INET 4021 – Lab #1
HTTP 1.0 server with CGI and Concurrency
May work in groups of up to 2 people

The goal of Lab #1 is to build a HTTP web server based on RFC/1945 (HTTP/1.0). To accomplish this you will be using the standard C socket library. Your server will be build based on not just the C socket library, but also with algorithms and abstraction we have discussed in class as well as in the textbook. The reason for the algorithms is to create the most efficient server. In order to create an efficient server, it is not just about speed, but also resources on the machine (if you create the fastest server in the world, but only 1 client can connect to it, it is not very efficient). This web server will be written in the C language only and you will need to use only the socket API/library (no abstracted libraries/etc).

Your server will require a configuration file for the following options:

- Number of simultaneous connections
- ROOT directory to start looking for html files
- INDEX filename if none given (e.g. index.html)
- PORT to run your server on (remember you need to be root for < 1024)

The above options will need to be in a configuration file called httpd.conf in your conf directory.

Your website will use the template provided for html and graphics as well as cgi scripts that you will need to incorporate into your html and cgi-bin directories:

Here is a description of what should be in which directory:

```
/index.html          /* starting file in the root of your directory */
/cgi-bin/            /* where your executable webserver will reside */
/src/                /* where your source code will reside */
/images/             /* obviously images, but this may depend on your site */
/logs/               /* logfiles to show access as well as errors */
/conf/          /* where your configuration file(s) reside */
```

Your server must conform to the following sections:

Sections 1-3 of the RFC will help define what the specification is about as well as grammars and syntax for the HTTP/1.0.

- Section 4 - you will need to know about all sections since they define message
- Section 5 deals with the request, and you will only need to use a "GET" method.
- Section 6 deals with the response, you will need to get the status-line and general header (simply just print out what you get).
- Section 7 is about the entity of the response, which you can assume that the content type you are interested in is only text/html or text/plain. You can see this type of information in a browser by finding the "page info" or properties of the page in
- Section 8 will define the GET method
- Section 9 will define the STATUS code returned in a response
- Section 10 defines many attributes used in both a request and a response
- Section 11 defines authentication (which is not required for this assignment)
- Section 12 defines security (which is not required for this assignment)

You will need to connect to your server with a web browser (such as Firefox), and send down data that is requested from the user (the Full-Request HTTP-message) back to that browser. This means that if there are graphics in the page, you will need to send those from your server also. Keep in mind that a web server is really nothing more than a file server, and when a client connects to the server and requests a "page" that page is made up of text/html as well as image/gif for graphics. Each "page" request may turn out to be many iterative calls to your server, the first one for the html, and subsequent calls for the images.

You will not need to be able to support all graphics types, but at least gif and jpg.. In order to send a “graphic file” down to a web browser you will need to create a Full-Response with a Status Line and the other headers as well as the Entity-Body. An example of a GIF file would look like this:

HTTP/1.0 200 OK // this is the status line
Content-Type: image/gif // header information
Last-Modified: Mon, 25 Apr 2005 21:06:18 GMT
Expires: Sun, 17 Jan 2038 19:14:07 GMT
Date: Thu, 09 Mar 2006 00:15:37 GMT

GIF89a<96>^@7^@÷^@^@T<87>ótvwnY^H·Ìô^EMëÉØö^T,`âJ5<9c><9b><9b>úõÐ×«^Ah
ÚnÑ'^P«^U^A©<94><8c>èèè¤¤£üüüÎÎÎæ¶^Ay<89>–<8e>Ø<92>æææN<85>QÔÔÔ<8a><99>
1⁄4^AEÛúúú^SF¶üÊ^Cñ<85>u<92>^R^A^B;¹s§øÚÚÚòiVò<93><85><96><96><96><8d>q^C<
86>nkÝÝÞ^QQ^Tººº Nm±°°

.....this goes on for a while, this is the actual file.....

So to send down a binary graphics file you will need to construct a header to explain to the browser what type of file it is, and simply append the data in the message.. Let the browser do the rest. If it helps you can define mime types and response headers in either a C header file “.h” or in your configuration file.

Once you have completed your server you should be able to have a HTML file with some graphics embedded in it, and your browser should display the web page as if it were posted on an Apache or IIS web server... This will be tested for your grade as well as the rest of the functionality.

NOTE: keep in mind the CRLF and other newlines that need to be a part of the header and responses.

LOGS

You will need 2 logfiles in your logs directory, an access log and an error log. Your standard out will go to your access log (who is asking for what) and all errors will go to the error logfile.

CGI Functionality / Concurrency

Based on your functioning http 1.0 server, you will need to add CGI (common gateway interface) and a form of concurrency that could be threading, fork/exec, or asynchronous I/O. The goal of this part of the lab is to take the next step in web server history and add functionality that allows for dynamic web interactions through the ability to add backend applications that will take data from a HTML form.

CGI is the addition of any program/application that can run in the webserver operating system that will take in arguments from a HTML form and will return HTML similar to what you did with files in your lab #1. These applications can be written in any language and they simply need to be executable (meaning that you can use compiled or interpreted code as long as the webserver, and user starting the webserver, has permissions and access to the executable). As a simple example, you could have a HTML form that will take in 2 integers and your CGI program/application will accept in these 2 arguments and add them together and then simply return/write a HTML document back from the CGI program itself. The HTML that the application writes back can be hard coded in your CGI, or you can use a template and fill in the answer from a variable, it really is up to you on how to accomplish the HTML write back to the client that made the request.

CGI will be added to your server by allowing an executable to be invoked upon HTML rendering as servers like Apache do today. CGI scripts can be any executable code whether it is compiled or interpreted, and there is no language requirement either, so you can use Perl, shell scripts, C code, Java/etc.... Basically your server will need to know that the request is not a simple file transfer (such as .html and graphics) that your simple http 1.0 server does by either a directory or extension on the URL request.

An example would be:

http://server/cgi-bin/cgi_script (directory)

or

http://server/myCode.cgi (extension)

Once you know that you will need to invoke a CGI program you will need to parse out that request and use a fork/exec to simply run the program, take in parameters, and write HTML back to the browser through the socket. Parameters are sent to the CGI application through either a GET or POST request. Both need to be implemented and the difference between them are that a GET request is simply to add the request to an environmental variable (Apache uses query_string), and a POST method puts the request in the "data" section of the request. Once the request has been received along with the parameters, the CGI application will simply run and is responsible for sending back the appropriate HTML to the browser through the socket. So in short, your http server will take in the request like it does for .html and .gif/.jpeg files, but you will be able to know that the request is for CGI due to something like the directory, an extension on the executable called .cgi, anything that will allow your server code to know that you will not be simply opening a file and sending the file data back to the browser, instead you will be simply "running" the code specified in the URL request, and the code is responsible for sending back a well formed HTML document.

CGI scripts allow a website to become more interactive through things like forms that will pass data to an executable (any kind of executable) and that executable will send back HTML back to the client socket it was passed. Keep in mind that when fork/exec is called, the child process (the cgi executable) will have access to the client socket descriptor since it has an ancestral relationship with the parent process. NOTE: the parent process should simply close the client socket descriptor, and when the CGI executable is done, it will be closed upon exit.

You will also need to add the 5xx (500s) codes for header responses for failed exec calls to your CGI program/application. It is a common issue to have the CGI fail due to the loose integration between the CGI program and your webserver.

Your CGI program will need to pass in at least the last line of Content-Type (shown below) to end the header (again, with 2 carriage returns!).

There are many ways to send data back to the client in a well formed HTML document. The most simplistic, but hardest to manage is to hard code the html in the executable. A better way to handle this is to use template files that the CGI process will use as a “header (not the HTTP header, your server will still do this)”, “body”, and “footer” model where the dynamic response from the CGI form will be embedded in the body somewhere, but the rest can come from pre-built html files. Keep in mind that the “dynamic” response to the query request is usually only in a specific section of the html document, so there is no need to have all of that data in the executable.

In order to create concurrency as well, you can know that CGI itself will be concurrent since the fork/exec will act as a method to create concurrency and the server can go back to accept, but what about the normal html files? You will need to incorporate complete concurrency by using fork/exec (no exec for non-cgi), threading, or async I/O for all other non-cgi requests.

CGI Issues to address

1. How to pass data to CGI program

Answer: Set QUERY_STRING in environment using setenv() or putenv()

2. How to write data from CGI back to client

Answer: close(1) // This will close standard out STDOUT

dup2(ssock, 1); // This will copy client to STDOUT

Now your CGI will just use print/write/etc to STDOUT

3. How to set 200 OK message

Answer: 500 Server Error // This will come from an exec fail in parent

print “Content-type text/html\n\n”; // This is enough for header

4. Exec command for any interpreted language

Answer: execv(“/bin/bash”, “bash”, script_name);

// pass your script_name

Suggested CGI Scripts:

- Form that takes 2 integers and adds them together and returns the answer to
- Form that takes 2 names and displays them back to the user in the browser
- Basically any form that takes in data, does something with the data and returns the answer back to the browser in a dynamic fashion.

Testing / Hand in:

For this assignment you will need to include the entire directory structure defined above inclusive of a directory of HTML/graphics that represent a "working" website.

What to "hand in"

1. Entire directory structure inclusive of source code in a .zip, .tar or .tar.gz format
2. README text file (no .doc or .docx) describing your code and what works/ not working
3. Directory with html and images as below

NOTE: Normally I will require a design document to describe what you are doing with your code, but in this case you will be using RFC/1945 as your guide, so no design guide this time.

Grading Requirements Rubric

Grading Topic	A Level Work (8-10 points)	B Level Work (6-8 points)	C Level Work (4-6 points)	D Level Work (2-4 points)	Points / 100
Request / Response HTML working	Program completely performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	Program works with no errors. Source code is complete with some documentation	Program works and has some documentation on usage	Program has some functionality, can compile and has a few to no warnings when compiling	15 pts
Graphics	Program completely performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	Program works with no errors. Source code is complete with some documentation	Program works and has some documentation on usage	Program has some functionality, can compile and has a few to no warnings when compiling	15 pts
Concurrency	Program completely performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	Program works with no errors. Source code is complete with some documentation	Program works and has some documentation on usage	Program has some functionality, can compile and has a few to no warnings when compiling	10 pts
CGI Get	Program completely performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	Program works with no errors. Source code is complete with some documentation	Program works and has some documentation on usage	Program has some functionality, can compile and has a few to no warnings when compiling	10 pts
CGI Post	Program completely performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	Program works with no errors. Source code is complete with some documentation	Program works and has some documentation on usage	Program has some functionality, can compile and has a few to no warnings when compiling	10 pts
Logging	Program completely	Program works	Program works	Program has	10 pts

	performs all functionality without errors. Source code is clean and well documented. No warnings during compilation.	with no errors. Source code is complete with some documentation	and has some documentation on usage	some functionality, can compile and has a few to no warnings when compiling	
Readme	Clearly defined program inclusive of algorithms, definitions of input/output, error handling, purpose for program and usage for both user and administrator. All students identified for group effort	Defined program inclusive of usage, algorithms, and purpose for program	Documented usage and how the program works	Lack of information around program purpose and usage. Example of poor documentation is to list functions without any definition of purpose	10 pts
Config file	Config file complete and is 100% in use	Config file working	Config file has functionality	Config file defined with at least one element working	10 pts
RFC 1945 Adherence	Meet 100% of defined sections of RFC1945	Meets > 80% RFC1945	Meets > 60% RFC1945	Meets > 40% RFC1945	10 pts

Suggested development path:

1. Read and understand RFC/1945
2. Understand communication between web servers and web browsers
3. Attain a website to use with your webserver
4. Write iterative pieces of code, don't try to write it all at one time...
    a. Write a piece of code that will read in the config file options
    b. Add a section for creating your socket to listen for browser connections
    c. Hard code some response headers to ensure a browser can talk to the server
    d. Keep adding iterative pieces to your code and KEEP A BACKUP of working versions.

EXAMPLE of how to test your CGI using a form and NOT a webserver

1. Create a file and insert the data below

```
<HTML>
<H1> Hello world</H1>
<FORM ACTION=http://localhost:9000/cgi-bin/bob.cgi METHOD=POST>
login: <INPUT NAME=LOGIN>
password: <INPUT NAME=PASSWORD>
      <INPUT TYPE=SUBMIT NAME=GO>
</FORM>
<H2> Thank you</H2>
</HTML>
```

2. Use the above HTML as "open file" in browser and then open a shell and use the nc command to emulate a web server:

i.e.

```
$ nc -l 9000
```