PROGRAMMING ASSIGNMENT 2
Due: October 20, 2020 11:55 pm
NOTE: This assignment is NOT a group assignment, but an individual assignment.

Objective
To create a HTTP-based web server that handles multiple simultaneous requests from users.

Foreword
Please take time to thoroughly read and understand the document. Only concepts, directions and objectives will be provided in this document. The assignment will teach you the basics of network programming, client/server communication message structures, and will provide a step toward building high performance servers. You must work individually on this assignment; we will check for plagiarism.

Background

What is a web-server?
A web server is a program that receives requests from a client and sends the processed result back to the client as a response to the command. A client is usually a user trying to access the server using a web browser (Google Chrome, Firefox).

The HTTP request consists of three substrings – request method, request URL, and request version. All three parts should be separated by one or more white spaces.

The request method should be capital letters like "GET", "HEAD", and "POST".
The request URL is a set of words which are case insensitive and separated by "/" and the server should treat the URL as a relative path of the current document root directory.

The request version follows the rule like "HTTP/x,y" where x and y are numbers.

Here is an example to make things clearer –
If you type an URL like http://www.w3.org/Protocols/rfc1945/rfc1945 in a web browser, the web browser will send an HTTP GET command to the server http://www.w3.org/ after establishing a TCP connection with port 80 on the server. In this scenario, the format for the HTTP GET command received at the web server is as follows:

```
GET /Protocols/rfc1945/rfc1945 HTTP/1.1
```

Based on the rules, we can identify three substrings like the following:
Request Method: GET
Request URI: /Protocols/rfc1945/rfc1945
Request Version: HTTP/1.1

Deliverables in detail -

Your server should start with port number. For example, if you run the webserver with port number 8888, then: ./webserver 8888

Your server should be running in a forever-running-loop once it has started. You are expected to exit out of the server gracefully when pressing the escape sequence (Ctrl + C or any key(s) of your choice).

In this assignment, the Web server will have a document root which would contain files with the extension ".html" ".htm", ".txt", ".jpg", ".gif". When the web-server receives a HTTP request from a client (from a web-browser such as chrome) for a particular file, it should open the file from this document root and then send the file back to the client with proper header information. Header information is the required to be sent along with the file so the web-browser would understand that the HTTP request was successful and the response is being received. In all cases, the names of files in the **Request URL** should be interpreted in the context of the current document root. If the server interprets the request (HTTP GET) and the requested file exists on the specified location, the server needs to send both **Content-Type** and **Content-Length** followed by an appropriate status code. The file content should be separated by a new line (meaning that you have to separate between the header and your file content by "\r\n\r\n"). For example, when the client requests an existing ".html" file on the server, such a file exists in my server, so server replies with a 200 status code.

The Status Code "200" indicates that the requested file exists and the server is going to send the processed data (the requested file) back to the client. The string next to Status Code "200" conveys the information that the requested document that will follow in this reply. Usually, many Web servers use "Document Follows" and "Ok" for that field.

Sample Header Format-

HTTP/1.1 200 Document Follows
Content-Type: <> # Tells about the type of content and the formatting of <file contents>
Content-Length:<> # Numeric value of the number of bytes of <file contents>
<new line>
<file contents>

In your program, the send buffer would have the following content for the sample header above:
"HTTP/1.1 200 Document Follows\r\n Content-Type: <>\r\n Content-Length: <>\r\n\r\n<file content>"

Content-Type:
The following content types need to supported in order to display index.html under /www directory, which you can unzip from the www.zip provided. Please download the updated www.zip as the previous version had old links which are not available anymore.

.html text/html
.txt   text/plain
.png   image/png
.gif   image/gif
.jpg   image/jpg

.css   text/css
.js   application/javascript

Default Page:
If the Request URL is the directory itself, the web server tries to find a default web page such as "index.html" or "index.htm" on the requested directory. What this means is that when no file is requested in the URL and just the directory is requested (Example: GET / HTTP/1.1 or GET /inside/ HTTP/1.1), then a default web-page should be displayed. This should be named either "index.html" and it should be present in the corresponding directory of the Request URL. The default web page and document root directory should be in the sub directory /www from where your webserver program runs. If the client sends a HTTP/1.0 request, the server must respond back with a HTTP/1.0 protocol in its reply, similarly for HTTP /1.1. HTTP/1.0 and HTTP/1.1 protocols are what you will be supporting. HTTP/1.0 is described in RFC 1945. The client browser will do the multiple requests automatically and nothing needs to be done in the server other than sending correct responses.

Handling Multiple Connections:
When the client receives a web page that contains a lot of embedded pictures, it repeatedly requests an HTTP GET message for each object to the server. In such a case the server should be capable of serving multiple requests at same point of time. This can be done using **select(), fork(), or pthread()** with following approaches:
1.   A multi-threaded approach that will spawn a new thread for each incoming connection. That is, once the server accepts a connection, it will spawn a thread to parse the request, transmit the file, etc.
2.   A multi-process approach that maintains a worker pool of active processes to hand requests off to and from the main server.

Error Handling:
When the HTTP request results in an error then the web-server should respond to the client with an error code. In this assignment, all error messages can be treated as the "500 Internet Server Error" indicating that the server experiences unexpected system errors. A 500 error is not the client's fault and therefore this error code conveys that it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response. It results in the following messages.

HTTP/1.1 500 Internal Server Error
These messages in the exact format as shown above should be sent back to the client if any of the above error occurs.

Extra Credits

Pipelining (10 points)

Supporting persistent connections and pipelining of client requests: You will need to add a logic to your web server to determine when it will close a "persistent" connection - say 10 seconds. That is, after the results of a single request are returned (e.g., index.html), the server should by default leave the connection open for some period of time, allowing the client to reuse that connection socket (print your socket descriptor integer id to observe this) to make subsequent requests. You can use 10 seconds for this timeout value.

The server must send its responses to those requests in the same order that the requests were received. So, if there is a **"Connection: Keep-alive"** header in the request then a timer has to start pertaining to the socket which had received this request. The server will keep that socket connection alive till the timeout value ends. The server should close the socket connection only after the timeout period is over if no other requests are received on that particular socket. If any subsequent requests arrive on that socket before the timeout period (essentially pipelined requests), the server should reset the timeout period for that socket back to the timeout value and continue to wait to receive further requests. If there was no **"Connection: Keep-alive"** in the request header, then the socket will be closed immediately; the timer will never start for this case. For example, if the server received this client request and the message is something like this:

```
GET /index.html HTTP/1.1
Host: localhost
Connection: Keep-alive
```

The response for this request should be something like the following:
```
HTTP/1.1 200 OK
Content-Type: <>
Content-Length: <>
Connection: Keep-alive
<file contents>
```

If there is no **"Connection: Keep-alive"** header in the request message, the server should close the socket upon responding and reply with "**Connection: Close**" header line. If there is a **"Connection: close"** header in the request message, the server should again close the socket upon responding and reply with the same "**Connection: Close**" header line.

Example Scenario:
Consider a request is received by the server with the **Keep-alive header**. At this point a timer is triggered. Now if another request from the same client socket is received after 2 seconds of the first request, the server will process the new request and reset the timer be reset to 10 seconds again. If no other request is received by the web-server within the 10 second timeout period, then the socket connection is closed by the web-server.

Testing pipelining on the Webserver:
Command to be typed on terminal:

If your system has 'telnet'

(echo -en "GET /index.html HTTP/1.1\r\nHost: localhost\r\nConnection: Keep-alive\r\n\r\n"; sleep 10; echo -en "GET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n") | telnet 127.0.0.1 8888

If your system has 'nc'

(echo -en "GET /index.html HTTP/1.1\r\nHost: localhost\r\nConnection: Keep-alive\r\n\r\n"; sleep 10; echo -en "GET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n") | nc 127.0.0.1 8888

Idle Response:

```
Trying 127.0.0.1...
Connected to localhost.lan.
Escape character is '^]'.
HTTP/1.1 200 OK
Date: Sun, 24 Sep 2017 17:51:58 GMT
Content-Length: 62
Connection: Keep-Alive
Content-Type: text/html

<html>
   <body>
      <h1>test</h1>
   </body>
</html>
HTTP/1.1 200 OK
Date: Sun, 24 Sep 2017 17:51:58 GMT
Content-Length: 62
Connection: Keep-Alive
Content-Type: text/html

<html>
   <body>
      <h1>test</h1>
   </body>
</html>
```

Supporting POST method (10 credits)

Implement your code to handle POST requests as well for **'.html'** files. When you send the POST request, you should handle it the same way you have handled GET and return the same web page as you returned for the GET request. (In this case: the requested POST URL). But in this reply webpage, you should have an added section with a header "<h1>**POST DATA**</h1>" followed by a <pre> tag containing the POST data. The **POST DATA** is everything in the POST request following the first blank line.

Earn criteria: Should be able to see the POST data in the server's response along with the requested URL's contents.

Example:
Post request: Content-length defines no. of characters to be read after the blank line
```
POST /www/sha2/index.html HTTP/1.1
Host: localhost
Content-Length: 9
<blank line>
POST DATA
```

Sample post request made to webserver while testing through telnet:
(echo -en "POST /index.html HTTP/1.1\r\nHost: localhost\r\nConnection: Keep-alive\r\n\r\nPOSTDATA")
| telnet 127.0.0.1 8888

You can use nc instead of telnet (the recent version of MacOS doesn't have telnet):
(echo -en "POST /index.html HTTP/1.1\r\nHost: localhost\r\nConnection: Keep-alive\r\n\r\nPOSTDATA")
| telnet 127.0.0.1 8888

Sample response sent by webserver over telnet, Success response:
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
HTTP/1.1 200 OK
Content-type: text/html
Content-size:3391
<html><body><pre><h1>POSTDATA </h1></pre><!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml> **so on and the rest of your HTML file.**

Submission Requirements:
1. Please submit the code for Web Server and README in the format
   <your_identity_key_PA2.tar.gz> after you tar-gzip the files.
2. Include short comments in your web server code and explain its proper usage in the README file.
   The readme file must explain what you have done and how to run it. The documentation does not
   have to be long, but does have to be very clear. Mention how you execute your program and which
   language you have used in that file.
3. The code should serve all the file formats in the provided document root directory (www.zip) and
   the client should receive these files when requested.
4. Simultaneous requests would be sent to the server. Both ways: through 2+ clients sending messages
   to server at the same time; or one client sending multiple requests at the same time. Your server
   should handle simultaneous requests.
5. Pipeline support should be present in the code if that extra credits are attempted.
6. Usage of any libraries for HTTP server is not accepted.

Testing your Web server:
In order to check the correctness of the server, you have to test your server with any of commercial Web
browsers such as FireFox or Internet Explorer. You have to test your web server with the sample document
root directory provided (www.zip).
You can send a request to http://localhost:8888/index.html from your browser and check for response. The
response should be ideally displayed on your web browser.

Helpful Links:
1. http://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html Socket Programming - TCP
   echo client and server with fork().

2. http://www.binarytides.com/server-client-example-c-sockets-linux/ : TCP echo client and server with threads.
3. http://www.csc.villanova.edu/~mdamian/sockets/echoC.htm : Echo server-client code with threads.
4. https://www.youtube.com/watch?v=eVYsIolL2gE : Basics of socket programming.