作业6: Server

Due: 春节后

Purpose

For the last assignment, you will implement a simple HTTP web server - an application that almost every company in the world runs. The web server should be able to respond to any page request, given that the requested page is stored locally (i.e., on the same system in which the server is running). You can use any web browser (such as Firefox) as a client for your HTTP web server.

In this assignment, you will create a program that:

1. accepts connections from web browsers
2. reads in the packets that the browsers send
3. prepares and sends back a response for the web browser from a local file

Since the web browser can only understand HTTP packets, you need to understand the basics of how HTTP works. When a web browser requests a page, it sends an HTTP request to the web server. The HTTP request has the following format:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:11.0) Gecko/20100101
Firefox/11.0
Accept: */*
Connection: Keep-Alive
```

This HTTP packet notifies the web server of what file the web browser has requested. The web server will then serve the file to the web browser, which it will display on the requestor's screen.

You should only modify a single file to complete this assignment: `server.c`. However, we have provided you with a dictionary and queue library that you have used in previous assignments.

You will always supply the port number when running your program. Since ports are shared globally on each system, it's important to choose a port number that someone else probably won't be using. Therefore, port numbers like 1234, 1111, or 777 are generally a bad idea since multiple people may choose to use the same port number. Your program should be ran by using a command like the following (with a unique port number, of course):

```
./server 1979
```

Instead of interacting with the command line, you will use a web brwoser to connect to your assignment. Instructions to do this is at the bottom of this page.

Tasks

Task 1:

Create a socket to listen for incoming TCP connections based on the port specified by the command line. Make certain to check for an error in the call of `bind()`, as this call will fail if someone is using the same port as you. You should use a backlog of 10.

Keep in mind that when you Ctrl+C your program, you'll find that you won't free your port for up to a minute. This means that you may need to change the port number if you're restarting your server right after terminating it.

System calls: `socket()`, `bind()`, `listen()`

Task 2:

Continuously accept incoming connections, launching a new thread for each connection.

System calls: `accept()`, `pthread_create()`

Task 3a:

In the worker thread for each of the connections, read the entire HTTP request header. An HTTP request header ends with the four characters `\r\n\r\n`. For example, the entire sample request header from a Firefox web browser is:

```
GET / HTTP/1.1\r\n
Host: google.com\r\n
Connection: close\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:11.0) Gecko/20100101
Firefox/11.0\r\n
Accept-Encoding: gzip\r\n
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7\r\n
Cache-Control: no-cache\r\n
Accept-Language: de,en;q=0.7,en-us;q=0.3\r\n
\r\n
```

You should **NOT** assume that the entire header will be read in one call to `recv()`. You need to keep `recv()`'ing on the buffer until you reach `\r\n\r\n`. For small requests, one call will be enough; for large requests, it will take multiple calls. To perform this step, you may want to implement your own parsing algorithm or use the provided `libhttp` library detailed here.

System call: `recv()`

Task 3b:

Parse the HTTP request header line-by-line. The first line of the HTTP request header, often called the "request line", should be passed into the helper function we provide to you, `process_http_header_request()`. All other lines should be stored in an accessible way (maybe the libhttp or the dictionary will be helpful?).

Task 3c:

Prepare the HTTP response. This may be one of three things:

- A successful page request (status code: 200), where the file that was requested is available on your machine and is sent back to the web browser.
- A file not found error (status code: 404), where the requested file was not found.
- A request that your server can't handle (status code: 501).

If `process_http_header_request()` returns NULL, you should return a 501 response. If `process_http_header_request()` returns a non-NULL pointer, it is the name of the file requested by the user. **Note that you should not open that file directly.**

If the file is exactly `/`, it should be processed as `/index.html`. When accessing any file that is requested by the user, you should load the file inside the `"web"` directory of your current folder. That is, a request to `/myfile.html` should result in an `fopen()` call `web/myfile.html`.

If the file that is requested does not exist on disk, you should return a 404 response. For 404 and 501 responses, we have prepared the entire response body text for you in the variables `HTTP_404_CONTENT` and `HTTP_501_CONTENT`. If the file that is requested does exist on disk, the response body text will be the entire contents of the file.

Task 3d:

Determine the content-type of the response. For 404 and 501 requests, the content type will always be `text/html`. For 200 requests, you need to examine the file name. Using the following list as a reference:

- If the file ends with `.html`, the content type is `text/html`
- If the file ends with `.css`, the content type is `text/css`
- If the file ends with `.jpg`, the content type is `image/jpeg`
- If the file ends with `.png`, the content type is `image/png`
- Otherwise, the content type is `text/plain`

Task 3e:

Send an HTTP response back to the user. Your response must include a "response line", the headers `Content-Type`, `Content-Length`, and `Connection` header, and the content itself.

The response line must be of the format `HTTP/1.1 200 OK`. This can be done with:

```
sprintf(..., "HTTP/1.1 %d %s", response_code, response_code_string);
```

The `response_code` variable is set by you in Task 3c. The `response_code_string` should be either the global variable `HTTP_200_STRING`, `HTTP_404_STRING`, or `HTTP_501_STRING` depending on your response code (you will notice we provide these for you at the top of `server.c`).

The header `Content-Type` must match what is set in Task 3d. The `Content-Length` must be the length of the content you will be sending (either the string we provide for you in 404 and 501 cases, or the length of the file, e.g. 200 B). Finally, if the request packet contains a `Connection` (case sensitive) header and if the `Connection` header matches `Keep-Alive` (case insensitive, use `strcasecmp()` instead of `strcmp()`), you must contain a `Connection: Keep-Alive` line in your response header. If the request packet does not contain `Connection: Keep-Alive` (either because `Connection` is not in the request or because the value is not `Keep-Alive`), you must contain a `Connection: close` line in your response header.

Just like the request, every line in the HTTP header must be separated by `\r\n` and the last line must contain an empty line. After the empty line to end the header, the content of the packet should be added.

Therefore, if the file that is requested contains the string "Hello World!", is an HTML file, and you're running on a "Keep-Alive" connection, your response needs to be:

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 12\r\n
Connection: Keep-Alive\r\n
\r\n
Hello World!
```

System call: `send()`

Task 3f:

Finally, if the connection is to be kept alive (see Task 3e), you should begin to `recv()` again by repeating 3a-3f. If the connection is closed, you should close the socket and exit the thread.

Task 4:

Catch the SIGINT signal (`Ctrl+C` on the keyboard). Upon catching a SIGINT signal, you should `close()` any open sockets, free all the memory in use, and exit from the server. (*If your SIGINT handler is broken and you can't exit your program, you can use Ctrl+\ to send a SIGQUIT. A SIGQUIT will terminate your program.*)

System call: `signal()` or `sigaction()`, `close()`

Compiling And Running

Running the web server:

To run your program, run the following commands:

```
%> make
%> ./server <port#> ...where <port#> is a port number.
```

Note:
1. When choosing your <port#>, choose a number above 1023 and below 60000. Making your number random (eg: not 1234, 2000, etc) will help avoid choosing a port used by another user.
2. Since ports are shared globally, your `bind()` call will fail if someone else is already using your port. If this happens, wait a few seconds and then try again. If bind still fails, choose a new port.

Running the client:

There are three ways to run the client (i.e., the web browser).

- **Option 1:** If you're VPN'd into develop LAN or if you're on a develop LAN computer, you will now be able to open up Firefox (or any web browser) and go to the following address:
  `http://someurl:<port#>/`
  ... where <port#> should be replaced by the port number you used to run your web server program and linux# should be replaced by the linux machine you're currently on (`linux-v1`, `linux-v2`, etc...)

- **Option 2:** When you run the web server and the browser on the same machine, you can simply use:
  `http://localhost:<port#>/`
  If you see a webpage, your program successfully served an HTTP request! We will not be grading anything on the command line output, so feel free to use stdout and stderr for any debugging or status messages you'd like.

- **Option 3:** If nothing seems to be working, you may find the Linux command-line 'web browser' useful. This 'web browser', wget, allows you to get useful command line output right there in a terminal (and avoids any VPN issues). To run wget, use:

```
wget -d http://someurl:<port#>/
```

We have populated your web/ folder with few html files. Use these files as well as any other test file you'd like to use to verify your webserver.