

BLG312E

Assignment-3

Multi-threaded Web Server Implementation

Nurselen Akın
ITU, Department of Computer Engineering
150200087

Abstract—This report details the implementation of a multi-threaded web server. The web server uses a pool of worker threads to process several HTTP requests at once. In order to assess the server's performance while it is being used by several threads, the client is altered to send queries to it concurrently.

I. INTRODUCTION

In this project, I implement a multi-threaded web server to efficiently handle concurrent HTTP requests. This project is using the simplest form of handling multiple http requests. For each incoming request, new thread is created by server. Furthermore, a multi-threaded client is designed to send simultaneous requests to the server and measure its performance.

II. IMPLEMENTATION

In this part, I will provide the implementation details for this project by explaining every meaningful part of the code.

A. Initializing

active_connections: Keeps track of the number of active connections being processed.

lock: A mutex to protect shared resources such as the connection queue.

available_slots and **used_slots:** Semaphores for managing the connection queue, ensuring threads wait for available slots before adding new connections and signal when a connection is added.

worker_params_t: A structure that holds parameters for each worker thread, including a pointer to the connection queue and a unique thread ID.

B. Functions

1) *get_args()*: The port number, the number of worker threads, and the size of the connection queue are obtained by the *getargs* function parsing the command-line arguments. It checks to make sure the right number of arguments are supplied.

2) *setup_worker_threads()*: The worker threads that will handle incoming connections are created and initialized by the *setup_worker_threads* function. A reference to the connection queue and a distinct thread ID are given to each worker thread. I started these threads and assign them to the *process_requests* function using the *pthread_create* function. As soon as a new connection is introduced to the queue, this method makes sure the worker threads are prepared to handle it.

3) *accept_connections()*: In order to continually accept incoming client connections using the *Accept* function that is provided in the header file, the *accept_connections* function executes in an infinite loop. The shared connection queue is expanded by each approved connection descriptor. I utilized mutexes and semaphores (*available_slots* and *used_slots*) to control concurrent access to this queue. The semaphores make sure that threads indicate when a connection is added and wait for open spaces in the queue before adding new connections.

4) *process_requests*: Each worker thread's protocol for handling incoming connections is specified by the *process_requests* function. Using the *retrieve_connection* function, it continuously pulls connection descriptors from the queue, processing them with *handle_request*. This function retrieves and processes connections from the queue frequently, ensuring that each worker thread can handle many connections quickly.

5) *retrieve_connection*: A connection descriptor is retrieved for processing from the connection queue by the *retrieve_connection* function. It makes sure that connections are handled thread-safe by using mutexes and semaphores to securely control queue access. This function fetches the connection descriptor, waits for a connection to become available in the queue, and then indicates that a slot in the queue is now accessible.

6) *main()*: It sets up the server by allocating memory for the connection queue, initializing the required synchronization primitives, and executing *getargs* to parse arguments. After that, it makes calls to *accept_connections* to begin accept-

ing client connections and `setup_worker_threads` to generate worker threads. Cleanup is included in the main function to delete semaphores and free up allocated memory. This process configures and starts the server, making sure it is prepared to effectively handle incoming connections.

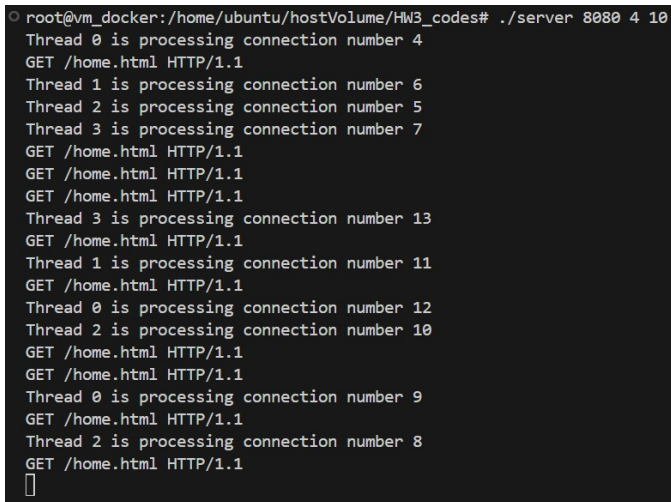
III. CONCLUSION

I developed a multi-threaded web server for this project in order to effectively manage several simultaneous HTTP requests. The server processes incoming connections by using a pool of worker threads. Mutexes and semaphores, two synchronization primitives, control secure access to shared resources like the connection queue.

By handling multiple connections at once, the server may handle multiple threads, which enhances responsiveness and performance. Race situations are avoided and resource integrity is preserved when synchronization techniques are used to guarantee that the system functions properly under heavy load.

To run the code:

`./server ;port_num; ;thread_num; ;queue_size;`

A terminal window showing the output of a multi-threaded web server. The prompt is 'root@vm_docker:/home/ubuntu/hostVolume/HW3_codes#'. The command executed is './server 8080 4 10'. The output shows multiple threads (0, 1, 2, 3) processing connections (4, 6, 5, 7, 13, 11, 12, 10, 9, 8) and receiving 'GET /home.html HTTP/1.1' requests. The threads are interleaved, demonstrating concurrent processing. The output ends with a cursor on a new line.

```
root@vm_docker:/home/ubuntu/hostVolume/HW3_codes# ./server 8080 4 10
Thread 0 is processing connection number 4
GET /home.html HTTP/1.1
Thread 1 is processing connection number 6
Thread 2 is processing connection number 5
Thread 3 is processing connection number 7
GET /home.html HTTP/1.1
GET /home.html HTTP/1.1
GET /home.html HTTP/1.1
Thread 3 is processing connection number 13
GET /home.html HTTP/1.1
Thread 1 is processing connection number 11
GET /home.html HTTP/1.1
Thread 0 is processing connection number 12
Thread 2 is processing connection number 10
GET /home.html HTTP/1.1
GET /home.html HTTP/1.1
Thread 0 is processing connection number 9
GET /home.html HTTP/1.1
Thread 2 is processing connection number 8
GET /home.html HTTP/1.1
█
```

Fig. 1. Example output recieved with using multi-threaded client