BLG312E
Computer Operating Systems
Project Report
"Multi Threaded Web Server"

Mustafa Can Çalışkan
Faculty of Computer and Informatics Engineering
Istanbul Technical University
Istanbul, Turkey
Email: caliskanmu20@itu.edu.tr

***Abstract*—I enhanced a basic single-threaded web server to a more efficient, multi-threaded version. The new implementation uses a thread pool to handle multiple HTTP requests simultaneously and employs semaphores for synchronization. Different scheduling policies (First-In-First-Out (FIFO), Recent File First (RFF), and Smallest File First (SFF)) were also implemented to prioritize requests. These changes improve the server's performance and response time, especially under high load conditions. Command line parameters allow configuration of the number of threads, buffer size, and scheduling policy.**

I. Introduction

Web servers are essential for delivering web content, and their performance is critical under high traffic. Traditional single-threaded web servers handle only one HTTP request at a time, leading to inefficiencies and performance bottlenecks. This project enhances a basic single-threaded web server by introducing multi-threading with a thread pool, allowing simultaneous request handling and improving performance. Different scheduling policies (First-In-First-Out (FIFO), Recent File First (RFF), and Smallest File First (SFF)) were implemented to prioritize requests. Semaphores ensure proper synchronization between threads. The server now accepts command line parameters for configuring thread count, buffer size, and scheduling policy. This report details the design, implementation, and performance improvements of the multi-threaded web server.

II. Design and Implementation

This section details the design and implementation phases of the project. The overall architecture, the creation and use of the thread pool, synchronization mechanisms, and the implementation of different scheduling policies will be discussed.

A. System Architecture

The system architecture outlines the core components of the web server and their interactions. In the project, I started with a basic single-threaded web server and enhanced it by introducing a multi-threaded approach to improve performance and efficiency. The primary components include the server socket, the thread pool, the request queue, and the scheduling mechanism.

- **Server Socket:** Listens for incoming HTTP requests and establishes connections.
- **Thread Pool:** Manages a pool of worker threads to handle multiple requests simultaneously.
- **Request Queue:** Stores incoming requests until they can be processed by a worker thread.
- **Scheduling Mechanism:** Determines the order in which requests are processed based on predefined policies.

B. Thread Pool Implementation

The thread pool is a crucial component that enables concurrent processing of HTTP requests. I implemented the thread pool to maintain a fixed number of threads, which are created at the server startup and reused throughout the server's lifetime. This approach reduces the overhead of creating and destroying threads for each request.

- **Thread Creation:** Threads are created during server initialization using `pthread_create`.
- **Worker Threads:** Each thread runs a loop, waiting for tasks from the request queue.
- **Task Execution:** When a request is dequeued, the thread processes the request and sends the response back to the client.
- **Reuse:** After completing a task, the thread returns to the pool, ready to handle another request.

C. Synchronization Mechanisms

To ensure proper synchronization between threads and prevent race conditions, I used semaphores and mutex locks.

- **Mutex Locks:** Used to protect shared resources, such as the request queue, from concurrent access. This ensures that only one thread can modify the queue at a time.
- **Semaphores:** Two semaphores, `empty_slots` and `filled_slots`, were used to manage the request queue. `empty_slots` tracks the number of available

slots in the queue, while `filled_slots` tracks the number of requests waiting to be processed.

D. Scheduling Policies

Different scheduling policies were implemented to prioritize requests based on various criteria. The policies include First-In-First-Out (FIFO), Recent File First (RFF), and Smallest File First (SFF).

- **FIFO (First-In-First-Out):** Requests are processed in the order they arrive. This simple approach ensures fairness but may not always be the most efficient.
- **RFF (Recent File First):** Requests for recently modified files are prioritized. This policy aims to serve the most up-to-date content quickly.
- **SFF (Smallest File First):** Smaller files are processed before larger ones. This can improve response times for simple requests, enhancing overall performance.

The implementation of these policies involved modifying the request queue to insert new requests based on the selected policy. The policy is specified as a command-line parameter when starting the server, providing flexibility in how the server handles incoming requests.

By incorporating these design elements, the multi-threaded web server achieved significant improvements in handling concurrent requests, reducing response times, and efficiently managing server resources.

III. Testing and Results

In this section, I present the testing procedures and results for the multi-threaded web server. The main objective of the tests was to verify the correct implementation of multi-threading, semaphore, and lock mechanisms. The server was configured to listen on port 5003, create 8 worker threads for handling HTTP requests, allocate 16 buffers for connections, and use various scheduling policies.

A. Test Functions

Several test functions were implemented in the `test.c` file to assess the server's performance and correctness under different conditions:

- **basic_functionality_test:** This function tests the basic functionality of the server by sending a single HTTP GET request for the file `home.html`. It verifies that the server correctly processes the request and returns the expected response.
- **load_testing:** This function simulates a moderate load on the server by sending 50 concurrent HTTP GET requests for the file `home.html`. It checks how well the server handles multiple simultaneous connections.
- **stress_testing:** This function applies extreme load conditions by sending 100 concurrent HTTP GET requests for the file `home.html`. It is designed to test the server's robustness and its ability to manage a high volume of requests efficiently.

The `test.c` file was compiled and executed as a separate process while the server was running. This ensured that the server's multithreading, semaphore, and lock mechanisms were tested under real-world conditions. The test file was compiled and executed using the commands `gcc -o test test.c -lpthread` and `./test`.

B. Results

All tests were conducted with the server running under various scheduling policies, including FIFO, RFF, and SFF. The results are summarized below:

- For the **basic_functionality_test**, the server successfully processed the single GET request and returned the correct response for `home.html`.
- During the **load_testing**, the server efficiently handled 50 concurrent GET requests, demonstrating its capability to manage moderate traffic without issues.
- Under the **stress_testing**, the server was able to process 100 concurrent GET requests successfully, indicating strong robustness and effective use of multithreading, semaphore, and lock mechanisms.

In all tests, the server returned the expected response `"GET /home.html HTTP/1.1"` successfully. These results confirm that the server's multithreading, semaphore, and locking implementations are functioning correctly across different load conditions and scheduling policies.

IV. Conclusion

The project successfully enhanced the basic single-threaded web server to a more efficient multi-threaded version. By utilizing a thread pool and implementing scheduling policies such as FIFO, RFF, and SFF, the server can handle multiple HTTP requests simultaneously and prioritize them effectively. The use of semaphores ensured proper synchronization between threads, improving overall performance and response time under high load conditions. The added flexibility of command line configuration for threads, buffer size, and scheduling policy further enhances the server's usability and adaptability. These enhancements demonstrate significant improvements in the server's capability to handle concurrent requests efficiently.