Practicum II Rubric & Self-Evaluation

Code Setup	**Exemplary**	**Good**
code is properly divided into header and source files	1	1
code uses functions and functions have appropriate names	1	1
functions have headers explaining signature	2	1
Project Setup	**Exemplary**	**Good**
code is compiled and linked using makefile or similar	1	0
Testing	**Exemplary**	**Good**
code is thoroughly tested	4	2
all test cases pass	1	0
Q1 / GET	**Exemplary**	**Good**
program is called "fget" and properly uses command line args	5	3
paths are recognized for files	5	2
files are transferred and stored properly locally	8	4
missing local file name defaults to remote file name	5	2
remote files are stored on USB device	5	2
USB device name/path is stored in an external config file on server	2	1

client and server communicate properly over TCP/IP	5	3
Q2 / INFO	**Exemplary**	**Good**
proper use of command line arguments	2	1
returns full file information	4	2
file information display in readable format	3	2
output of program can be easily piped to another with |	1	0
Q3 / MD	**Exemplary**	**Good**
proper use of command line arguments	2	1
correct path is created on USB device on server	8	4
Q4 / PUT	**Exemplary**	**Good**
proper use of command line arguments	2	1
local files are transferred and stored remotely	13	8
missing server file name defaults to client file name	5	3
Q5 / RM	**Exemplary**	**Good**
proper use of command line arguments	1	0

file is properly deleted on server	4	2
Q6 / MIRROR	**Exemplary**	**Good**
files are stored in mirrors in separate USB devices	4	2
reads still function when a device is removed	4	2
test cases demonstrates that mirrors work	2	1
Q7 / THREADING (BONUS)	**Exemplary**	**Good**
new thread created for each client connection	5	3
file command run in new thread	5	2
code is thread-safe	5	2
test cases demonstrate that multi-threading works	5	2
Q8 / DISTRIBUTION (BONUS)	**Exemplary**	**Good**
files are read in parallel from distributed devices	25	15
test cases demonstrate that distributed reads work	5	3

		Name:
		Term:
	120	**Section:**

Poor	Points	Code Chunk/Lines
0	1	All Files
0	1	All Files
0	2	All Files

Poor	Points	Code Chunk/Lines
0	1	N/A

Acceptable	Points	
0	4	
0	1	

Poor	Points	Code Chunk/Lines
0	5	Makefile line 11, client_main.c line 18, client_handler.c lines 265-267
0	5	client_handler.c lines 61-71 & 104, server_handler.c lines 393-466
0	8	client_handler.c lines 80-91 & 152-178, server_handler.c lines 60-75
0	5	client_handler.c lines 118-127 & 141-150
0	5	config/drive_path_settings.txt, lines 1-2
0	2	config/drive_path_settings.txt, lines 1-2

	5	client_handler.c lines 26-52 & 203-256, server_handler.c line 591-650, server_main lines 63-108

Poor	Points	Code Chunk/Lines
0	2	client_handler.c lines 61-71 & 267-273
0	4	server_handler.c lines 492-526
1	3	server_handler.c lines 492-527
0	1	common.c lines 158-161, client_handler.c lines 41-51

Poor	Points	Code Chunk/Lines
0	2	client_handler.c lines 61-71 & 273-278
0	8	server_handler.c lines 45-49 & 529-545

Poor	Points	Code Chunk/Lines
0	2	client_handler.c lines 138-184 & 279-280
3	13	client_handler.c lines 153-184, server_handler.c lines 60-76 & 547-572
1	5	client_handler.c lines 138-150

Acceptable	Points	
0	1	client_handler.c lines 61-71 & 281-287

1	4	server_handler.c lines 85-90, 574-590

Acceptable	Points	
1	4	config/drive_path_settings.txt, lines 1-2, server_handler.c 362-389, all of handle_[ACTION]
1	4	server_handler.c 25-35
1	2	server_test.c lines 15-26 & 40-118

Acceptable	Points	
1	5	server_main.c lines 63-108, 168-176
1	5	server_main.c lines 63-108
1	5	server_main.c lines 31-32, 88-90
1	5	client_test_multi.c

Acceptable	Points	
10	0	N/A
1	0	N/A

John Ciolfi
Spring 2023
Schedlbauer, CRN 38920

Justification/Explanation of Approach

There is a client_main.c and a server_main.c that act as entrypoints to fget/fserver. These use methods defined in client_handler.h and server_handler.h to carry out request/response handling
The code is broken down by delegating to functions in the client/server handlers, which also have a number of private helper functions to use as well. There is also a common.h/common.c for code/structures used by both handlers.
There are javadoc-like signatures for each method. Methods defined in header files have signatures written in the header file, and private helper methods have signatures written in the .c file.

Justification/Explanation of Approach

A Makefile is used to build the code with `make`.

Test cases in server_test (to test server-side handlers), client_test_single.sh to test full functionality/invalid inputs, and multi_test to test multithreaded functionality.
The test files listed above all function as expected.

Justification/Explanation of Approach

The program is named fget. The number of command line args are validated (too few/too little) as well.
File paths are taken in from the command line and sent to the server. The client takes in paths to build a request and forward it to the server. The server prepends the drive paths from a config to determine the "remote" path. The client also uses paths to save any files (e.g. with GET).

For GET, the response sends the data for the file, which is written into the file on the client side. For PUT, the request sends the data for the file, which is written into the file on the server side.

The client-side determines how many args were passed and adjusts accordingly. For GET and PUT, if the remote name is omitted, the local name is used for the remote name.
Two 8GB SAMDATA drives were used to store remote files. They were reformatted from FAFT32 to APFS to allow INFO to work properly.
This file is an external config on the server. The paths are read in and set when the server starts, and passed along to relevant functions.

Client/server set up sockets with AF_INET to communicate over TCP/IP. SOCK_STREAM is used as a two-way connection where the client/server can send/receive data from one another.

Justification/Explanation of Approach

The client side verifies INFO was the action and checks the right number of parameters were supplied. It then builds the request with the arguments supplied and sends it to the server.
struct stat file_stat is used to determine the owner ID, last modified timestamp, and permissions. ftell is used to determine the file size.
The timestamp is converted to a YYYY-MM-DD HH:MM:SS format for readability. The permissions are extracted using a bitmask. Each field is separated with commas and has a name before it.

printf is used to display the output of the response from the server, which cna be easily piped.

Justification/Explanation of Approach

The client side verifies MD was the action and checks the right number of parameters were supplied. It then builds the request with the arguments supplied and sends it to the server.

The directory with the given path is created on both drives on the server-side, and checked if they were successful on both drives, neither, or just one of them.

Justification/Explanation of Approach

The client side verifies PUT was the action and checks the right number of parameters were supplied. It then builds the request with the arguments supplied and sends it to the server.

The client sends the data of the file to the server, and the server writes the data into the given filename sent from the client (remote or local depending on the client-supplied arguments). The file being put is checked if it was unsuccessful as well.
The number of arguments is checked to determine whether the server file name should defaul to the client file name. If there are not enough arguments, then the client file name will be used.

The client side verifies RM was the action and checks the right number of parameters were supplied. It then builds the request with the arguments supplied and sends it to the server.

The file path for each drive is determined and removed on the server. The server checks if the file was actually removed (ie if it existed) for each drive and returns an appropriate response based on the resulting call to remove() on each drive.

The drives paths are in a config on the server, and for each handler (e.g. handle_PUT, handle_MD, handle_RM, etc.), the resulting actions are applied to both drives. For read operations, the first drive is checked and used if it is up, otherwise the second drive will attempt to be used.

If a drive is down, the other drive will attempted to be used to read from.

There is a server-side test suite that checks if the file actually appears on both drives, and outputs if it does/not. This is used to check MD, PUT, and RM in particular.

Every time there is a new connection, a thread is made and sent to a function to handle the client's request. There is a maximum limit on the number of connections as well. handle_client is the function to handle a client's request. This includes receiving a response from a client, parsing it, performing any tasks, and returning a response. There is a lock on parse_request which will access the drives. To ensure consistency, no two threads are able to access the drives at the same time. However, threads are able to serialize responses and send to their client in parallel.
There is another test suite to create multiple threads and run different actions on the server. There are multiple threads that will try to GET/PUT a file onto the server at around the same time. As discussed in the README, this is best demonstrated by adding a delay for the server to show that multiple clients can be connected at the same time, unlike the single-threaded approach.

N/A

N/A