problem:

In this project, we were tasked with writing a separate set of file commands that will contact a file server to perform read(), write(),open(), and close() commands. The project uses sockets and other native structures to facilitate the communication between the client and server files. The client takes an input argument with the file pathname and the command (ie. read(),write(),open(), and close()), and sends it to the server, which then carries out the command and sends a response message back to the server. Our file server also provides three types of file connections: unrestricted mode, exclusive mode, and transaction mode.

Approach:

In the main function, the client takes the inputs of a file pathname and the command as well as any other parameters that are unique to the command. Once the command and the appropriate parameters have been received, *createSocket()* creates a netsocket to connect with the server_socket on the server. The input parameters are then concatenated together in a buffer to create the client message, which is then written to the netsocket to be sent to the server. On the server socket, *createServerSocket()* creates the server_socket and defines the server address. The socket is then bounded to the address and listens for connections. *handleRequest()* reads the client message from the clientsocket and tokenizes the message to get the first input argument which is supposed to be the command. Since each command is associated with a unique number between 1 to 4, the appropriate function for each command is called accordingly. The *handleOpen()*, *handleRead()*, *handleWrite()*,*handleClose()* functions all tokenize the client message into the separate input parameters and then call the corresponding function (ie. open, read, write, close). If it is successful, the server then writes back to the client socket with a success message. For *handleRead()*, the server will also send the client a buffer with the bytes read. If it is unsuccessful, the server writes back to the client socket with an error message. The server is also keep track of all of the open fds that are currently in use. Each time *handleOpen()* is called and a new file is opened, an fdNode is create and added to a linked list with the file path, file and open modes and client and server fds. Each time *handleClose()* is called, the corresponding fdNode is removed from the linked list. As part of extension A, *handleOpen()* also takes in an additional parameter of fileMode, which indicates whether the file will be opened in unrestricted, exclusive or transaction mode. For extension C, we created a structure called QueueNode to hold all the pending file operations, and are deleted as the file operations are being completed on a specific file. Additionally, the structure helps us determine which operations are ready to be executed along with which operations have been waiting for more than 2 seconds. As part of extension D, we have a monitor thread* that runs every three seconds and cleans up any QueueNodes that have been waiting for more than 2 seconds and reports a time out error to the client that requested the operation.

*Extension D: server will only report a time doubt error only if a second client requests access

Defined Structs:

`struct fdNode`

char* path - holds path of file being used

int fileMode - holds file mode being used (ie. unrestricted,exclusive, transaction)
int openMode - holds open mode being used(ie. O_RDONLY, O_WRONLY, O_RDWR)
int clientfd - holds fd sent back to client (usually a negative number)
int serverfd - holds fd of file opened

`struct QueueNode`

int valid - flag to check for all operations that have been waiting for more than 2 seconds
time_t secs - keeps track of how much time each operation has been waiting for
pthread_t tid
char* path- holds path of file
int openMode - holds open mode being used
int fileMode - holds file mode being used
int ready- flag to check for operations that are ready
struct QueueNode* next

Client Functions:

`int createSocket()`
Creates the netsocket on the client side to facilitate communication between the client and the server

`int netserverinit(char*hostname, int fileMode)`
Verifies the ip address of the hostname. Returns 0 if hostname exists and -1 if it does not exist

`int netopen(char* path, int flags)`
Function handles the open command on the client side by creating the client message using the user inputted parameters and writing it to the netsocket. After it receives a message from the server, it tokenizes the server message and outputs the fd of the file opened if successful.

`int netread(int fd, void*buf, size_t bytes)`
Function handles the read command on the client side by creating the client message using the user inputted parameters and writing it to the netsocket. After it receives a message from the server, it tokenizes the server message and outputs the number of bytes read if successful.

`int netwrite(int fd, void*buf, size_t bytes)`
Function handles the write command on the client side by creating the client message using the user inputted parameters and writing it to the netsocket. After it receives a message from the server, it tokens the server message and outputs the number of bytes written if successful.

`int netclose(int clientfd)`
Function handles the close command on the client side by creating the client message using the user inputted parameters and writing it to the netsocket. After it receives a message from the server, it tokens the server message and outputs 0 if successful and -1 if unsuccessful.

Server Functions:

`int createServerSocket()`
Creates the server_socket on the server side to facilitate communication between the client and the server

`void* handleRequest(void* arg)`
Function tokenizes first parameter of the client message to determine the command. Each command is given a different number, and based on the number the corresponding function is then called.

`void handleOpen(char* cmessage, int client_socket)`
Function handles the open command on the server side. The client message (cmessage) is tokenized into its inputted parameters- open, and fd, and based on those parameters, the open function is called. The server then writes the outputted fd to the server_socket to then send to the client.

`void handleRead(char* cmessage, int client_socket)`
Function handles the read command on the server side. The client message (cmessage) is tokenized into its inputted parameters- fd, char buffer, bytes requested - and based on those parameters, the read function is called. The server then writes the number of bytes read to the server_socket to send to the client.

`void handleWrite(char* cmessage, int client_socket)`
Function handles the write command on the server side. The client message (cmessage) is tokenized into its inputted parameters- fd, char buffer, bytes requested - and based on those parameters, the write function is called. The server then writes the number of bytes written to the server_socket to then send to the client.

`void handleClose(char* cmessage, int client_socket)`
Function handles the close command on the server side. The client message (cmessage) is tokenized into its inputted parameters- fd - and based on those parameters, the close function is called. The server then writes the success message to the server_socket to then send to the client.

`void insertfdNode(fdNode* node)`
Function inserts an fdNode into the linked list each time open is called on a new file. This is to keep track of all of the opened fds.

`int getFreeClient()`
Function outputs a negative number other than -1 to send back to the client side during `handleOpen()`. The negative number is the clientfd.

`fdNode* get_Node_from_cfd(int clientfd)`
Given a clientfd, the function traverses through the linked list to find the fdNode that contains the clientfd.

`fdNode* get_Node_from_path(char* path, fdNode* start)`
Given a pathname, the function traverses through the linked list to find the fdNode that contains the path.

`void deletefdNode(fdNode* node)`
Function deletes an fdNode from the linked list each time close is called on a file descriptor. This is to keep track of all of the opened fds.

`void insertQueueNode(QueueNode* node)`
Function inserts file operation as new QueueNode to queue

`void removeQueueNode(QueueNode* node)`
Function removes file operation when it is completed or has been waiting for more than 2 seconds