Operating Systems

Fall 2025

Due on 2 Jan 2026 at 23:59

by Asst. Prof. Tankut Akgül

Multithreaded Webserver

1. Objective

In this multi-phase project, you will design, implement, and evaluate a simplified HTTP web server using low-level Unix system calls. You will begin with a minimal single-threaded server, then extend it to support a thread pool, a bounded request queue, and custom scheduling. Finally, you will conduct a performance evaluation and produce a report analyzing the results.

2. General Requirements

2.1 The project should be done by a group of 2-3 students.

2.2 The project phases must be distributed among students. Which student is responsible for which project phase must be clearly documented in the project report.

2.3 In your report, you must include a diary of problems you faced while implementing the code, how you resolved them and add screenshots of your debugging sessions as a proof.

2.4 Webserver should be written in C and should not use any external http libraries like libcurl.

3. Implementation Details

Phase 1 — Single-Threaded HTTP Server

Goal: Build a minimal, correct webserver that handles one request at a time.

Requirements

1. Create a TCP listening socket (using socket, bind, listen, accept).

2. Handle simple **GET /path** requests and extract the requested filename. See below websites for more information:
   https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages
   https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/GET
3. Serve extracted file from the current directory.
4. Return properly formatted HTTP/1.0 responses, including:
   - status line (200 OK or 404 Not Found)
   - headers (Content-Length, Content-Type)
   - file contents
5. Support large file streaming (read in fixed-size chunks).
6. The server may process only one client at a time in this phase.

Phase 2 — Thread Pool + Bounded Request Queue

Goal: Introduce concurrency via pthreads and implement the producer–consumer pattern.

Requirements

1. Create a fixed-size **thread pool** at server startup.
2. Implement a **bounded request queue** to hold client sockets:
   - Main thread acts as **producer**, pushing client requests
   - Worker threads act as **consumers**, popping requests in FIFO order.
3. Use a mutex and two condition variables to enforce correctness (not_empty, not_full).
4. Ensure your implementation avoids:
   - deadlocks
   - race conditions
   - busy waiting

Phase 3 — Scheduling Policy + Static File Serving Improvements

Goal: Add a scheduling policy and enhance file handling.

Requirements

1. Implement **Smallest File First (SFF)** scheduling. The server serves the requests from the queue in order of increasing file sizes instead of FIFO order.
2. Add logging for each served request showing:
    - arrival time and a monotonically incrementing sequence count for each request
    - worker thread ID that is serving the request
    - file size that is being served by the worker thread
    - request sequence count that is being served by the worker thread

Example log:

REQUEST seq=1998 path="large.html" time=2025-12-06T13:47:27.937
REQUEST seq=1999 path="small.html" time=2025-12-06T13:47:27.937
WORKER 1 picked request with seq=1998 size=10220325
WORKER 2 picked request with seq=1999 size=9006

Phase 4 — Performance Evaluation and Report

Goal: Benchmark your server and understand system-level effects.

Requirements

1. Use the provided benchmarking script to measure:
    - throughput (requests/sec)
    - latency
    - effect of varying thread count
    - effect of queue size
    - comparison of scheduling policies
2. Produce graphs or tables summarizing results.

Report Content

- Team members and their responsibilities
- Experimental setup (hardware, OS, tools used) how to compile and run your code
- Methodology
- Debugging diary with screenshots
- Results with graphs/tables
- Interpretation (why results differ, bottlenecks observed)
- Reflection on concurrency, scheduling, and server design
- **Any AI usage must be documented**

You should use the included report template.

4. Deliverables

- **Source Code (your code should be properly commented)**
- **Assignment Report (PDF File)**

Zip your source code and report file and submit a single .zip file.

5. Benchmark Program

Provided with this project is a benchmark program that you must use to measure the performance of your webserver. Also, provided with the project are sample test HTML pages in differing sizes (**do not try to directly open large.html on your browser as it may crash your browser due to its size**). The benchmark program is a BASH shell script that uses curl utility. If not already installed, you may follow the below command to install it:

Linux (Ubuntu): sudo apt install curl

Mac OS: brew install curl

Make sure bench.sh has executable permissions using the following command:

chmod +x bench.sh

Place the sample HTML pages on the same folder as your webserver. Assuming the server is running on the same machine as the machine you are running bench.sh and the server is listening on port 8080, you can use the following command to test the server:

./bench.sh 10000 100 http://localhost:8080/ small.html medium.html large.html

- 10000 is the number of times each file will be requested from the server.
- 100 is the number of parallel requests to the server.
- http://localhost:8080 is the URI and port to the server (listening on the same machine on port 8080). If the server is running on a different machine with IP address, say, 192.168.1.1, use http://192.168.1.1:8080 instead.
- The rest of the arguments are the name of the files to fetch from the server.

The script will run and will produce a report similar to below:

Generating workload of 10000 requests...
Testing server at: http://localhost:8080/ (port 8080)
Parallelism: 100
Starting...
==
BENCHMARK RESULTS
==
URL: http://localhost:8080/content
Port: 8080
Requests: 10000
Parallelism: 10

Average Wait: 0.002064s
Average Latency: 0.002610s
Min Latency: 0.000291s
Max Latency: 0.018360s

Throughput: 666.6666666666666666 req/sec
Wall Time: 15.000000s
==