

# Computer Operating Systems

BLG 312E

## Project 3 Report

EREN CULHACI

150220763

culhaci22@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 24.05.2024

# CONTENTS

0.1	Introduction .....	2
0.2	Modifications.....	2
0.2.1	server.c.....	2
0.2.2	client.c.....	3
0.2.3	Makefile .....	5
0.3	Outputs and Results .....	5
0.4	Conclusions.....	8

**Abstract**—This project introduces a multithreaded HTTP client in C, designed for concurrent communication with multiple HTTP servers. Client creates threads to handle each request concurrently, enhancing efficiency. The client offers scalability and is suitable for scenarios requiring rapid communication with multiple servers, such as web crawling or distributed data retrieval. The server handles multiple requests by a multi-threaded approach with its new functionality.

## 0.1 Introduction

In this project, I managed to implement a new functionality in a given web server and client. I improved the server by implementing a multi-threaded approach into the web server. I tested it by altering the client to work as multi-threaded and send requests simultaneously.

## 0.2 Modifications

In order to add multi-thread functionality in server and client I modified some codes.

### 0.2.1 server.c

server.c file includes different functions to handle multi-threading. It handles incoming connections concurrently. To break the code down:

#### Header Files and Definitions

We included `blg312e.h` and `request.h` as necessary files, and defined constants as `MAX_THREADS` and `MAX_BUFFERS` representing the max number of worker threads and connection buffers.

#### Connection Buffer Structure (`connection_buffer_t`)

This structure defines a shared connection buffer using an array of connection descriptors (`int connections[MAX_BUFFERS]`). It also includes front and rear index to track the position of the elements in buffer. Semaphores (`sem_t`) are used for synchronization and mutual exclusion:

- `mutex`: Protects access to the buffer.
- `slots`: Counts the number of empty slots in the buffer.
- `items`: Counts the number of filled slots in the buffer.

#### Global Variables

`conn_buffer`: Shared connection buffer.

`worker_threads`: An array to hold worker threads.

## Functions

- `init_connection_buffer(int buffers)`: Initializes the connection buffer and associated semaphores.
- `enqueue_connection(int connfd)`: Adds a connection descriptor to the buffer.
- `dequeue_connection()`: Removes a connection descriptor from the buffer.
- `worker_thread(void *arg)`: Function executed by worker threads. It dequeues a connection descriptor, handles the request associated with it, and closes the connection.
- `getargs(int *port, int *threads, int *buffers, int argc, char *argv[])`: Parses command-line arguments to obtain the port number, number of threads, and number of buffers.

## Main Function

The main function:

- Parses command-line arguments and initializes the connection buffer.
- Creates worker threads based on specified number.
- Opens a listening socket and accepts incoming connections.
- For each accepted connection, it enqueues the connection descriptor to the shared buffer.

## Concurrency

Multiple worker threads (`pthread_t worker_threads[MAX_THREADS]`) concurrently dequeue connection descriptors from the shared buffer and handle requests which associated with them. Semaphores (`mutex, slots, items`) ensuring the thread safety and preventing race conditions when accessing the shared buffer. So in brief, I implemented a multithreaded server functionality where multiple threads handle incoming connections concurrently and provide concurrent processing for requests.

### 0.2.2 client.c

#### Header Files and Definitions

The code includes necessary headers "`blg312e.h`" which includes declarations like `Open_clientfd`, `Gethostname`, `Rio_writen`, `Rio_readlineb`, etc.

## Constants and Data Structures

- `MAX_THREADS`: Max threads that can be created.
- `MAX_URL_LENGTH`: Maximum length of a URL.
- `request_info_t`: A structure to hold information about HTTP request, such as the host, port, and filename.

## Global Variables

- `pthread_t threads[MAX_THREADS]`: An array to hold thread IDs.
- `int thread_count`: Counter to keep track of the number of threads that are created.

## Function Definitions

- `clientSend(int fd, char *filename)`: Sends an HTTP GET request for the specified filename over provided socket descriptor `fd`.
- `clientPrint(int fd)`: Reads and prints the HTTP response received on the provided socket descriptor `fd`.
- `request_handler(void *arg)`: A thread routine that handles an individual HTTP request. It takes a `request_info_t` argument, opens a connection to the specified host and port, sends an HTTP request, reads and prints the response, and then closes the connection.

## Main Function

- Parses command line arguments to extract the filename of an input file containing multiple URLs.
- Opens the input file and reads URLs line by line.
- Parses each line to extract the hostname, port number, and filename.
- Allocates memory for a `request_info_t` structure, copies the hostname and filename into it.
- Creates a new thread for each URL using `pthread_create`.
- Waits for all threads to finish with `pthread_join`.
- Exits the program.

So in brief, I managed to create a client that reads URLs from an input file line by line, extracts the hostname, port number, and filename from each line, and creates a new thread to handle each HTTP request concurrently which is provided by multi-threaded approach.

I also added the `url.txt` file which you can add URLs in it to send requests.

### 0.2.3 Makefile

Since I also used threads in `client.c`, while compiling `client.c`, I need to add `-lpthread` library at the end of the compilation code like this:


```
gcc -o client client.o blg312e.o -lpthread
```

So I added this in Makefile:

```
$(CC) $(CFLAGS) -o client client.o blg312e.o $(LIBS)
```

## 0.3 Outputs and Results

First to compile every file with Makefile we got and create output files, I simply type `make` in Linux terminal.



```
root@vm_docker:/home/Ubuntu/hostVolume/HW3_codes# make
gcc -g -Wall -o server.o -c server.c
gcc -g -Wall -o request.o -c request.c
gcc -g -Wall -o blg312e.o -c blg312e.c
gcc -g -Wall -o server server.o request.o blg312e.o -lpthread
gcc -g -Wall -o client.o -c client.c
gcc -g -Wall -o client client.o blg312e.o -lpthread
gcc -g -Wall -o output.cgi output.c
mkdir -p public
cp output.cgi favicon.ico home.html public
root@vm_docker:/home/Ubuntu/hostVolume/HW3_codes#
```

Figure 0.1: Make

Then I start my server with:

```
./server [portnum] [threads] [buffers]
```

i.e **./server 5003 8 16**

for port number 5003, 8 threads and 16 buffers.

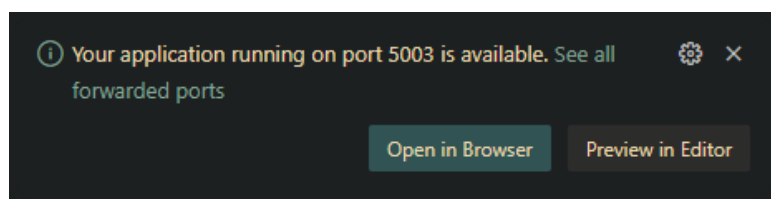
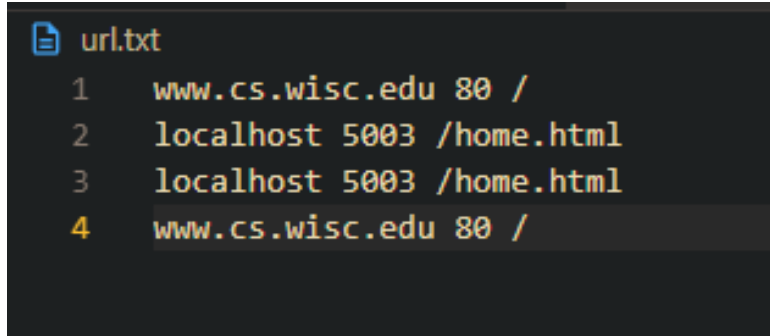


Figure 0.2: Server running on port 5003

Now we can send requests to this server and check if it handles concurrent request in multi-threaded fashion.

In url.txt we have the URLs, portnumbers and files for different URLs to send requests like this:

A screenshot of a text editor showing a file named 'url.txt'. The file contains four lines of text, each representing a request. The lines are numbered 1 through 4 on the left. Line 1: 'www.cs.wisc.edu 80 /'. Line 2: 'localhost 5003 /home.html'. Line 3: 'localhost 5003 /home.html'. Line 4: 'www.cs.wisc.edu 80 /'. The text is in a monospaced font with a light blue background.

```
url.txt
1  www.cs.wisc.edu 80 /
2  localhost 5003 /home.html
3  localhost 5003 /home.html
4  www.cs.wisc.edu 80 /
```

**Figure 0.3:** url.txt file

To start our client and send requests, we type this command in another terminal window:

**`./client url.txt`**

Now it will read the url.txt file and send requests concurrently with multi-threaded approach. In this example it will send localhost:5003 .html two requests and www.cs.wisc.edu:80 / two requests. These are all sent concurrently since we managed to do client multi-threaded. And the two requests which are sent to our server will be handled concurrently since we managed to do it multi-threaded either.

When we type `./client url.txt` in terminal, it will print the results of the request as shown in the next page:

```

root@vm_docker:/home/Ubuntu/hostVolume/HwB_codes# ./client url.txt
Header: HTTP/1.0 200 OK
Header: Server: blg312e Web Server
Header: HTTP/1.0 200 OK
Header: Server: blg312e Web Server
Length = 247
Header: Content-Length: 247
Header: Content-Type: text/html
Length = 247
Header: Content-Length: 247
Header: Content-Type: text/html
<html>
<html>

<head>
  <title>537 Test Web Page</title>
</head>

<head>
  <title>537 Test Web Page</title>
</head>

<body>

<h2> 537 Test Web Page</h2>

<p> Test web page: simply awesome.</p>

<p> Click <a href="https://www.itu.edu.tr/"> here</a> for something
even more awesome.</p>

</body>
</html>

<body>

<h2> 537 Test Web Page</h2>

<p> Test web page: simply awesome.</p>

<p> Click <a href="https://www.itu.edu.tr/"> here</a> for something
even more awesome.</p>

</body>
</html>

Header: HTTP/1.1 202 Accepted
Header: Server: awselb/2.0
Header: Date: Thu, 16 May 2024 12:36:11 GMT
Length = 0
Header: Content-Length: 0
Header: Connection: keep-alive
Header: x-amzn-waf-action: challenge
Header: Cache-Control: no-store, max-age=0
Header: Content-Type: text/html; charset=UTF-8
Header: Access-Control-Allow-Origin: *
Header: Access-Control-Max-Age: 86400
Header: Access-Control-Allow-Methods: OPTIONS,GET,POST
Header: HTTP/1.1 202 Accepted
Header: Server: awselb/2.0
Header: Date: Thu, 16 May 2024 12:36:11 GMT
Length = 0
Header: Content-Length: 0
Header: Connection: keep-alive
Header: x-amzn-waf-action: challenge
Header: Cache-Control: no-store, max-age=0
Header: Content-Type: text/html; charset=UTF-8
Header: Access-Control-Allow-Origin: *
Header: Access-Control-Max-Age: 86400
Header: Access-Control-Allow-Methods: OPTIONS,GET,POST

```

**Figure 0.4:** Results of concurrent requests



Here, as you can see, we sent all the requests and they are handled well with our multi-threaded approach. It can also be seen that even though we read first `www.cs.wisc.edu` `80 /` in the `url.txt` it handled the two `localhost 5003 /home.html` requests first. This is due to concurrency and it can be a proof that we managed to do it concurrently with multi-threaded fashion.

## **0.4 Conclusions**

In summary, the integration of multi-threading capabilities into both the HTTP client and server gives a significant efficiency enhancement, scalability, and responsiveness where multiple requests come to the server. This project demonstrates how multi-threading can be achieved in HTTP servers and clients and how to handle multiple requests concurrently by multi-threading.