CS 1652: Project 1

Overview
In this project, your group will build a simple web client and a succession of servers to which it can connect. The goal is to slowly introduce you to Unix and socket programming and prepare you for project 2 (where you'll be implementing a TCP layer that your web client/servers can run on top of). There is also an extra credit server you can build that has a basic structure similar to real web servers that need to handle many connections.

HTTP and HTML
The combination of HTTP (Hypertext Transport Protocol) and HTML (Hypertext Markup Language) forms the basis for the World Wide Web. HTTP provides a standard way for a client to request typed content from a server, and for a server to return such data to the client. "Typed content" simply means a bunch of bytes annotated with information (a MIME type) that tells us how we should interpret them. For example, the MIME type text/plain tells us that the bytes are unadorned ASCII text. You will implement a greatly simplified version of HTTP 1.0.

HTML (type text/html) content provides a standard way to encode structured text that can contain pointers to other typed content. A web browser parses an HTML page, fetches all the content it refers to, and then renders the page and the additional embedded content appropriately.

HTTP Example
In this project, you will implement a subset of HTTP 1.0. HTTP was originally a very simple, but very inefficient protocol. As a result of fixing its efficiency problems, modern HTTP is considerably more complicated. Its specification in RFC 2616 is over a hundred pages long! Fortunately, for the purposes of this project, we can ignore most of the specification and only implement a small subset.

The HTTP protocol works on top of TCP, a reliable stream-oriented transport protocol, and is based on human-readable messages. Because of this, we can use the telnet program to investigate how HTTP works. We'll use telnet in the role of the client and www.washington.edu in the role of the server. (why University of Washington? well, there just aren't that many unencrypted HTTP websites around anymore...) The typed content we'll transfer is University of Washington's home page. This is essentially the same as fetching the home page using your favorite web browser.

The following shows what this looks like for the URL http://www.washington.edu/. The text in bold is what you would type, and the rest is an example output from when I ran this on January 13.

```
$ telnet www.washington.edu 80
Trying 128.95.155.134...
Connected to www.washington.edu.
Escape character is '^]'.
GET / HTTP/1.0
(blank line)
HTTP/1.1 200 OK
Date: Thu, 13 Jan 2022 16:32:39 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/1.0.1e-fips
PHP/7.2.23 mod_pubcookie/3.3.4a mod_uwa/3.2.1
Last-Modified: Thu, 13 Jan 2022 01:07:09 GMT
ETag: "180098-100be-5d56c4e378940"
Accept-Ranges: bytes
Content-Length: 65726
Vary: Accept-Encoding,User-Agent
Connection: close
Content-Type: text/html
…

Connection closed by foreign host.
$
```

The first thing to notice is that we are opening a TCP connection to port 80. Telnet does a DNS lookup on the host www.washington.edu and finds that it is at IP address 128.95.155.134 (note you may get a different IP address when you try this – can you think of why that might happen?). It then opens the connection and lets us type.

"`GET / HTTP/1.0`" is the most basic form of an HTTP 1.0 request, and the form that you will implement. It says "please give me the file (`GET`) that corresponds to the top-level page (`/`) using the 1.0 version of the HTTP protocol (`HTTP/1.0`)." The blank line demarcates the end of the request. This is necessary because a more complex request may place further conditions (on additional lines) on what the client is willing to accept and how it is willing to accept it.

The response always begins with a line that states the version of the protocol that the server speaks ("`HTTP/1.1`" in this case), a return/status/error code ("`200`"), and a textual description of what that error code means ("`OK`"). Next, the server provides some information about the content it is about the send as well as information about itself and what kinds of services it can provide. The most critical line is "`Content-Type: text/html`", which tells us how to interpret the content we receive. A blank line demarcates the end of the response header and the beginning of the actual content. After the content has been sent, the server closes the connection.

Part 0: Install and configure your Linux environment

We will be evaluating your work using an Ubuntu 18 LTS Virtual Machine, so you need to make sure your submission works in that environment.

We are providing a preconfigured VM. The VM has a default account already set up:
Username: cs1652
Password: cs1652

This account has sudo access and full administrative capabilities.

The network configuration for the VM should already be set up, but you should confirm that it is configured with NAT-based network configuration.

Check VM NAT network configuration in VirtualBox:
Devices->Network: make sure "Connect Network Adapter" is checked
Device->Network->Network Settings…: Make sure it has "Attached To: NAT"

Part 1: HTTP Client

Write a client program that supports the following command line and semantics.

```
http_client <server_name> <server_port> <server_path>
```

When run, `http_client` should open a connection to port `server_port` on the machine `server_name`, and then send an HTTP request for the content at `server_path`. It should then read the HTTP response the server provides. If the response is that `server_path` is valid and includes the data, `http_client` should write the data out to standard out and exit with a return code of zero. You can then view this output using a web browser (e.g. Firefox, Chrome, Edge). If there is an error, `http_client` should write the response to standard error and exit with a return code of -1. For example,

```
    http_client www.washington.edu 80 index.html
```

should print the University of Washington home page to standard out and return zero, while

```
    http_client www.washington.edu 80 junk.html
```

should print the response to standard error and return -1.

Part 2: Connection-at-a-time HTTP Server

Write an HTTP server that handles one connection at a time and that serves files in the current directory. This is the simplest kind of server. The command-line interface will be:

```
  http_server1 port
```

You will then be able to use http_client, telnet, or any web browser, to fetch files from your server. For example, if you run:

```
  http_client host port http_server1.c
```

you should receive the contents of your source file.

Note that you will not be able to run your server on port 80. Ports less than 1500 are reserved, and you need special permissions to bind to them.

Your server should have the following structure:

1. Create a TCP socket to listen for new connections on (What packet family and type should you use?)
2. Bind that socket to the port provided on the command line. We'll call this socket the *accept socket*.
3. Listen on the accept socket (What will happen if you use a small backlog versus a larger backlog? What if you set the backlog to zero?)
4. Do the following repeatedly:
    a. Accept a new connection on the accept socket (When does accept return? Is your process consuming cycles while it is in accept?) Accept will return a new socket for the connection. We'll call this new socket the connection socket. (What is the 5-tuple describing the connection?)
    b. Read the HTTP request from the connection socket and parse it. (How do you know how many bytes to read?)
    c. Check to see if the file requested exists.
    d. If the file exists, construct the appropriate HTTP response (What's the right number?), write it to the connection socket, and then open the file and write its contents to the connection socket.
    e. If the file doesn't exist, construct a HTTP error response and write it back to the connection socket
    f. Close the connection socket.

Part 3: Simple Select-based Multiple-connection-at-a-time Server

The server you wrote for part 2 can handle only one connection at a time. Try the following. Open a telnet connection to your `http_server1` and type nothing. Now make a request to your server using your `http_client` program. What happens? If the connection request is refused, try increasing the backlog you specified for listen in http_server1 and then try again. After `http_server1` accepts a connection, it blocks (stalls) while reading the request and so is unable to accept another connection. Connection requests that arrive during this time are either queued, if the listen queue (whose size you specified using listen) is not full, or refused, if it is.

Consider what happens if the current connection is very slow. Your server is spending most of its time idle waiting for this slow connection while other connection requests are being queued or refused. Note that reading the request is only one place where `http_server1` can block: it can also block on waiting for a new connection, on reading data from a file, and on writing that data to the socket.

Write an HTTP server, `http_server2`, that avoids just two of these situations: waiting for a connection to be established, and waiting on the read after a connection has been established.

You can make the following assumptions:

- If you can read one byte from the socket without blocking, you can read the whole request without blocking.
- Reads on the file will never block
- Writes will never block

It is important to note that if you have no open connections and there are no pending connections, then you **should** block (i.e. your program should not consume processing cycles or "busy-wait" in this case).

To support multiple connections at a time in `http_server2`, you will need to do two things:

- Explicitly maintain the state of each open connection
- Block on multiple sockets, file descriptors, events, etc.

It is up to you to decide what the contents of the state of a connection are and how you will maintain them. However, Unix, as well as most other operating systems, provides a mechanism for waiting on multiple events. The Unix mechanism is the `select` system call. `select` allows us to wait for one or more file descriptors (a socket is a kind of file descriptor) to become available for reading (so that at least one byte can be read without blocking), writing (so that at least one byte can be written without blocking), or to have an exceptional condition happen (so that the error can be handled). In addition, `select` can also wait for a certain amount of time to pass.

Your server should have the following structure:

1. Create a TCP socket to listen for new connections on
2. Bind that socket to the port provided on the command line.
3. Listen on that socket, which we will call the *accept socket*.
4. Initialize the list of open connections to empty
5. Do the following repeatedly:
    a. Make a list of the sockets we are waiting to read from the list of open connections. We shall call this the *read list*.
    b. Add the accept socket to the read list. Having a new connection arrive on this socket makes it available for reading, it's just that we use a strange kind of read, the accept call, to do the read.
    c. Call `select` with the read list. Your program will now block until one of the sockets on the read list is ready to be read.
    d. For each socket on the read list that `select` has marked readable, do the following:

        i. If it is the accept socket, accept the new connection and add it to the list of open connections with the appropriate state
        ii. If it some other socket, perform steps 4.b through 4.f from the description of `http_server1`. After closing the socket, delete it from the list of open connections.

Test your server using telnet and `http_client` as described above.

Extra Credit: Complex Select-based Multiple-connection-at-a-time Server

`http_server2` can handle multiple connections at a time, but there remain a number of places where it can block. These are implicit in the assumptions we have made. In general, almost any system call can block. In particular, if `select` tells us that a file descriptor is readable, it only means that at least one byte can be read. Reading any subsequent byte may block. The same holds true for writes.

To avoid unnecessary blocking, then, the program must check each system call that may block, and certainly read and write, before it executes the system call. Does this mean that we have to call `select` before we read or write each byte? Not necessarily. We can instead use non-blocking I/O. If we set a file descriptor to operate in non-blocking mode, then system calls on that file descriptor will fail with an `EAGAIN` error instead of blocking. `EAGAIN` means "I can't do that right now because doing so would block you and you asked me never to let that happen." To read more about non-blocking I/O, see the man page for `fcntl`. `fcntl(fd,F_SETFL,O_NONBLOCK)` is one way to set a file descriptor to non-blocking I/O. To learn how to retrieve error codes from system calls, check out the man page for `errno`.

For extra credit, you can build an HTTP server, `http_server3`, that uses `select` and non-blocking I/O to provide availability even in the face of blocking on any of the reads, writes, and accepts, as well as dealing with partial reads and writes. The overall structure of the code is as follows:

1. Create a TCP socket to listen for new connections on
2. Bind that socket to the port provided on the command line.
3. Listen on that socket, which we will call the *accept socket*.
4. Initialize the list of open connections to empty. You should associate with each connection its state, the file descriptor for the file it is reading, and any other information you need.
5. Do the following repeatedly:
    a. Make a list of file descriptors we are waiting to read from the list of open connections. This will include both sockets and file descriptors for files you are in the process of reading. We shall call this the *read list*.
    b. Add the accept socket to the read list.
    c. Make a list of sockets we are waiting to write from the list of open connections. We shall call this the *write list*.

d. Call `select` with the read list and the write list. Your program will now block until one of the sockets on the read list is ready to be read or written.
e. For each socket on the read list that `select` has marked readable do the following:
    i. If it is the accept socket, accept the new connection, set its socket to be non-blocking, and add it to the list of open connections with the appropriate state
    ii. If it's some other socket, look up its connection in the list of open connections, figure out how much you have left to read, and then read until you get an `EAGAIN` or you've read the whole request.
        1. If you get the `EAGAIN`, update the connection's state accordingly.
        2. If you've read the whole request, open the file, set its file descriptor to non-blocking, add it to the connection state, and update the state to note that you're in the process of reading the file.
    iii. If it's some other file descriptor, look up its connection in the list of open connections, figure out how much you have left to read, and then read until you get an `EAGAIN` or you've read the whole file.
        1. If you get the `EAGAIN`, update the connection state to reflect you much you have read.
        2. If you've read the whole file, close the file, update the connection state to reflect that you are ready to start writing the contents to the socket.
f. For each socket on the write list that `select` has marked writeable do the following:
    i. Look up its connection in the list of open connections, figure out how much you have left to write, and then write until you get an `EAGAIN` or you've read the whole request.
        1. If you get the `EAGAIN`, update the connection state to reflect how much you've written.
        2. If you've written the whole file, close the socket and remove the connection from the list of open connections.

Logistics

- You should work in teams of exactly two students
- You are encouraged to use version control to collaborate with your partner on the same codebase. Using Github for this purpose is encouraged, but your repositories must be PRIVATE and shared only with your partner
- Your code must be written in C and must compile with the default versions of gcc shipped with Ubuntu LTS 18. We will expect that running "make" will generate the executables `http_client`, `http_server1`, `http_server2`, and (if you decide to do the extra credit), `http_server3` according to our Makefile.

- You will submit http_client.c, http_server1.c, http_server2.c and (if you do it) http_server3.c. If you modify our makefile, you should include that too.
- Assignments will be submitted in Canvas

Some Useful Resources

- Beej's guide to network programming: http://beej.us/guide/bgnet/
- Section 2.2 of your textbook provides more information about HTTP and shows an example of simple HTTP interactions
- Lecture 4 slides will provide examples of TCP clients and servers using C
- You can (and should) play with www.washington.edu or some other web server using telnet to port 80.
- You can use Wireshark (already installed in the course VM image) to capture and view network traffic
- RFC 1945 (HTTP 1.0) available via http://www.ietf.org
- Later RFCs may be of interest, although they are more complex than what you need to do for this assignment (e.g. HTTP 1.1 RFC 2616)