COMS W3157 Advanced Programming, Lab #7 (150 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

Please refer to the lab retrieval and submission instruction. The requirements regarding README.txt and Makefiles remain the same as the previous labs unless specified otherwise.

For this lab, do not make part1 and part2 directories. Just put your part2 files directly in lab7 directory. There are no files to submit for part1.

Checking memory errors with valgrind

Do not include valgrind output in README.txt. You should keep using valgrind to check your program for memory error (and the TAs will do the same when grading), but you don't have to include the output in README.txt anymore.

Part 1: Build a web page on your account (20 points)

Follow this instruction from CRF to setup a web page on your CLAC account:

http://www.cs.columbia.edu/~crf/newweb/index.html

Your page at "http://clac.cs.columbia.edu/~YOUR_UNI/cs3157/tng/" will show the same web page that http://clac.cs.columbia.edu/~jae/cs3157/tng/ shows. (You are allowed to tweak it though, as described later.)

You can copy the files that make up the web page (index.html, crew.jpg and ship.jpg) from Jae's account. Your job is to construct an appropriate directory structure under ~/html to place the 3 files in order for the URL to work. You may also need to modify the permissions on the directories and files. Directories under ~/html should have 711 and files should have 644.

There is nothing to submit for this part, other than your statement in your README.txt that you have setup your web page. The TAs will test this part by pointing browser to "http://clac.cs.columbia.edu/~YOUR_UNI/cs3157/tng/".

If you feel creative, or don't care much about Star Trek, you are welcome to replace the content of the web page. But please don't change the file names, directory hierarchy, and the web page structure. That is, there must be index.html, crew.jpg, ship.jpg (with the same directory hierarchy as mine), and index.html must have two <img> tags, images/ship.jpg and images/crew.jpg. Those images should be immediately visible without user interaction, JavaScript, or CSS. Apart from those guidelines, feel free to get creative! Post a link to the listserv if you make something cool or funny.

Part 2: Write a web server (130 points)

(a) Serving static contents

In this part, you are writing a web server, named http-server, that can serve static HTML and image files. The http-server takes the following parameters:

```
./http-server <server_port> <web_root> <mdb-lookup-host> <mdb-lookup-port>
```

server_port is the port that you're listening for HTTP requests and web_root is the top level directory for your HTML files. The last two parameters are for part 2(b). For example:

```
./http-server 8888 ~/html localhost 9999
```

should serve the Star Trek page to the following request:

```
http://the.machine.your.server.is.running.on:8888/cs3157/tng/index.html
```

Writing a web server is not a trivial task. Although you're all super-coders at this point, since you only have a few days, we need to cut some corners. Here is the list of what is expected and what is not expected from your web server:

- It will only support the GET method. If a browser sends other methods (POST, HEAD, PUT, for example), the server responds with the status code 501. Here is a possible response:

  ```
  HTTP/1.0 501 Not Implemented

  <html><body><h1>501 Not Implemented</h1></body></html>
  ```

  Note that the server adds a little HTML body for the status code and the message. Without this, the browser will display a blank page. This should be done for all status codes except 200.

- Our server will be strictly HTTP 1.0 server. That is, all responses will say "HTTP/1.0" and the server will close the socket connection with the client browser after each response.

  The server will accept GET requests that are either HTTP/1.0 or HTTP/1.1 (most browsers these days sends HTTP/1.1 requests). But it will always respond with HTTP/1.0. The server should reject any other protocol and/or version, responding with 501 status code.

- The server should also check that the request URI (the part that comes after GET) starts with "/". If not, it should respond with "400 Bad Request".

- In addition, the server should make sure that the request URI does not contain "/../" and it does not end with "/.." because allowing ".." in the request URI is a big security risk--the client will be able to fetch a file outside the web root.

- You may find the following sequence of code handy for parsing the request line:

```
char *token_separators = "\t \r\n"; // tab, space, new line
char *method = strtok(requestLine, token_separators);
char *requestURI = strtok(NULL, token_separators);
char *httpVersion = strtok(NULL, token_separators);
```

  See man strtok for explanation.

- The server must log each request to stdout like this:

```
128.59.22.109 "GET /cs3157/tng/images/ship.jpg HTTP/1.1" 200 OK
```

  It should show the client IP address, the entire request line, and the status code and reason phrase that it just sent to the browser.

- The server should be robust against client failure. For example, if the client browser crashes in the middle of sending a request, the server should simply close the socket connection and move on to the next client request.

  This means that, in your code, you can't just die() on every failure. You need to think about which errors are recoverable (or ignorable) and which are not.

  You can simulate the client failure by using netcat: just Ctrl-C in the middle of typing a request.

- If the request URI ends with '/', the server should treat it as if there were "index.html" appended to it. For example, given

```
http://localhost:8888/cs3157/tng/
```

  the server will act as if it had been given

```
http://localhost:8888/cs3157/tng/index.html
```

- If the request URI is a directory, but doesn't have '/' at the end -- "http://localhost:8888/cs3157/tng", for example -- you should append "/index.html" to it.

  Use stat() function to determine if a path is a directory or a file.

- The server sends "404 Not Found" if it's unable to open the requested file.

- For reading the file, you can use fread() or read(). You should read the file in chunks and send it to the client as you read each chunk. The chunk size should be 4096 bytes (that's 4K, the optimal buffer size for disk I/O for many types of OS/hardware).

  Do not read the file one character at a time using fgetc() or getc(). Do not read the file one line at a time using fgets()--this may not work for image files.

(b) Serving dynamic contents

In this part, you'll add the mdb-lookup functionality to your web server. Your web server will work in conjunction with mdb-lookup-server from lab6, part1.

You can test it out using my versions, available in /home/jae/cs3157-pub/bin:

1) Start mdb-lookup-server with the class database file:

```
./mdb-lookup-server mdb-cs3157 9999
```

2) Open up another terminal window into the same machine and run http-server, which will connect to the mdb-lookup-server you started earlier:

```
./http-server 8888 ~/html localhost 9999
```

3) Point your browser to:

```
http://the.host.name:8888/mdb-lookup
```

4) If you type "hello" into the text box and submit, you will see a few messages containing hello nicely formatted in a HTML table. If you look at the location bar of the browser, you'll see that it went to this URL:

```
http://the.host.name:8888/mdb-lookup?key=hello
```

Here are the requirements and hints for implementing this functionality in your server:

- You must make a TCP connection to the mdb-lookup-server when the web server starts up, and keep using the SAME socket connection. There is only a single persistent connection to the mdb-lookup-server during the execution of the web server.

  Using that connection, you can send the search string and read back the result rows. You can detect the end of result by reading a blank line.

  Also, make sure to append "\n" to the search string before you send it to mdb-lookup-server. (Think about why.)

- When the request URI is simply "/mdb-lookup", you send the submit form. You can hard-code the form in your program like this:

```
const char *form =
    "<h1>mdb-lookup</h1>\n"
    "<p>\n"
    "<form method=GET action=/mdb-lookup>\n"
    "lookup: <input type=text name=key>\n"
    "<input type=submit>\n"
    "</form>\n"
    "<p>\n";
```

(Note that C language automatically concatenates adjacent string literals.)

- If the request URI starts with "/mdb-lookup?key=", you send the lookup result table in addition to the form.

  Since you don’t know how big the lookup result will be, it would be easiest to send the resulting HTML table piece-by-piece as you read each line from the socket connection to the mdb-lookup-server. It is easy to format a HTML table on the fly (see the HTML source of the lookup result from your browser to learn how to construct an HTML table).

- Make sure the logging you implemented in part 2(a) still works for the mdb-lookup requests.

--

Good luck!