Implementing Multithreaded Web Server Using Semaphore in C

Metin Ertekin Küçük

Computer Engineering
Istanbul Technical Universtiy
Istanbul, Maslak
150210061
ITU-mail: kucukm21@itu.edu.tr

***Abstract*—This report details the design and implementation of a multi-threaded web server developed as part of Homework 3 for the BLG312E Computer Operating Systems course. The project enhances a basic single-threaded web server by introducing multi-threading to improve performance and handle multiple client requests concurrently.**

I. Introduction

Single-threaded web servers suffer from significant performance limitations as they handle only one client request at a time. This can lead to long wait times for clients, especially under high load or when serving long-running requests. The objective of this project is to design and implement a multi-threaded web server that can process multiple client requests concurrently. The server will use a thread pool and a bounded buffer to manage incoming connections and employ synchronization mechanisms to ensure thread-safe operations.

II. Implementation

The implementation consists of four main components: `server.c`, `request.c`, `client.c`, `blg312e.c`.

A. server.c

The primary goal of the implementation is to create a robust and efficient web server capable of handling multiple concurrent connections. This involves:

- Manages the main server logic, including setting up the server socket and accepting incoming connections.
- Implements buffer management using semaphores to control access to the buffer.
- Creates worker threads to handle requests asynchronously.
- Setting up a server socket to listen for incoming connections.
- Managing a buffer to queue incoming requests.
- Creating worker threads to process requests from the buffer.
- Implementing request handling logic to serve static and dynamic content.

B. request.c

- Implements request handling logic, including parsing HTTP requests and serving static or dynamic content.
- Provides functions for reading request headers, parsing URIs, and serving files.
- Could be used for debugging and testing multi-threaded running capabilities of server.

C. client.c

- Can be used for testing and running server code and it could take some filename parameter to test server code.

D. blg312e.c

- Contains wrapper functions to efficiently handle robust input output operations.

The implementation begins with parsing command-line arguments to get the port number, number of threads, and buffer size.

The server uses a circular buffer to store incoming connections, protected by semaphores and mutexes for synchronization.

Worker threads continuously fetch connections from the buffer and process them.

The main function sets up the server, initializes synchronization primitives, and starts the worker threads.

III. Design and Outputs

A. Design Overview

The multi-threaded web server comprises a main thread that listens for incoming connections and a pool of worker threads that handle these connections. The design involves:

- A fixed-size connection buffer implemented as a circular queue.
- Semaphores to manage the availability of buffer slots.
- Mutexes to protect shared data structures and ensure thread safety.
- A master thread to accept new connections and distribute them to worker threads via the buffer.

```c
void push_to_buffer(int connfd) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    buffer[buffer_index_in] = connfd;
    buffer_index_in = (buffer_index_in + 1) % buffer_size;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}

int pop_from_buffer() {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    int connfd = buffer[buffer_index_out];
    buffer_index_out = (buffer_index_out + 1) % buffer_size;

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);

    return connfd;
}
```

Fig. 1. Buffer implementation as a circular queue

- Worker threads to process HTTP requests and send responses.

```c
int main(int argc, char *argv[]) {
        exit(1);
    }

    thread_pool = (pthread_t *)malloc(sizeof(pthread_t) * num_threads);
    if (thread_pool == NULL) {
        fprintf(stderr, "Error: malloc failed\n");
        exit(1);
    }

    for (int i = 0; i < num_threads; i++) {
        if (pthread_create(&thread_pool[i], NULL, client_handler, NULL) != 0) {
            fprintf(stderr, "Error: pthread_create failed\n");
            exit(1);
        }
    }
```

Fig. 2. Thread creation

B. Outputs

The server implementation has been tested and produces the following outputs:

- Successful handling of incoming HTTP requests.

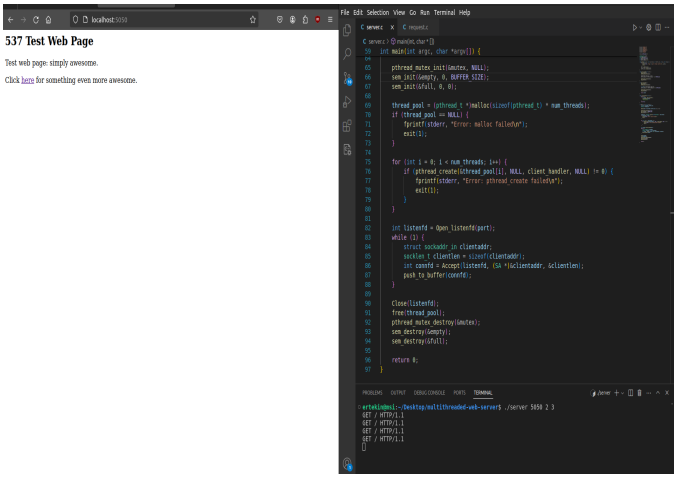

Fig. 3. Handling of incoming HTTP requests

- Buffer management to handle multiple concurrent requests.
  This achieved by adding some debugger print lines into the request.c file and we have seen that thread creation and management handled correctly.

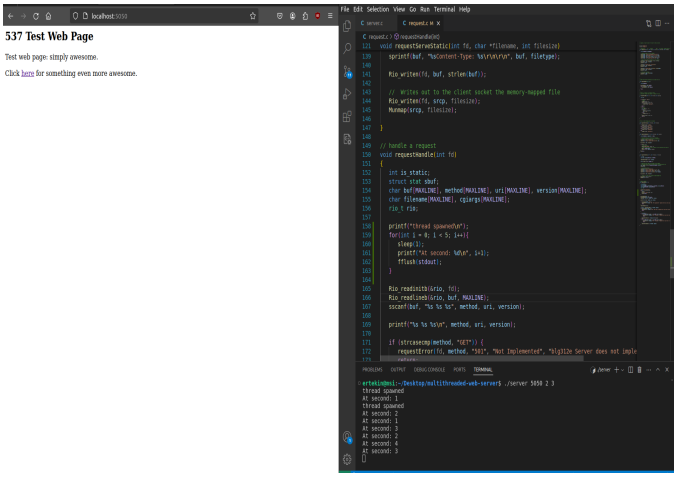

Fig. 4. Multiple concurrent requests

IV. Results

The multi-threaded web server was tested with multiple concurrent client connections. The server successfully handled multiple requests simultaneously, demonstrating improved performance and reduced client wait times compared to the single-threaded version. The synchronization mechanisms ensured thread-safe operations without any data races or deadlocks. By using a thread pool and a synchronized connection buffer, the server improved its performance and scalability.

References

[Codevault Youtube] https://www.youtube.com/watch?v=YSn8_XdGH7c/