universidade de aveiro

deti
departamento de eletrónica,
telecomunicações e informática

Technical Report

Multi-Threaded Web Server with IPC and Semaphores

Operating Systems

Academic Year 2024/2025

Martim Gil Student ID: 124833

Nuno Leite Faria Student ID: 112994

Supervisor: Prof. Pedro Azevedo Fernandes

December 8, 2025

Abstract

This document outlines the design, implementation, and evaluation of a concurrent HTTP server built in C for the Operating Systems course at the University of Aveiro. The primary objective was to develop a system capable of effectively managing numerous client requests simultaneously using a hybrid process–thread architecture.

The system incorporates sophisticated synchronization and interprocess communication methods, specifically POSIX semaphores and shared memory, guaranteeing consistency and mutual exclusion during access to shared resources. Additional modules were implemented for safe concurrent logging, LRU-based file caching, and real-time statistics monitoring.

The final solution underwent extensive functional, concurrency, and stress testing. The results demonstrate the reliability, scalability, and correctness of the synchronization mechanisms, as well as the system's ability to maintain consistent performance under high workloads.

Contents

1 Introduction

Creating concurrent servers is among the most effective and demonstrative uses of synchronization, inter-process communication, and thread management concepts explored in operating systems courses. This project focused on creating and deploying a straightforward yet robust HTTP server that can handle multiple concurrent client requests simultaneously.

The C programming language was used for the system's implementation, utilizing POSIX system calls for thread creation and synchronization, semaphore management, shared memory management, and inter-process communication. The structure adopts a modular design to guarantee the maintainability and extensibility of various system elements.

2 Implementation Details

The concurrent web server was implemented entirely in C using POSIX system calls for process management, thread creation, synchronization, and interprocess communication (IPC). The implementation follows a modular architecture where each subsystem corresponds to a dedicated source file in the `src/` directory. This structure promotes reusability, maintainability, and clear separation of concerns between the networking layer, synchronization mechanisms, and data management components.

2.1 System Architecture

At a high level, the project consists of three main execution entities: the **master process**, the **worker threads**, and the **auxiliary tools**. The master process is responsible for system initialization, socket setup, configuration parsing, and coordination of shared resources. Each worker thread, managed by a thread pool, handles an individual client connection. Auxiliary programs, such as `stats_reader`, access the server's shared memory region to retrieve runtime statistics without interfering with normal operation.

The core modules include:

- **master.c** – Initializes the system, loads configuration, and manages the main listening loop.
- **worker.c** / **worker.h** – Implements the logic executed by each worker thread.
- **thread_pool.c** / **.h** – Manages the pool of worker threads and the task queue.
- **cache.c** / **.h** – Provides an in-memory LRU file cache.
- **logger.c** / **.h** – Implements thread-safe logging.
- **stats.c** / **.h** – Manages global statistics in shared memory.
- **semaphores.c** / **.h** – Encapsulates POSIX semaphore operations.
- **http_parser.c** / **http_builder.c** – Handle HTTP protocol parsing and response construction.

2.2 Process and Thread Management

The concurrent HTTP server follows a hybrid process–thread architecture. The `master` process is responsible for system initialization and the creation of a fixed-size thread pool. This approach avoids the overhead associated with creating and destroying threads on demand while allowing high levels of concurrency.

Once initialized, the `master` process accepts incoming TCP connections and submits them to the thread pool. The thread pool, implemented in `src/thread_pool.c`, maintains a synchronized queue of tasks. Worker threads wait on a condition variable; when a new connection is enqueued, a thread is signaled to wake up and process the request.

2.3 Synchronization and IPC

To guarantee secure operation in a multithreaded setting, the system uses POSIX semaphores, mutexes, and shared memory.

Producer–Consumer Coordination. The interaction between the master and worker threads follows a producer–consumer model. The master produces tasks (socket descriptors), and workers consume them. Access to the task queue is controlled by a counting semaphore (available tasks) and a mutex (queue integrity).

Shared Statistics. Global runtime metrics (requests, bytes sent, cache hits) are stored in POSIX shared memory. Updates to these counters are protected by binary semaphores to ensure atomicity, preventing race conditions where multiple threads could try to increment the same counter simultaneously.

Shared Logging. Logging operations utilize a named binary semaphore to ensure mutual exclusion. This prevents "interleaved" log lines where output from multiple threads would otherwise be mixed together in the file.

2.4 Resource Management (Cache)

To minimize disk I/O, the server incorporates an LRU (Least Recently Used) in-memory cache (`src/cache.c`). The cache uses **POSIX reader–writer locks** (`pthread_rwlock_t`) to maximize concurrency. Multiple threads can read from the cache simultaneously (read lock), but writing to the cache (loading a new file) requires exclusive access (write lock). This significantly improves performance under high read loads compared to a simple mutex.

2.5 HTTP Processing

The HTTP processing layer is split between parsing and response generation:

3 Challenges and Solutions

The development of a concurrent HTTP server presented challenges related to synchronization, communication, and system scalability.

3.1 Race Conditions

During early development, race conditions in the shared statistics subsystem caused lost counts. The solution was the introduction of an **exclusive binary semaphore** to protect the shared `stats` structure. This ensures that only one thread can modify the statistics at any given time, guaranteeing atomicity. Verification through stress tests confirmed that the reported request counts matched the actual number of served requests.

3.2 Memory Management and Leaks

Memory management was critical for stability. **Valgrind** was used extensively to detect leaks. Issues with unreleased cache buffers and unclosed file descriptors were corrected. To ensure shared memory and semaphores are cleaned up even after a forced termination (`Ctrl+C`), the master process installs custom signal handlers for `SIGINT` and `SIGTERM`. These handlers invoke a cleanup routine that unlinks all semaphores and unmaps shared memory, preventing orphaned resources in the OS.

3.3 Log Synchronization

Concurrent writing to the log file initially resulted in interleaved/garbled text. The solution involved using a **named binary semaphore** for the log file and implementing thread-local buffering. Threads format their complete log message into a local buffer first, then acquire the semaphore, write the entire buffer in one operation, and release the semaphore.

3.4 Zombies and Signals

To prevent zombie processes (defunct child processes), the master process handles the `SIGCHLD` signal. The handler uses `waitpid(-1, NULL, WNOHANG)` to immediately reap any terminated worker processes, keeping the process table clean.

4 Testing Methodology

Validation was performed through a structured framework combining automated functional tests and concurrency stress tests.

4.1 Functional Tests

The `tests/test_suite.sh` script verifies HTTP compliance using `curl`. It checks for correct status codes (200, 404, 500), proper MIME types, and handling of special files. All functional tests passed consistently.

4.2 Concurrency and Stress Tests

Concurrency was validated using:

- **ApacheBench (ab):** Simulating up to 100 concurrent clients to measure throughput and latency.
- **Helgrind & ThreadSanitizer:** Dynamic analysis tools used to detect data races. These tools confirmed that our synchronization primitives (mutexes, rwlocks) effectively protected shared data.
- **Stress Script:** A custom script sending random requests for extended periods to check for memory leaks or long-term instability.

5 Performance Analysis

5.1 Test Environment

5.2 Test Environment

5.3 Results: Throughput

5.4 Results: Latency

5.5 Discussion of Results

6 Conclusion