OS 2019 – Exercise 5
FileApp

Supervisor – Dr. Neta Rozen schiff

Due to: **28.6.2019**
Note: This exercise is a bonus exercise.

In this exercise, you are required to create a simple file transfer application which we name FileApp. FileApp consists of server and clients communicating over TCP. FileApp code should be compiled into one executable file, where command line arguments dictate a client or server operation. In order to transfer a file, FileApp should be executed first as a server which waits for clients, and then in another process and/or computer the FileApp should be executed as client instructed to connect (using IP address and port) to the FileApp server.

Based on the command line arguments, FileApp can only be executed in one of following 3 ways:

1. A server (indicated by –s), provided with the local directory where uploaded files are stored (local_dir_path) and the (TCP) port number (port_no) it should listen for clients` requests:
   FileApp -s <local_dir_path> <port_no>

2. A client single upload (indicated by –u) provided with the path of the local file that should be sent (uploaded) to server, the remote name that will be used to store the file at the server, the port number of the server and the IP address of the server:
   FileApp -u <local_path> <remote_name> <port_no> <server_IP>
   The exit code of the client process should be zero if and only if the upload was successful (it got confirmation from the server that it received and stored the entire file).

3. A client single download (indicated by –d) provided with the local path where to save the file received (downloaded) from the server, the remote name that will be requested from the server, the port number of the server and the IP address of the server:
   FileApp -d <local_path> <remote_name> <port_no> <server_IP>
   The exit code of the client process should be zero if and only if the download was successful.

Server

The server's rule is to listen for incoming request. These requests can be: upload file from the client or download a file.

Notes:

- The server receives a port number and listens to this port. It waits for clients to connect to this port (using TCP connection) and serves their requests (more details below).When the server receives a request it needs to parse it and executes the request.
- In the real world when a server receives a new request from a client it must stay responsive for other clients. In this exercise, you are not required to do so.
- During the server's activity, it should not have any memory leaks. It means that when a connection with a client is terminated, the server should release all the relevant resources.
- The maximal number of pending connections is 3 (the backlog parameter in the listen function).
- Due to security considerations, the server should not allow access for files outside the local_dir_path of the server (an accepted solution is to refuse to serve a client that refers a remote name that includes the character "/").
- The server should make sure that the resulting file path (combination of server's local_dir_path with client's remote_name) is valid: at most 4095 chars and the file name part at most 255 chars.
- The server should not crash upon receiving illegal requests.
- When "quit" is typed into stdin, the server should stop handling more clients, finish what it is currently doing, release all resources and exit. You may find FD_SET and select useful for this purpose.

Messages to the user:

- After a successful bind, the server would print its' IP: "server bind IP: %s".
- Whenever the server is not handling a client (before and after dealing with a client requests), the server would print "Waiting for a client...".

- After a successful connection of a client the server should print
    - Client IP: <ip>
    - command: <d/u>
    - filename: <filename>
    - file_path: <path>

  (where file path is combination of server's local_dir_path with client's remote_name).
- In case of unsafe file name or invalid file path the server should print "filename error!" and report it to the client (which should print the same message as well).
- When the server can not open a file (for example when client requests to download a file which do not exists), the server should print "my file error!", and report to the client (so it can print "remote file error!".
- If the client's command completed successfully it should print "success" otherwise "failure"

Client

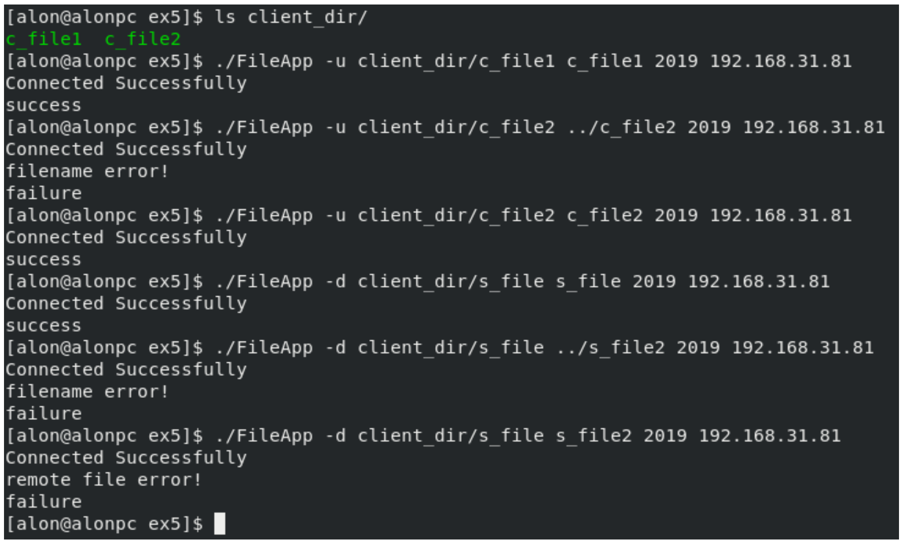
Upon execution, the client will try to connect to the specified server. After a successful connection, the client will print "Connected Successfully.".

Messages to the user:

- When the client cannot open a file (for example when client requests to upload a file which do not exists), the client should print "my file error!", and report to the server (so it can print "remote file error!".
- When the client receives a report from the server that there is a problem with file name (for example not safe or beyond maximal length) it should print "filename error!".
- When the client receives a report from the server that there is a problem with opening file (for example if the requested file does not exist) it should print "remote file error!".
- If the command completed successfully it should print "success" otherwise "failure".

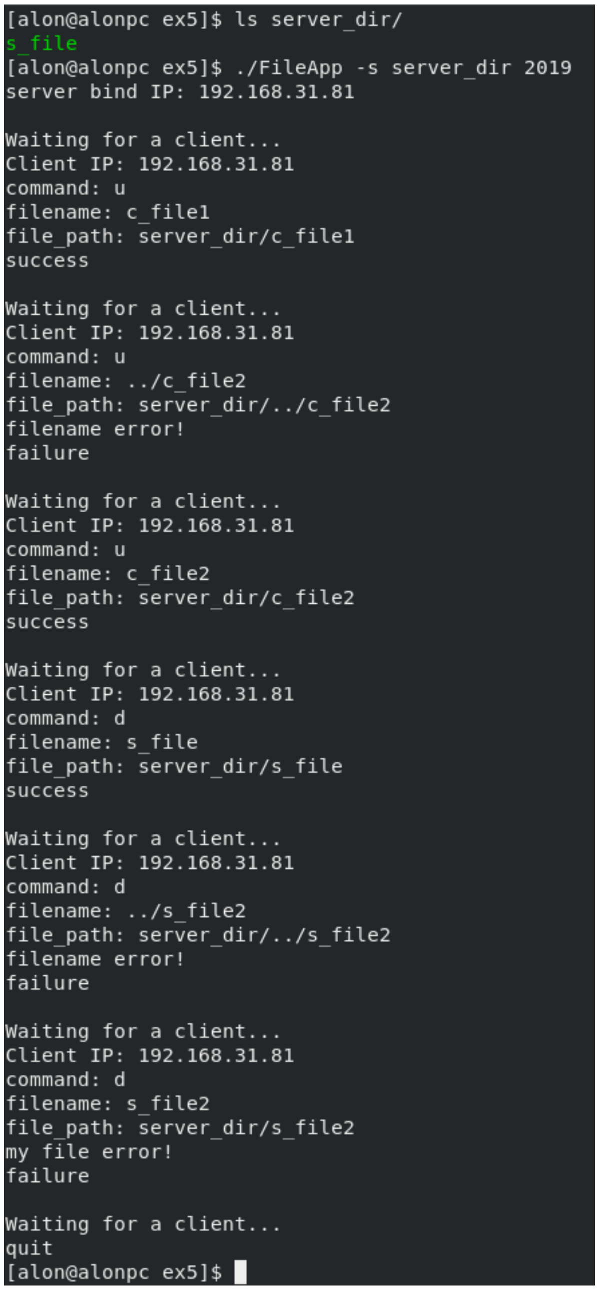
Example

Client:



```
[alon@alonpc ex5]$ ls client_dir/
c_file1  c_file2
[alon@alonpc ex5]$ ./FileApp -u client_dir/c_file1 c_file1 2019 192.168.31.81
Connected Successfully
success
[alon@alonpc ex5]$ ./FileApp -u client_dir/c_file2 ../c_file2 2019 192.168.31.81
Connected Successfully
filename error!
failure
[alon@alonpc ex5]$ ./FileApp -u client_dir/c_file2 c_file2 2019 192.168.31.81
Connected Successfully
success
[alon@alonpc ex5]$ ./FileApp -d client_dir/s_file s_file 2019 192.168.31.81
Connected Successfully
success
[alon@alonpc ex5]$ ./FileApp -d client_dir/s_file ../s_file2 2019 192.168.31.81
Connected Successfully
filename error!
failure
[alon@alonpc ex5]$ ./FileApp -d client_dir/s_file s_file2 2019 192.168.31.81
Connected Successfully
remote file error!
failure
[alon@alonpc ex5]$ 
```

Server:



```
[alon@alonpc ex5]$ ls server_dir/
s_file
[alon@alonpc ex5]$ ./FileApp -s server_dir 2019
server bind IP: 192.168.31.81

Waiting for a client...
Client IP: 192.168.31.81
command: u
filename: c_file1
file_path: server_dir/c_file1
success

Waiting for a client...
Client IP: 192.168.31.81
command: u
filename: ../c_file2
file_path: server_dir/../c_file2
filename error!
failure

Waiting for a client...
Client IP: 192.168.31.81
command: u
filename: c_file2
file_path: server_dir/c_file2
success

Waiting for a client...
Client IP: 192.168.31.81
command: d
filename: s_file
file_path: server_dir/s_file
success

Waiting for a client...
Client IP: 192.168.31.81
command: d
filename: ../s_file2
file_path: server_dir/../s_file2
filename error!
failure

Waiting for a client...
Client IP: 192.168.31.81
command: d
filename: s_file2
file_path: server_dir/s_file2
my file error!
failure

Waiting for a client...
quit
[alon@alonpc ex5]$ 
```

Guidelines

- You can assume the executable file's input is correct, and is one of the above commands.
- Numbers should be sent in network bytes order, as learned in class.
- Both client's and server's address should include AF_INET family, as taught in class.
- Don't forget to check the return value of all system calls. Especially, note that return values of read and write functions are 0 in case the socket is close from the remote side.
- We are not supplying you with any headers (other than prints.h), or defining the application layer protocol (over TCP), you should use to implement the server and client - that is up to you as long as you make sure the output is correct.
- The header prints.h is supplied to make sure you use the correct format for the messages the client and server are supposed to print. Make sure to use its values (as is) for printing the required text messages to the terminal, so that you don't fail the automatic testing.

Submission

Submit a tar file containing the following:

- A README file. The README should be structured according to the course guidelines. In order to be compliant with the guidelines, please use the README template that we provided.
- The source files for your implementation of the application.
- Your Makefile. Running *make* with no arguments should generate the *FileApp* executable. You can use this Makefile as an example.

Make sure that the tar file can be extracted and that the extracted files do compile.

Late submission policy						
Submission time	**28.6, 23:55**	**30.6, 23:55**	**31.6, 23:55**	**1.7, 23:55**	**2.7, 23:55**	**3.7**
Penalty	0	-3	-10	-25	-40	-100