CSCI 4061: Introduction to Operating Systems Project 4: Web Server Socket Programming

Instructor: Jon Weissman

Due (Final): 11:59 PM Dec. 17, 2022

Due (Interim): 11:59 PM Dec. 9, 2022

1 Overview

In Project 3, we built a multi-threaded web server using POSIX threads and synchronization mechanisms. While doing so, several utility functions were provided which helped you implement the network communication portion of the web server. In this project, you will be implementing the networking utility functions using POSIX socket programming.

2 Description

The utility functions used in the web server are responsible for receiving requests and sending responses between clients and the server. We have provided a solution file for the Project 3 multi-threaded web server (`server.o`) that will integrate with the utility functions that you will write in `util.c` through the interface described in `util.h`.

There are five main utility functions that you must implement. The interface of each of these functions is described in detail in the provided `util.h` header file with additional comments in utils.c file. A high- level overview of what each function should do is provided below.

```
void init(int port);
```

This function is run once by the web server `main()` during initialization to set up a socket for receiving client connection requests on the `port` provided. The newly created socket should be used by all dispatch threads for accepting client connection requests.

```
int accept_connection(void);
```

This function is called repeatedly by each dispatch thread to accept a new client connection request. If successful, it should return the new socket descriptor for communicating with the client.

```
int get_request(int fd, char *filename);
```

This function is called repeatedly by each dispatch thread after a call to `accept_connection()` to receive an HTTP GET request from the client. It should parse the request and, if successful, copy the requested file path into the `filename` output parameter.

```
int return_result(int fd, char *content_type, char *buf,
int numbytes);
```

This function is called repeatedly by each worker thread to return a successfully handled file request back to the client.

```
int return_error(int fd, char *buf);
```

This function is called repeatedly by each worker thread to return a failed file request back to the client.

Only one request is serviced per connection, so `return_result()` and `return_error()` should close the socket when complete.

3 Returning Results

When a client makes a request to your web server, it will send an HTTP GET request formatted similar to the text below. For our web server, we only care about the first two strings (the request type and the file path) in the first line. Everything else can be safely ignored.

```
GET /path/to/file.html HTTP/1.1
(Other header information)
(Blank line)
```

When returning a file to the client, you should follow the HTTP protocol. Specifically, if everything went OK, you should write back the following to the socket descriptor (each line ends in a newline):

```
HTTP/1.1 200 OK
Content-Type: [content-type-here]
Content-Length: [file-length-in-bytes-here]
Connection: Close
(Blank line)
[file-contents-here]
```

For example:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 11
Connection: Close

Hello World
```

Similarly, if something went wrong, you should return an error back to the client by writing the following to the socket descriptor:

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: [error-message-length-in-bytes-here]
Connection: Close
(Blank line)
[error-message-here]
```

For example:

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 25
Connection: Close

Requested file not found.
```

The C standard library functions `sscanf()` and `sprintf()` may be useful for formatting requests and responses.

4 Passive Socket Port Reuse

The web server may be terminated at any time. When an unexpected termination occurs, by default the port number used by the web server continues to be held by the OS until a timeout has occurred. To avoid having to wait for the port number to be released after the timeout, the socket created for accepting client connection requests during `init()` should be setup for port reuse. This can be done using `setsockopt()` with the `SO_REUSEADDR` option. This will allow us to restart our server on the same port immediately after a previous run.

5 Request Error Handling

The web server may receive bad requests when dealing with many clients, some of whom may be sending malicious attacks. Malicious attacks may try to access files outside of the server root directory or cause a buffer overflow to gain access to or crash the server. Our network programming code should be able to handle invalid client requests without causing our web server to crash.

As always, you should error check your system calls and either `exit()` or return an error to the server code depending on the behavior described in `util.h`. For errors that do not result in termination of the server, you should also make sure to release any resources associated with that connection.

In addition, you should check for the following errors in client HTTP requests:

- The first line of the request should contain at least two strings (the request type and the requested file) separated by a space
- The request type should be a "GET" request
- The requested file should not contain ".." or "//" and should be a maximum of 1023 bytes in length. This is to prevent a malicious user from being able to access a portion of the file system other than the web server file root

A helper function makeargv() is provided in utils.c file to help you with request string parsing. Look at the comments in utils.c for description and usage. But you are free to parse any way you wish.

6 How to run the server

Run it as you did for Project 3. The only change is that because you are no longer responsible for the threads, the cache, and the request queue, those values no longer need to be passed as command line arguments. They are handled entirely in server.o. So the command to run the server is now:

```
./web_server <port> <path>
```

7 Provided Files and How to Use Them

We have provided some starter files which you should use to complete this assignment:

- `util.c`: This is the only file that requires any modifications for this project. We have provided a template file to get you started.
- `server.o`: A solution to the Project 3 portion of the web server
- `util.h, Makefile, testing.zip, web_server_sol`: All serve the same purpose as from Project 3 (please refer to section 10 of Project 3)

8 How to test your server

Please refer to section 11 of Project 3 for most details.

To test `return_result()` and `return_error()`, it may be useful to examine the HTTP header that is being returned to the client. Use the “-S” flag on wget to print the server response.

```
% wget -S http://127.0.0.1:9000/image/gif/0.gif
```

By default, wget does not wait for the body of a 404 Not Found message. To test `return_error()`, use the “--content-on-error” flag on wget to receive the error message. If `return_error()` is successful, the error message sent to the client should be stored in the file name specified in the wget request.

```
% wget –-content-on-error http://127.0.0.1:9000/image/gif/not_a_file
% cat not_a_file
Requested file not found.
```

9 Defines and Simplifying Assumptions

1 The server should accept a backlog of up to 20 connection requests

2 The maximum size of a GET request will be 2047 bytes (2048 byte buffer with NULL terminator)

3 The maximum size of a GET response header (200 OK or 404 Not Found) will be 2047 bytes (2048 byte buffer with NULL terminator). Note that this is just for the header. The file size or error message may be much larger.

10 Interim Evaluation

Requirements: Your web server should be able to accept connections and print out the first line of a single received request. Only the first line is required since the web server does not care about the remaining portion of the GET request. At this point your web server does not have to handle parsing the request or responding. In order to accomplish this you must have completed `init()`, `accept_connection()`, and a very small portion of `get_request()`.

11 Deliverables

A tar file containing your code and a Makefile that can be used to build and execute your code, and a README file. The README indicates how the work was partitioned within your group. There could be overlap. We will check this to see that everyone participated. You submit ONLY ONE tar file per group. You will be docked points if you do not follow these instructions.

At the top of your README file, please include the

```
/* CSCI-4061 Fall 2022 – Project #4
 * Group Member #1: <name> <x500>
 * Group Member #2: <name> <x500>
 * Group Member #3: <name> <x500> */
```

12 Grading (Tentative)

5% README

10% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers

15% Interim evaluation

70% Correctness, error handling, meeting the specification. Broken down, tentatively, as follow:

  - 20% for valid request handling
  - 10% for invalid request handling
  - 10% for error checking (system calls, C standard library I/O, etc.)

- 30% for code evaluation and meeting the specification