The Lean Webserver

George Mamidakis - ics22091

September 2, 2024

1 Introduction

The Lean Webserver is a multithreaded webserver employing prethreading created as a project for the Parallel & Distributed Computing course (CSC603) of the University of Macedonia taught by Margariths Kwnstantinos. The project was inspired by the Tiny webserver developed for the CS:APP [BO15a] book. The following paper is a complementary document where I discuss the structure of the project, design choices, problems that arose during development, benchmarking, and future work related to this and relevant projects. All development was done on an Ubuntu 23.04 VM.

2 Development

2.1 Inspiration

As mentioned, the project was inspired by the CS:APP book. In it, Randal E. Bryant and David R. O'Hallaron created the Tiny webserver [BO15b], a simple webserver serving both static and dynamic (in the form of the cgi interface). The server implemented simple client error handling for different HTTP status codes (specifically 501 for non implemented methods, and 404 for a non-existent file), path parsing to serve the correct file and filetype, and serving of cgi scripts. A noteable implementation detail of that webserver is the usage of the CSAPP API, a set of wrapper functions to make using C easier for students and readers of the book. These functions range from simple wrapper functions primarily for error handling

```c
void *Malloc(size_t size)
{
    void *p;

    if ((p  = malloc(size)) == NULL)
  unix_error("Malloc error");
    return p;
}
```

Listing 1: malloc wrapper

to big packages of functions, like the Robust Input Output package, which is used to provide an easier interface to perform different IO operations like receiving a line over a file descriptor

```c
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
{
    int n, rc;
    char c, *bufp = usrbuf;

    for (n = 1; n < maxlen; n++) {
        if ((rc = rio_read(rp, &c, 1)) == 1) {
      *bufp++ = c;
      if (c == '\n') {
                n++;
        break;
            }
  } else if (rc == 0) {
      if (n == 1)
    return 0; /* EOF, no data read */
      else
    break;    /* EOF, some data was read */
  } else
      return -1;    /* Error */
    }
    *bufp = 0;
    return n-1;
}
```

Listing 2: Robust IO implementation of a recvline function

A big part of the project's initial statement was to not rely on the provided API, and instead use the standard Linux API functions by myself, performing any other operation or error handling on my own. This did not prove to be difficult, since as mentioned most were simple wrapper of their original functions.

2.2 Development of Lean

The general flow code follows that of the tiny webserver. We have a main function to bind the port to listen on for incoming connection, which passes the client file descriptor to a 'handle_request' function which is where the main webserver logic starts. This simple calling convention of the function later made porting it to the threading logic quite easy

2.2.1 handle_request

```c
void handle_request(int client_fd)
{
    char request[BUFFER_SIZE];
    int bytes_read = recv(client_fd, request, BUFFER_SIZE - 1, 0);
    if (bytes_read <= 0)
    {
        perror("Error reading request:");
        return;
    }
    request[bytes_read] = '\0';
    printf("Request:\n%s", request);

    char method[BUFFER_SIZE], path[BUFFER_SIZE], version[BUFFER_SIZE];
    parse_request(request, method, path, version);

    if (strncmp(method, "GET", 4) != 0)
    {
        client_error(client_fd, version, method, 501, "Not Implemented", "Lean doesn't
 support this method");
        return;
    }

    char filename[BUFFER_SIZE];
    parse_path(path, filename);
    send_response(client_fd, version, filename);
}
```

Listing 3: The handle_request function

The flow of the code is simple, it uses the 'recv' function to read the HTTP request headers, and leaves one byte so we can set the null byte later and not cause problems with the data. The BUFFER_SIZE constant is 1024, which did not create problems during development even when tested against a real browser. Proper error handling is performed. Afterwards 3 arrays are created holding the method used by the request, the path of the requested file, and and the HTTP version sent by the server (e.g. 1.0, 1.1, ...). These are then handled in the 'parse_request' function, which is a simple sscanf call

```c
void parse_request(char *request, char *method, char *path, char *version)
{
    sscanf(request, "%s %s %s", method, path, version);
}
```

Listing 4: The parse_request function

The rest of the HTTP header is of no interest to us in the context of the Lean webserver. After getting these values, we check if the method used is GET. If not, we send a 501 as a response, since only GET is implemented in the Lean webserver. If a GET request was issued, the request path is parsed to get the file

```c
void parse_path(char *path, char *fname)
{
    strcpy(fname, ".");
    strcat(fname, path);
    if (path[strlen(path)-1] == '/')
        strcat(fname, "index.html");
}
```

Listing 5: The parse_path function

Essentially, it starts from the path where the server binary is running, and checks if a slash is used. If a / is found, that means a path to a file is specified. If not, then we specify ‘index.html‘ as the standard html file to serve. This is a common behaviour of big webservers as well. Lastly, a call to ‘send_response‘ is sent, with the client file descriptor, the HTTP version, and the filename as arguments. Before we go into the ‘send_response‘ function, the ‘client_error‘ will be briefly discussed.

2.2.2 client_error

The client_error function was developed as a general function for handling errors. It has many arguments to it, and dynamically constructs a HTML page that is sent to the client

```c
void client_error(int cfd, char *version, char *error_cause, int error_code, char *
    error_msg, char *error_description)
{
    char html[DATA_SIZE], response[BUFFER_SIZE];

    sprintf(html, "<html><title>Lean Error</title>");
    sprintf(html, "%s<body><h1>%d: %s</h1>\r\n", html, error_code, error_msg);
    sprintf(html, "%s<p>%s: %s</p></html>\r\n", html, error_description, error_cause);

    create_http_response_headers(response, version, error_code, error_msg, strlen(html
    ), "text/html");

    send(cfd, response, strlen(response), 0);
    send(cfd, html, strlen(html), 0);
}
```

Listing 6: The client_error function

On top of that, it calls another function, ‘create_http_response_header‘, which is another abstraction to a more general function for creating the response headers

```c
void create_http_response_headers(char *resp, char *version, int status_code, char*
    status_msg, int content_length, char *content_type)
{
    sprintf(resp, "%s %d %s\r\n", version, status_code, status_msg);
    sprintf(resp, "%sServer: Lean Web Server\r\n", resp);
    sprintf(resp, "%sConnection: close\r\n", resp);
    sprintf(resp, "%sContent-length: %d\r\n", resp, content_length);
    sprintf(resp, "%sContent-type: %s\r\n\r\n", resp, content_type);
}

```

Listing 7: The create_http_response_header function

The ‘sprintf‘ function was chosen to construct the dynamic string concatenation, inspired by it’s usage in the Tiny webserver and it’s ease in dealing with data types beyond strings like ints. The function ‘strcat‘ could be used alternatively, though it requires better handling than ‘sprintf‘, because conversion of other types to strings is required, and it keeps concatenating to the end of the buffer after one request is served, which can potentially easily lead to a buffer overflow. Though ‘sprintf‘ is also an unsafe function, the strict calling of ‘client_error‘ mitigates this a bit. However it should be noted that ‘version‘ could be controlled by an attacker.

2.2.3 send_response

This function acts like client_error, but instead here we have reading of a general data file, like a HTML file.

```c
void send_response(int cfd, char *version, char *filename)
{
    char response[BUFFER_SIZE], data[DATA_SIZE];

    int file_fd = open(filename, O_RDONLY);
    if (file_fd < 0)
    {
        client_error(cfd, version, &filename[1], 404, "Not found", "Lean couldn't find this file");
        return;
    }
    else
    {
        int filesize = 0, bytes_read = 0;
        while ((bytes_read = read(file_fd, data, DATA_SIZE)) > 0)
            filesize += bytes_read;
        data[filesize] = 0;

        char filetype[BUFFER_SIZE];
        get_filetype(filename, filetype);

        create_http_response_headers(response, version, 200, "OK", strlen(data), filetype);

        printf("Response:\n%s", response);

        send(cfd, response, strlen(response), 0);
        send(cfd, data, filesize, 0);

        close(file_fd);
    }
}
```

Listing 8: The send_response function

First we try and get a file descriptor to the data file we want to serve to the client. Per the open manpage, open() returns the new file descriptor (a non-negative integer), or -1 if an error occurred. So from that we could get that the file requested does not exist, and thus return a 404 error. Otherwise if it exists, it is read into a buffer of size DATA_SIZE = 8192, and then a call to ‘get_filetype‘ is performed before the response headers and data file are sent back to the client

```c
void get_filetype(char *fname, char *ftype)
{
    if (strstr(fname, ".html"))
      strcpy(ftype, "text/html");
    else if (strstr(fname, ".gif"))
      strcpy(ftype, "image/gif");
    else if (strstr(fname, ".png"))
        strcpy(ftype, "image/png");
    else if (strstr(fname, ".jpg"))
      strcpy(ftype, "image/jpeg");
    else
      strcpy(ftype, "text/plain");
}

```

Listing 9: The get_filetype function

A simple pattern matching and string copying function. This covers the functionalities of the core ‘lean.c‘ file.

2.3 Development of Prethreading

Prethreading is covered in Chapter 12 of CSAPP, where the usage of semaphores is first introduced to tackle time scheduling of common resources between threads [BO15c]. Classic problems of time

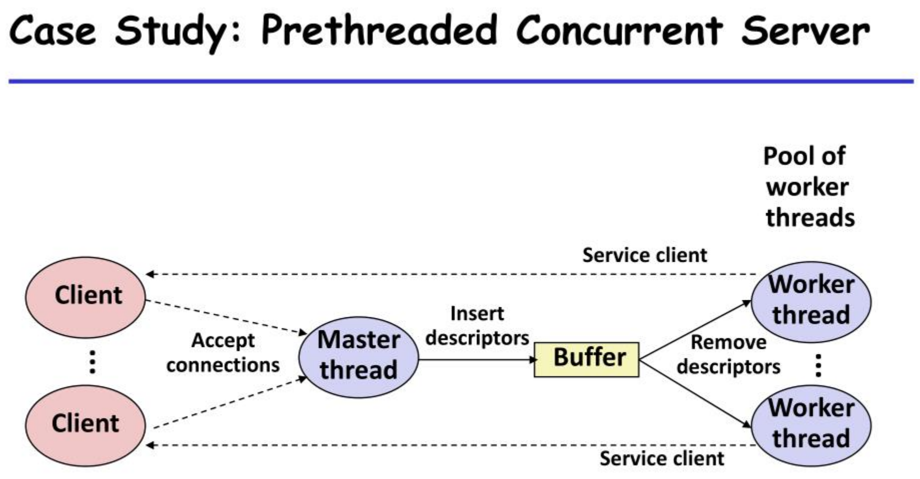


Figure 1: A diagram example of a prethreading server architecture.

scheduling of common resources are the producer-consumer problems. In them, the producer thread creates and adds new data, while the consumer uses and removes existing data from common memory areas. While mutual exclusion can be achieved, there are many cases when the applications employing such logic are time sensitive. A producer can not wait to access memory until a spot is available, and a consumer can not access it if the memory is empty. These two problems can be solved by creating a FIFO queue shared buffer API [BO] which initializes a mutex, available slots, and data items in storage, and locks them whenever a data is removed from or inserted into the shared buffer.

When we refer to the Lean server as a multithreaded server with prethreading, we mean that it will employ the sbuf API. We will create a threadpool of NTHREADS threads, where each will be used to serve a new client. This will attempt to share the workload of serving the clients.

As mentioned, the sbuf API developed for CSAPP relies on using semaphores. However in the context of CSC603, we were taught the use of mutexes predominantly. So an attempt was made on my part to convert the semaphore logic to mutex logic. Searching for online implementations, we uncover this threadpool implementation on Github [pmi], which uses the pthread API to perform mutual exclusion. Combining the two, this is the main threading logic employed by the program.

2.3.1 The main program

Below is the main program, which serves as a handler of the prethreading logic, as well as a wrapper around the Lean webserver. After covering it, we will proceed to analyze the sbuf API implementation

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "lean.h"
#include "prethreading.h"

#define SHAREDBUFSIZE  4
#define NTHREADS  16

static struct sbuf_t shared_buf;
```

```

int open_serverfd(int port)
{
    int fd = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)))
    {
        perror("bind error:");
        exit(2);
    }

    if (listen(fd, 1))
    {
        perror("listen error:");
        exit(2);
    }

    return fd;
}

void *thread_handler(void *ptr)
{
    while (1)
    {
        int client_fd = sbuf_remove(&shared_buf);
        handle_request(client_fd);
        close(client_fd);
    }
}

int main(int argc, char **argv)
{
    int server_fd;
    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        fprintf(stderr, "using default port 1337\n");
        server_fd = open_serverfd(1337);
    }
    else server_fd = open_serverfd(atoi(argv[1]));

    sbuf_init(&shared_buf, SHAREDBUFSIZE);
    pthread_t threads[NTHREADS];
    for (int i = 0;  i < NTHREADS; i++)
        pthread_create(&threads[i], NULL, thread_handler, NULL);

    while (1)
    {
        struct sockaddr_storage caddr;
        socklen_t caddr_len = sizeof(caddr);
        int client_fd = accept(server_fd, (struct sockaddr *) &caddr, &caddr_len);
        sbuf_insert(&shared_buf, client_fd);
    }
}
```

Listing 10: The main program

The main function gets arguments to define the port the server will listen on for incoming connections. The ‘open_serverfd‘ function performs simple socket operations to bind the given port. Afterwards, ‘sbuf_init‘ is called to initialize the global ‘shared_buf‘ variable of type ‘sbuf_t‘. After that a loop creates the worker threads which will handle the requests inside ‘thread_handler‘. After they are set, a set of client connections are created upon connection to the server, and are inserted into the ‘shared_buf‘ variable.

On the other hand inside ‘thread_handler‘, ‘sbuf_remove‘ removes a client file descriptor from the global variable, and passes it inside the ‘handle_request‘ function which was explained earlier. The while true loops inside the main thread and the workers ensures there is constant handling of incoming requests and optimal usage of available client file descriptors inside the common buffer. Problems such as mutual exclusion will be answered in the explanation of the prethreading API.

2.3.2 The Prethreading API

Below are both the source and header file of the prethreading API

```c
// prethreading.c
#include "prethreading.h"

void sbuf_init(struct sbuf_t *sb, int n)
{
    sb->buf = (int *) malloc(n * sizeof(int));
    sb->n = n;
    sb->front = sb->rear = 0;
    pthread_mutex_init(&sb->mutex, NULL);
    pthread_cond_init(&sb->slots, NULL);
    pthread_cond_init(&sb->items, NULL);
}

void sbuf_delete(struct sbuf_t *sb)
{
    free(sb->buf);
    sb->buf = NULL;
    pthread_mutex_destroy(&sb->mutex);
    pthread_cond_destroy(&sb->slots);
    pthread_cond_destroy(&sb->items);
}

void sbuf_insert(struct sbuf_t *sb, int item)
{
    pthread_mutex_lock(&sb->mutex);
    while ((sb->rear - sb->front) == sb->n)
        pthread_cond_wait(&sb->slots, &sb->mutex);
    sb->buf[sb->rear % sb->n] = item;
    sb->rear++;
    pthread_cond_signal(&sb->items);
    pthread_mutex_unlock(&sb->mutex);
}

int sbuf_remove(struct sbuf_t *sb)
{
    pthread_mutex_lock(&sb->mutex);
    while (sb->front == sb->rear)
        pthread_cond_wait(&sb->items, &sb->mutex);
    int item = sb->buf[sb->front % sb->n];
    sb->front++;
    pthread_cond_signal(&sb->slots);
    pthread_mutex_unlock(&sb->mutex);
    return item;
}

// prethreading.h
#include <pthread.h>
#include <stdlib.h>

struct sbuf_t
{
    int *buf;
    int n;
    int front;
    int rear;
    pthread_mutex_t mutex;
    pthread_cond_t slots;
    pthread_cond_t items;
};

void sbuf_init(struct sbuf_t *sb, int n);
void sbuf_delete(struct sbuf_t *sb);
void sbuf_insert(struct sbuf_t *sb, int item);
```

```c
int sbuf_remove(struct sbuf_t *sb);
```

Listing 11: The prethreading (or sbuf) API

For starters, ‘sbuf_init‘ remains very much the same, with the main difference being that instead of initializing semaphores, we are initializing both a pthread mutex, and two pthread condition variables.

Next, ‘sbuf_delete‘ has a big change, not only freeing the allocated heap chunk of the shared buffer (and setting the heap pointer to NULL to mitigate potential uase-after-free vulnerabilities), but also destroying the mutex and condition variables.

Then the two main operations ‘sbuf_insert‘ and ‘sbuf_remove‘ remain mainly the same, with the exception of how the conditions are handled. The two conditions are interchanged between insert and remove, making sure that an element is removed if there are items in it, and that an item is inserted if there are available slots. For both operations the mutex ensures that an operation is atomic, since the shared_buf struct is shared amongst all threads.

2.4 Problems During Development

A main problem during development was after developing the alpha version of the webserver and benchmarking it, the program yould have very poor performance. To measure the proper performance of the threading logic, I ported the tiny webserver into the threading logic to compare. The difference between the 2 version could be as much as the lean server being over 22 times slower than the tiny server. A possible root cause was the use of a custom ‘recvline‘ function similar to that of the rio package

```c
ssize_t recvline(int cfd, char *buf)
{
    ssize_t i = 0;
    while (1) // else for (i = 0; i < MAXSIZE; i++)
    {
        char c;
        ssize_t rcvd_len = recv(cfd, &c, 1, 0);
        if (rcvd_len == 1)
        {
            i++;
            *buf++ = c;
            if (c == '\n')
                break;
        }
    }
    *buf = '\0';
    return i;
}
```

Listing 12: A custom recvline function

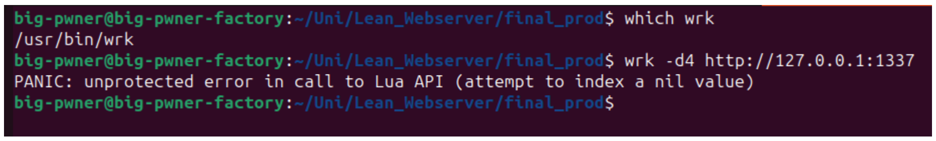
However after performing some analysis on the time to respond between my ‘recvline‘ and the rio ‘recvline‘, it came out that there was no real difference over one request. So the problem was over handling of multiple requests. After many hours tinkering, some better error handling in the original receiving of the request headers seemed to do the job. Also it should be mentioned that the standard ‘recv‘ function was chosen over using the ‘recvline‘ due to no need for the basic case of our program (i.e. we don’t need to receive the headers line by line to parse them to extract any information).

Another problem was the difficulty of including the ‘sbuf_delete‘ function to clean up the threads and the shared buffer. An idea was to create a signal handle for the SIGINT signal. However problems arose specifically in regards to the threads ending. Obvioiusly in such an implementation, the threads, server file descriptor, and shared buffer need to be global

```c
static pthread_t threads[NTHREADS];
static struct sbuf_t shared_buf;
static int server_fd;

void server_shutdown()
{
    printf("Lean shutting down gracefully...\n");

    for (int i = 0; i < NTHREADS; i++)
      pthread_join(threads[i], NULL);
```



```
big-pwner@big-pwner-factory:~/Uni/Lean_Webserver/final_prod$ which wrk
/usr/bin/wrk
big-pwner@big-pwner-factory:~/Uni/Lean_Webserver/final_prod$ wrk -d4 http://127.0.0.1:1337
PANIC: unprotected error in call to Lua API (attempt to index a nil value)
big-pwner@big-pwner-factory:~/Uni/Lean_Webserver/final_prod$
```

Figure 2: An error in the LuaJIT component of wrk

```

    sbuf_delete(&shared_buf);
    close(server_fd);
    exit(0);
}
...
signal(SIGINT, server_shutdown);
```

Listing 13: A potential signal hooking function to gracefully kill Lean

Beyond these problem which consumed multiple working hours and made me remember how punishing C can be, everything else proceeded smoothly and kept my faith and love in C.

3 Benchmarking

3.1 Benchmarking Setup

Benchmarking was previously mentioned in 2.4, but not in-depth. After a quick search for HTTP benchmarking, I came across the wrk tool [Glo], which serves our purpose. However the error shown in 2 would appear upon execution

This seems to be a documented issue since 2017, but hasn't been resolved since then. On the comments of the issue however, a fork of wrk was mentioned which didn't have the above error. So this fork of wrk was used for benchmarking going forward.

To provide proper comparison between different implementations, I decided to benchmark:

1. My custom Lean Webserver without prethreading
2. The CSAPP Tiny Webserver without prethreading
3. My custom Lean Webserver & prethreading implementation
4. My custom Lean Webserver implementation with the CSAPP prethreading implementation
5. The CSAPP Tiny Webserver implementation with my custom prethreading implementation
6. The CSAPP Tiny Webserver & prethreading implementation

The wrk flags used for benchmarking where:

- -c, –connections, Connections to keep open
- -d, –duration, Duration of test
- -t, –threads, Number of threads to use

3.2 Performing Benchmarking

Based on the aforementioned flags, we set the connections to 400, the threads to 8, and the duration to 4 seconds. The numbers inside the table represent the number of requests received and handled during that time period. Also for the CSAPP prethreading logic I used the code developed by DreamAndDead [Dreb]. In it, he developed another technique for thread balancing, which I removed from the testing for fairness of results.

To begin, it should be noted that performance may vary. Implementations that have on all 3 runs over 10000 may produce runtimes with performances below 10000. So it's obvious that to get a more representative approximation of each implementations actual performance, we would require

Execution	Simple Lean	Simple Tiny	Lean Prethreading Custom	Lean Prethreading CSAPP	Tiny Prethreading Custom	Tiny Prethreading CSAPP
1st	9888	9466	10286	14985	13044	11604
2nd	16123	9135	13314	17736	12431	13607
3rd	15737	9121	12977	16619	11736	12931

Table 1: Benchmarking results

more than 3 executaion instances. However they can still provide us with some answers. Below are some data we can infer from the benchmark table:

1. The simple tiny implementation performed the poorest, being unable to respond to over 10000 in all execution instances. So a case might be made that while the CSAPP API provides a friendlier programming experience, it shouldn't be used for production projects
2. The best performing implementation is the Lean webserver with the CSAPP prethreading logic. In this case, this might be an indicator as to the CSAPP sbuf API providing better performance, potentially due to how semaphores are handled.
3. When it comes to the Tiny benchmarking, no real difference can be identified between the custom and csapp prethreading logics. This, along with 1. might indicate that Tiny is slow in general, however the custom prethreading implementation doesn't indicate and performance gains.
4. The standalone Lean implementation has some high performances for the second and third executions, which are the 2nd and 4th highest performances accross the board respectively. Along with 2. this could indicate that the Lean webserver is more optimized.

Considering the times produced by the simple lean implementation, a question arises in regards to the potential performance gains of the prethreading technique. Or more specifically the performance of the implementations tested, since no impactful advantage can be gained by introducing prethreading. A good example of the ideal performance of a prethreading server can be found in DreadAndDead's website [Drea]. Even though as we noted the code had a function to adjust the number of threads based on the load of the server, the results are 90380 total requests served, which translates to 22039.72 requests every second for 4 seconds. Though 2 threads and 10 connections were used for benchmarking, benchmarking the exact same code as the one found on the repo, our results are 15002, 15110, and 12190 requests for 3 different execution instances. A noticable improvement over the previous results, but nowhere near 90380. Especially considering the increased connections and available threads.

This could potentially raise questions about other important factors in benchmarking besides the code itself. After all as mentioned, the wrk used for benchmarking is a fork of the original one. We can't know which version DreamAndDead used, however it's still a factor which could have caused such a difference in data. Another one is the underlying hardware of the machine used, and the fact that a virtual machine was used. Perhaps running on bare metal could have provided better results. But still the performance of the singlethreaded Lean webserver is something that questions the potential effectiveness of some factors in painting a more proper picture of the actual performance of each implementation. Lastly there are some differences between what the implementations output, which might make a difference considering that output is a time consuming process. Though that still potentially accounts for some millisecons, rather than even a second.

It should be noted that wrk produces more data, like the average number of requests per second, information about latency, the size of the data transferred and many more. Also it should be noted that most of the implementation's processes stopped after one or two tests with wrk. More detailed benchmarking results can be found in the accompanying files.

4 Future Work

There is room for future work on this project on many fronts to make it a proper HTTP server.

4.1 Enriching the HTTP Protocol Support

The HTTP functionalities of the server are implemented to a minimal, as the Lean server only implements the GET method and just displaying back HTML and various other data forms. Proper HTTP webservers have multiple other methods like POST, PUT, HEAD, DELETE, and many more. Also, it supports a multitude of HTTP headers. In our code, no HTTP header is explicitly supported in terms of it being parsed and different actions being taken based on that. And on the response, the response headers being sent are kept to a functioning minimal. Lastly, HTTP has a range of status codes, all having a different meaning and improtance, which should be supported as well. To consider we have a proper HTTP server, we need to support the entire HTTP protocol.

4.2 Supporting HTTPS

Going beyond simple HTTP, HTTPS is a modern standard which should be enforced by every website. So a webserver should support it.

4.3 Enriching the Webserver

Besides the HTTP(S) protocols, modern webservers have a multitude of other features, such as log files, configuration files, and many more.

4.4 Identifying Prethreading Performance Root Cause(s)

As mentioned, there is no impactful performance gains from implementing prethreading, in contrast to some public benchmarking [Drea] that could support otherwise. So further research needs to be performed to potentially find the root cause or causes of the recorded performances.

4.5 Implementing More Sophisticated Threading Techniques

Another direction the research of threading could go towards is including more threading techniques beyond prethreading. An example is the aforementioned technique of adjusting dynamically threads.

5 Conclusion

This was a very interesting project, and it was enjoyable thinking about how to implement some standard HTTP functionalities in C. It would have been better if I had gotten clearer better results for the prethreading, however as discussed the results probably go beyond potential errors in my implementation. I would like to thank professor Margariths for giving me the opportunity to do this project, and for the help and guidance he gave me in learning threading concepts during this semester.

References

[BO15a] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective 3rd Edition*. Pearson, 2015. ISBN: 9780134092669.

[BO15b] Randal E. Bryant and David R. O'Hallaron. "Computer Systems: A Programmer's Perspective 3rd Edition". In: Pearson, 2015. Chap. 11.6.

[BO15c] Randal E. Bryant and David R. O'Hallaron. "Computer Systems: A Programmer's Perspective 3rd Edition". In: Pearson, 2015. Chap. 12.5.4.

[BO] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective 3rd Edition*. URL: https://csapp.cs.cmu.edu/3e/ics3/code/conc/sbuf.c. (last accessed: 06.07.2024).

[Drea] DreamAndDead. *CSAPP 12.38 Solution Benchmarking*. URL: https://dreamanddead.github.io/CSAPP-3e-Solutions/chapter12/12.38/. (last accessed: 08.07.2024).

[Dreb] DreamAndDead. *CSAPP 12.38 Solution Code*. URL: https://github.com/DreamAndDead/CSAPP-3e-Solutions/blob/master/site/content/chapter12/code/12.38/main.c. (last accessed: 08.07.2024).

[Glo] Will (wg) Glozer. *Modern HTTP benchmarking tool*. URL: https://github.com/wg/wrk. (last accessed: 08.07.2024).

[pmi] pminkov. *A simple thread pool in C*. URL: https://github.com/pminkov/threadpool. (last accessed: 06.07.2024).