COMS W3157 Advanced Programming, Lab #6

Please refer to our instructions for lab retrieval and submission. You must include the required information in your README.txt. This lab consists of multiple parts, so your code for each part must be in a subdirectory (named "part1", "part2", etc.). Like before, you are required to have 5 Git commits with meaningful commit messages, a Makefile for each part where you write code, and Valgrind output in your README.

Part 0: personal web page

Create a personal web page on CLAC that you can access with a browser at:

    http://clac.cs.columbia.edu/~<username>/cs3157/

Note that your CLAC username should be your UNI.

Here are some hints and requirements:

- The HTTP server we run on CLAC is the Apache HTTP server. It serves files out of the ~/html directory in each user's home directory.

- You must make sure you set the appropriate permissions under the html directory; otherwise, Apache will be unable to access your files. Directories should have 711 and files should have 644. You must also set your home directory to have 711 permissions.

- You can copy John's web page files (index.html, photo.jpg, and pokemon.jpg) into your ~/html directory to get started. You may leave the contents of the web page unchanged if you do not feel creative.

- If you DO feel creative, make sure to follow these guidelines:

    - Your web page must be located at:

          http://clac.cs.columbia.edu/~<username>/cs3157/

      The page should be organized according to the same directory structure as John's web page.

    - Your web page must contain AT LEAST one image, located at:

          http://clac.cs.columbia.edu/~<username>/cs3157/images/photo.jpg

      The image should be displayed on your page using an <img> tag, and should be visible without user interaction/JavaScript/CSS/etc.

    - Apart from those guidelines, feel free to get creative! Post a link to the listserv if you make something cool or funny.

There is nothing to submit for this part, other than your statement in your README.txt that you have set up your web page. We will test this part by pointing a browser to http://clac.cs.columbia.edu/~<username>/cs3157/.

Part 1: static web server

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

```
./http-server <server-port> <web-root>
```

http-server listens on <server-port> for HTTP requests, and serves files out of <web-root>. For example:

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

should serve your personal page at the following URL:

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

As usual, here are some hints and requirements:

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

  ```
  HTTP/1.0 501 Not Implemented

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

  Note that http-server adds a little HTML body for the status code and the message. Without this HTML body, the browser will display a blank page. You should do this for all status except 200.

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

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

- http-server should also check that the request URI starts with "/". If not, it should respond with "400 Bad Request".

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

  Note that some browsers will "fix" broken URLs like these if you type them into the address bar. You should test this with netcat.

- You may find the following 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.

- http-server must log each request to stderr like this:

  ```
  128.59.22.109 "GET /cs3157/images/photo.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.

To obtain the client IPv4 address as a string, you should use inet_ntoa().

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

  This means that, in your code, you cannot just die() on every failure. Think about which errors are recoverable/ignorable and which are not.

  Also recall that, by default, the operating system will terminate your process with SIGPIPE if the process tries to write to a disconnected socket. Before handling any clients, you should use sigaction() to configure http-server to ignore potential SIGPIPEs.

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

- http-server should send "404 Not Found" if it is unable to open the requested file.

- For reading the file, you should 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 (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(); this is extremely inefficient.

  Do not read the file one line at a time using fgets(); this may not work for binary files such as images.

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

  ```
  http://clac.cs.columbia.edu:8888/cs3157/
  ```

  http-server should act as if it had been given:

  ```
  http://clac.cs.columbia.edu:8888/cs3157/index.html
  ```

- Use stat() to determine if the requested file path is a file or a directory, like this:

  ```
  struct stat st;
  if (stat(file_path, &st) == 0 && S_ISDIR(st.st_mode)) {
      // file_path is a directory
  }
  ```

  If the request is for a directory but the request URI does not end with "/", you should send a "301 Moved Permanently" response, along with a "Location" header with the trailing "/" attached to the URI.

  Here is how the response from our reference implementation looks:

  ```
  $ echo -e -n "GET /cs3157 HTTP/1.0\r\n\r\n" | \
  ```

```
    nc clac.cs.columbia.edu 8888
HTTP/1.0 301 Moved Permanently
Location: /cs3157/

<html><body>
<h1>301 Moved Permanently</h1>
<p>The document has moved <a href="/cs3157/">here</a>.</p>
</body></html>
```

- We generally forbid the use of memset() in this class because it can hide memory bugs, but getaddrinfo() and sigaction() both expect zero-initialized structs. So, for this lab, you are allowed to use memset() to zero-initialize the struct addrinfo you pass to getaddrinfo(), and the struct sigaction you pass to sigaction().

Part 2: multi-process static web server

Your http-server from part 1 has a serious limitation: it can handle only one connection at a time. A malicious client could easily take advantage of this limitation by opening a connection and never closing it, preventing your server from handling additional requests.

In this part, we will address this limitation by extending http-server to handle multiple requests simultaneously. The easiest way to do so (from a programmer's point of view) is to create a child process for each new connection using the fork() system call, and handle each HTTP process in that child process while the parent continues to accept() subsequent connections.

Make sure you have finished part 1 before you attempt this part. You should complete part 2 by copying your solution from part1/ to part2/, and adapting your single-process http-server code to use multi-process request handling.

Here are some hints and requirements:

- Remember that after a server accept()s a client connection, it will have two socket file descriptors: one for the client, and one for the server. You should close anything you don't need as early as possible; in particular, the child process should close the server socket (since it does not need to accept() any new connections), and the parent should close the child socket (since it does not need to communicate over the client connection).

- Don't forget to terminate your children after they finish handling the client request; you can do so by calling exit(0).

- Don't let your children become zombies... At least not for too long. You should install a SIGCHLD signal handler in the parent process that reaps any dead children.

  You should install the SIGCHLD handler using sigaction(), and make sure to specify the SA_RESTART flag to ensure accept() is restarted after SIGCHLD is handled.

- The parent process should reap its children in a non-blocking manner, using waitpid() and the WNOHANG flag.

- Modify the logging so that it includes the PID of the child process handling the request, e.g.:

    128.59.22.109 (243157) "GET /cs3157/images/photo.jpg HTTP/1.1" 200 OK

This logging should be performed by the child process.

- This part is easiest to complete if your part 1 code is well-organized, where you separate the server accept() loop and client-handling logic; then all you have to do is add in the fork()ing logic and signal handler code, and modify the logging code.

  Compared to my part 1 solution, my part 2 solution adds fewer than 50 lines of code.

Note that you SHOULD NOT modify part 1's http-server to be a multi-process server; you should submit two separate http-server solutions, a single-process server in part1/ and a multi-process server in part2/.

Part 3: Orphans and zombies (optional)

(a)

When a process's parent dies, it becomes an "orphan," and is adopted by the init process (whose PID is 1). Write a program that creates an orphan.

Capture this condition using the ps command to prove to yourself that a process is really an orphan. Include the ps output in your README.txt along with your explanation.

Are there any other orphan processes on CLAC (that weren't created by other students)? And why might orphan processes be useful? Answer these questions in your README.txt as well.

Make sure to kill your orphans after you have completed this part.

(b)

When a process has terminated but has not been reaped by its parent (using waitpid()), it is called a zombie. Write a program that produces a zombie.

Capture this condition using the ps command to prove to yourself that a process is really a zombie. Include the ps output in your README.txt along with your explanation.

Make sure to reap your zombies after you have completed this part.

--

Good luck!