

Concurrent Web Server with Scheduling Policies

Cooper Braun - CPSC 346

Problem & Concept

My project explores request scheduling in a concurrent web server, based on the OSTEP concurrency web server project. Reimplemented in Go using mutexes and condition variables (rather than channels) to really mirror the C implementation and what we learned in class.

It addresses the convoy effect. In First-Come-First-Served (FCFS) scheduling, short requests become stuck behind long-running requests, causing poor average response times. In this project, I compared two scheduling algorithms:

FCFS: Processes requests in arrival order.

SFF (Smallest File First): Prioritizes smaller files, approximating Shortest Job First.

Implementation

Architecture

The server follows a producer-consumer architecture with a bounded buffer. A master thread accepts connections and enqueues requests; a configurable pool of worker threads dequeues and handles requests. The scheduler manages the buffer using a mutex and two condition variables (notEmpty, notFull) for synchronization without busy-waiting.

Scheduling Implementation

FCFS dequeues from the front of the buffer. SFF scans the buffer to find the smallest file (determined via `os.Stat()`) and dequeues that request. An artificial delay proportional to file size ($\text{size} / 10000 \text{ ms}$) simulates disk I/O to make scheduling effects observable.

Results & Measurements

Test Configuration

Server: 2 worker threads, buffer size of 5

Workload: 5 large file requests (594 KB, ~59ms processing) followed by 10 small file requests (205 bytes, ~0.02ms processing)

Client: Burst of 15 concurrent requests with 50ms stagger between large and small files

Performance Comparison

Metric	FCFS	SFF
Small Files Average	73.7 ms	11.6 ms
Big Files Average	113.8 ms	112.0 ms
Improvement	—	6.4x faster

Key Insights

1. The Convoy Effect is Measurable: Under FCFS, small files averaged 73.7 ms despite requiring only 0.02 ms to process, a 3,685x slowdown caused by waiting behind large requests.

2. SFF Dramatically Improves Small Request Latency: Prioritizing small files reduced response time by 6.4x (73.7 ms → 11.6 ms) while leaving large file performance pretty much unchanged (113.8 ms → 112.0 ms), demonstrating that intelligent scheduling benefits short requests without starving large ones.

3. Scheduling Policy Matters for Mixed Workloads: When workloads vary significantly in size, priority-based scheduling substantially improves average response time and user experience.

Conclusion

This implementation pretty successfully demonstrates some different concepts like concurrency control, producer-consumer synchronization, and request scheduling. The results clearly show the convoy effect under FCFS and the substantial benefits of SFF for mixed workloads. Using Go with low-level synchronization primitives provided me with some good experience with using mutexes, condition variables, and thread pool architecture while still remaining faithful to the original C project I was kind of basing this off of.