Machine Problem 1

HTTP Client + Server

Due: Thursday, Feb 13th, 11:59pm

In this assignment, you will implement a simple HTTP client and server. The client should be able to GET correctly from standard web servers, and browsers should be able to GET correctly from your server.

1. Environment Setup

1.1 Ubuntu Virtual Machines

Like in MP0, your submitted code should be able to be compiled (by `make`) and run on Ubuntu 18.04.1 machines. Using virtual machines (VMs) is recommended as this machine problem requires two Ubuntu machines to test the HTTP functions between a HTTP server and a HTTP client. Similarly, the auto-grader will test your programs on two Ubuntu VMs. In each test, the auto-grader will use either your client program or `wget` as the HTTP client, and your server program or `thttpd` as the HTTP server to test through multiple use cases.

HTTP uses TCP – you can use Beej's `client.c` and `server.c` as a base. Note that you don't have to support caching or recursively retrieving embedded objects. Details for the implementation requirements are given in the following sections.

1.2 Your GitLab Repository

You will use git to maintain your code submission. This machine problem uses the same private repository that you use in MP0:

```
https://gitlab.engr.illinois.edu/cs438-sp2020/mp0-mp1/<netid>
```

Use your UIUC login credentials to access this repository. We will grade your programs that are pushed to this repository. Feel free to use this repository to maintain your code during development. **For this machine problem, you must maintain your code in the *mp1* folder.**

2. The Assignment – HTTP Client

2.1 The Basics

Your code should compile to an executable named ***http_client*** with the following usage:

```
./http_client http://hostname[:port]/path/to/file
```

For example:

```
./http_client http://cs438.cs.illinois.edu/
./http_client http://localhost:5678/somedir/somefile.html
```

If *:port* is not specified, default to ***port 80*** – the standard port for HTTP.

Your client should send an **HTTP GET request** based on the ***first argument*** it receives. Your client should then write the **message body** of the received response to a file named ***output***. Your client should also be able to handle redirects as described in detail later.

You **MUST** use `SO_REUSEADDR` and `SO_REUSEPORT` when creating a socket. Failing to do so will lead to a penalty (see Section 5 for the grade breakdown).

2.2 Details on HTTP GET Request

Here's the very simple HTTP GET request generated by `wget`:

```
GET /test.txt HTTP/1.0
User-Agent: Wget/1.15 (linux-gnu)
Accept: */*
Host: localhost:3490
Connection: Keep-Alive
```

- `GET /test.txt` instructs the server to return the file called `test.txt` in the server's top-level web directory.
- `User-Agent` identifies the type of client.
- `Accept` specifies what types of files are desired – the client could say "I only want audio", or "I want text, and I prefer html text", etc. In this case it is saying "anything is fine".
- `Host` is the URL that the client was originally told to get from – exactly what the user typed. This is useful in case a single server has multiple domain names resolving to it (maybe www.cs.illinois.edu and www.math.illinois.edu), and each domain name actually refers to different content. This could be a bare IP address, if that's what the user had typed. The 3490 is the `port` – this server was listening on 3490, so I called "`wget localhost:3490/test.txt`".
- `Connection: Keep-Alive` refers to TCP connection reuse, which will be discussed in class.

Only the first line is essential for a server to know what file to give back, so your HTTP GETs can be just that first line. HTTP specifies that the end of a request should be a marked by a blank line – so be sure to have two newlines at the end. (This demarcation is necessary because TCP present you a stream of bytes, rather than packets.) Note that the newlines are technically supposed to be CRLF - "**\r\n**".

You may use either HTTP 1.0 or 1.1. However, notice that if you are using HTTP/1.0, the Host header is NOT REQUIRED. If you are using HTTP/1.1, the Host header is REQUIRED. But be aware that the Host header may still be necessary for some URLs even in HTTP/1.0.

2.3 Handling Redirections

"The HTTP response status code **301 Moved Permanently** is used for permanent URL redirection, meaning current links or records using the URL that the response is received for should be updated. The new URL should be provided in the Location field included with the response." – Wikipedia page on HTTP 301

In some cases, the server may send back a status code 301, indicating that the content requested has been moved:

```
HTTP/1.1 301 Moved Permanently
Location: <new_url>
```

In such cases the client should attempt to retrieve the document in the <*new_url*> given in the Location field. Here is an example from Wikipedia page:

Client request:

```
GET /index.php HTTP/1.1
```

Server response:

```
HTTP/1.1 301 Moved Permanently
```

```
Host: www.example.org
```

```
Location:
http://www.example.org/index.asp
```

3. The Assignment – HTTP Server

3.1 The Basics

Your code should compile to an executable named ***http_server***. It should take one command line argument (the port number) and start listening on the port specified once started. Usage examples:

```
sudo ./http_server 80
./http_server 8888
```

(The `sudo` is there because using any port <1024 requires root access.)

You **MUST** use `SO_REUSEADDR` and `SO_REUSEPORT` when creating a socket. Failing to do so will lead to a penalty (see Section 5 for the grade breakdown).

The server should handle HTTP GET requests by sending back HTTP responses, as described in detail below.

Your server program should treat all file paths it's asked for as being relative to its current working directory. (Meaning just pass the client's request directly to `fopen`: if the client asks for somedir/somefile.txt, the correct argument to `fopen` is "somedir/somefile.txt").

Warning: running this http_server program essentially makes all files accessible by the http_server process accessible by anyone who can send a request to the host. Discretion is advised (Only run it on a VM)!

3.2 Details on HTTP response

Here's what Google returns for a simple GET of /index.html:

```
HTTP/1.0 200 OK
Date: Wed, 21 May 2014 17:39:46 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=d985d415c1aaf0cc:FF=0:TM=1400693986:LM=1400693986:S=jBoFRLMuiYWpB6sl;
expires=Fri, 20-May-2016 17:39:46 GMT; path=/; domain=.google.com
Set-Cookie: NID=67=UEN1ApahELM_UhkJDWgHbwLmw1thhjwfucoYpC2E-
UpqH6bwR8Rq9YAqY1ptRu3qCeIjkHLBwY867JmRn4fzFQzJpgId1TLzXhBhLjAKCpGx0DQpVSDFjAPByCQo37
K4; expires=Thu, 20-Nov-2014 17:39:46 GMT; path=/; domain=.google.com; HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/answer.py?
hl=en&answer=151657 for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic

<!doctype html><html itemscope=""much more of the document follows...
```

Another example:

```
HTTP/1.1 200 OK
Date: Sun, 10 Oct 2010 23:26:07 GMT
Server: Apache/2.2.8 (Ubuntu) mod_ssl/2.2.8 OpenSSL/0.9.8g
Last-Modified: Sun, 26 Sep 2010 22:04:35 GMT
ETag: "45b6-834-49130cc1182c0"
Accept-Ranges: bytes
Content-Length: 13
```

```
Connection: close
Content-Type: text/html

Hello world!
```

Your server's headers can be much simpler (but still correct and complete): just the status line (the first line of the header, contains the status code).

- When correctly returning the requested document, use "**HTTP/1.0 200 OK**".
- When the client requests a non-existent file, use "**HTTP/1.0 404 Not Found**". Note that you can still have document text on a 404, allowing for nicely formatted/more informative message such as "whoops, file not found!"
- For any other errors, use "**HTTP/1.0 400 Bad Request**".

Per the HTTP standard, again an empty line (two CRLF, "\r\n\r\n") marks the end of the header, and the start of message body.

To summarize, your whole response should be: **header + empty line + message body (the file requested if available)**

Reading the Wikipedia page HTTP message body may be helpful.

3.3 Handling Concurrent Connections

Your server must support concurrent connections: if one client is downloading a 10MB object, another client that comes looking for a 10KB object shouldn't have to wait for the first to finish.

Hint: you can do so by handling each connection in a new thread.

4. What to Submit

You must submit (i.e., commit and push) your code to the private GitLab repository provided to you in Section 1.2. We will only grade your programs that are pushed to the `master` branch in that repository. In your repository, it must contain (at least) the following files:

- `/mp1/http_client.c` # your client main source code
- `/mp1/http_server.c` # your server main source code
- `/mp1/Makefile` # the makefile

You can include other .c and .h files if needed. You must create the makefile so that execution of the command

```
make
```

compiles and generates the executable files named `http_client` and `http_server`, in the same *mp1* folder. This executable `http_client` and `http_server` should be able to run with arguments as specified in Section 2.1 and 3.1, respectively.

Finally, running

```
make clean
```

should delete all executable files and any temporary files that the makefile or your program creates.

You may commit and push your code as many times as you like before the deadline. It's in fact a good practice to commit your code whenever there is a change that is worth noted. We will use the last commit you made for the repository when we grade your MP.

Your program cannot be graded if it has not been pushed to the repository. Failure to commit and push your code on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed and pushed to your repository by the due date.

5. Grading

- 10%: you submitted your assignment correctly
- 10%: it compiles correctly on the auto-grader
- 5%: your client can retrieve `http://cs438.cs.illinois.edu`.
- 5%: **Use the (SO_REUSEADDR | SO_REUSEPORT) socket option**
- 15%: `wget` can retrieve files from your HTTP server
- 10%: your server can handle invalid client request and request for non-existing files properly
- 20%: your client can retrieve files from your HTTP server
- 15%: multiple clients can download files from your server simultaneously (concurrency)
- 10%: client can handle redirects correctly
- Late penalty: 2% of total possible score per hour

6. Important Notes

1) You must use C or C++.
2) Your code must be runnable in 64-bit Ubuntu 18.04.1 LTS VMs. This is the environment in which the auto-grader runs. Your program must fulfill the given assignment requirements to get said scores.
3) If you need to use a library for data structures, you MUST get the approval of the course staff. Additionally, you MUST acknowledge the source in a README in *mp1* folder. However, algorithms MUST be your own.
4) The auto-grader will test every student's current version every night, starting at 6PM. The TAs will try their best to make sure this is the case, but bugs happen. We will make an announcement of its starting date.
   - Do a git pull the next morning to see your score in *mp1/results.txt*.
   - The score you are given before the deadline is for your reference and has no effect on your final grade.
   - The late penalty is applied to the last git commit you make for anything in the folder *mp1*. If you want to make any change in *mp1* that does not change your code, please wait until 48 hours after the deadline to make it.
5) Do not make your private repository public. You will be held partially responsible for any resultant plagiarism.
6) You must work alone (for MP0 and MP1). Your code must be your own. You can discuss very general concepts with others, but if you find yourself looking at a screen/whiteboard full of pseudo code (let alone real code), you are going too far.
   - Refer to the class slides and official student handbook for academic integrity policy. In summary, the standard for guilt is "more probable than nor probable", and penalties range from warning to recommending suspension/expulsion, based entirely on the instructor's impression of the situation.
   - The Grainger College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.