ICS 53, Fall-2022
Assignment 6: Multi-threaded File Server

You will write a **networked client/server application** that allows clients to remotely access files. The structure of the client program is similar to Assignment 5. A server is running on a machine which contains files needed by the client. A client connects to the server and sends commands to the server to perform file operations. The server performs the operation and sends back the result to the client. The server program must be multi-threaded so multiple clients can perform file operations at the same time. You can assume that no more than 4 clients will ever connect to the server at one time.

1. Running the Client/Server

The application is actually two different programs, the client program, and the server program. These two programs should both be running as two separate processes which are invoked by the user in a shell. These two programs may be running on the same machine but they may be running on two different machines. Since the client and server are communicating using TCP/IP, you will need to select a port to use for communication. ICS Computer Support has advised us that ports 30000 – 60000 are all available for your use, so you can use any one of those ports.

2. Commands Summary

Your program should accept the following commands related to the file operations:

- **openRead**: opens a file for reading, allowing the user to perform read operations. This command takes 1 argument, the name of the file on the remote server to open. The file is assumed to exist on the server in its local directory. The user cannot perform read operations until the file has been opened for reading using the openRead command. The user can only have one file open for reading at one time. If the client attempts to open a file for reading while the same client has another file already open for reading, the error "A file is already open for reading" should be printed on the screen of the client.

- **openAppend**: opens a file for appending, allowing the user to append characters to the end of a file. This command takes 1 argument, the name of the file on the remote server to open. The file is assumed to exist on the server in its local directory. The user cannot perform append operations until the file has been opened for appending using the

openAppend command. The user can only have one file open for appending at one time. If the user attempts to open a file for appending while another file is already open for appending, the error “A file is already open for appending” should be printed on the screen of the client. If the file is currently open for another client then the error “The file is open by another client.” should be printed on the screen of the client.

- **read**: reads bytes from a file. This command takes 1 argument, the number of bytes to read from the file (max bytes read at once = 200). The bytes are read starting from the last byte which was read since the file was opened for reading. If the number of bytes requested is greater than the number of bytes left in the file, the server should read all of the remaining bytes in the file. The bytes that are read should be printed on the screen of the client and the command should be printed on the screen of the server. If the file contains no remaining bytes then nothing should be printed on the screen of the client.

- **append**: appends bytes to a file. This command takes 1 argument, a string that contains the bytes to be appended to the file (max bytes appended at once = 200). The bytes are appended at the end of the file, after any previous bytes which may have been appended. This command should not print any output to the screen of the client, but the command should be printed on the screen of the server.

- **close**: This command closes a file so that the client can no longer access the file. The command takes one argument, the name of the file to close. The file which is passed as an argument can be assumed to be open before the close command is issued.

- **getHash**: This command will calculate the Md5 hash of the file as indicated by the argument to the command. This hash is then returned to the client. Note that if a file is open for appending, the server is unable to calculate the hash and the same error message as the one for a client trying to open a file for appending which is already open in append mode should be printed.

- **quit:** This command ends the client, but does not quit the server.

3. File Access Constraints

The following access constraints must be implemented.

- A single client can only have one file open at a time, however, they can still get the hash of multiple files.
- Multiple clients can read a file at the same time and calculate the hash.
- If a client currently has a file open for append then no other client can access that file, even for the hash.
- A client can only open a file for append if that file is not currently being accessed by any other client.

Since the server program is a multi-threaded program and some constraints require exclusive access to the file, you need to implement proper synchronization.

4. Starting the Server

If we assume that the name of the server's compiled executable is "server" then you would start the server by typing the following at a linux prompt,

"./server 30000"

(The argument is the port number that the server will listen to.)

The server will wait for the clients to connect and will create a thread to communicate with each client which requests a connection. The server should be able to handle multiple clients at the same time and the maximum number of clients which will ever attempt to connect to the server at the same time is 4.

5. Starting the Client

If we assume that the name of the client's compiled executable is "client" then you would start the client by typing the following at a Linux prompt:

"./client server.ics.uci.edu 30000"

(The first argument is the domain name of the server and the second argument is the port number the server is listening to.)

Remember server.ics.uci.edu is just an example of the domain name of the machine that the server is running on. If you run both client and server on the same machine, you can use "localhost" instead to loop back the connection between client and server. When the client

starts it should print a “> ” prompt on the screen and wait for input from the user. (Note: “> “ is one > and one space.)

6. Message Format

You can format the messages any way that you want to.

7. User interface of the Client

The client prints a prompt, “> “, on the screen, indicating that it is waiting for the user to enter a command. When the user enters a command, the client will send the command to the server. The server will execute the command and return a response if the command is a read command, or an error message if necessary. Notice that no response from the server is needed if the command is openRead, openAppend, append, or close, unless an error occurs. The client will print the received information to the screen, print a prompt on a new line, and wait for more input from the user. An example of user interaction with the client is shown below. In thie example below, a.txt is a file in the local directory of the server program. It contains:123456<EOF>

> openRead a.txt
// file opened
> read 3
123
> read 2
45
> close a.txt
> openAppend a.txt
> getHash a.txt
A file is already open for appending.
> append 78
> close a.txt
> getHash a.txt
<The digest is printed in a hexadecimal format>
> quit
←*you are back to the linux prompt.*

a.txt after the execution of client’s commands:12345678<EOF>

The client will continue in this loop until the user enters “quit” which will cause the client’s process to exit. (Note: No need to print anything after quit was entered.)

8. User Interface of the Server

The server should print “server started” on its screen when it is started. Once running, the server will accept connection requests and commands from clients. When the server receives a request from a client, the server will print the *requested query command* in the message on the screen on a new line. An example of the printed output of the server when communicating with the client is shown below.

server started
openRead a.txt
read 3
read 2
close a.txt
openAppend a.txt
getHash a.txt
append 78
close a.txt
getHash a.txt

The server will continue responding to requests and printing the associated information until its process is killed externally, for instance by a ctrl-c typed at the keyboard.

9. Example execution

- The “//” means comment and it’s just there for clarifying the events.
- Contents of the files on the server local directory before execution:
  a.txt:12345678<EOF>
  b.txt:abcd<EOF>
  c.txt:efgh<EOF>
- In the example below, the server is started before either of the clients are started.
- The clients are executed concurrently.

Client 1

$ **./client localhost 30010**
> **openAppend a.txt**
> **append abc**
> **append d**
> **openAppend b.txt** //since a.txt is still open for this client
A file is already open for appending
> **getHash b.txt** // works since we are not opening a new file for reading or appending

<hash printed in hexadecimal>
> close a.txt
> **openAppend c.txt**
The file is opened by another client. //since another client is appending to this file
> **quit**
$

Client 2
$ **./client localhost 30010**
> **openAppend c.txt**
> **append 123**
> close c.txt
> **quit**
$

Server
$ **./server 30010**
server started
openAppend a.txt //client 1
openAppend c.txt //client 2
append abc   //client 1
append d //client 1
openAppend b.txt //client 1
A file is already open for appending
getHash b.txt // client 1
close a.txt // client 1
openAppend c.txt //client 1
The file is open by another client.
append 123 //client 2
close c.txt // client 2

^C
$
Contents of the files after execution:
a.txt:12345678abcd<EOF>
b.txt:abcd<EOF>
c.txt:efgh123<EOF>

10. Implementation Details

- All commands entered into the client will be valid commands.
- All files whose access is requested by the user will be files which are present on the server in the local directory in which the server is running.
- If the user attempts to read a file before it has successfully opened the file for reading, the error message “File not open” should be printed on the screen of the client.
- If the user attempts to append to a file before it has successfully opened the file for appending, the error message “File not open” should be printed on the screen of the client.
- If the user attempts to get the hash of a file that doesn’t exist, the error message “File not found” should be printed on the screen of the client.
- If a client attempts to open a file, for reading or appending, while the same client has another currently open, the error “A file is already open” should be printed on the screen of the client.
- If a client attempts to open a file for appending when the file is currently open for another client then the error “The file is opened by another client.” should be printed on the screen of the client.
- A client will never attempt to read more than 200 characters at one time.
- A client will never attempt to append more than 200 characters at one time.
- A client will never attempt to close a file which it does not currently have open.
- The server should never quit once it is started. It should continually listen for new connection requests from clients. The only way to end the server is to kill it manually (for example, with ctrl-C).
- The commands and arguments should be case-sensitive.
- The files are text files rather than binary.
- The zip file provided to you with this assignment contains code that allows you to calculate the hash of a given file. The code is provided with a test code that can demonstrate how you get the hash of a file. Please read the included Readme.md for detailed instructions.

11. Submission Instructions

Your source code must be two C files (client.c and server.c). Be sure that your program must compile on openlab.ics.uci.edu using gcc version 11.2.0. You can check your gcc version with the “gcc -v” command. Make sure both files can compile with “gcc -pthread filename.c”. Submissions will be done through Gradescope. The first line of your submitted file should be a comment which includes your name and UCNETID your partner’s (if you are working with a partner).