Multi-Threaded Web Server with IPC and Semaphores

SO 25/26

Technical Report

Department of Electronics, Telecommunications and Informatics

Bernardo Mota Coelho - 125059
Tiago Francisco Crespo do Vale - 125913

9/12/2025

1. Implementation details

File	Description
`cache.c`	Implements an in-memory file cache for the HTTP server, using a hash table and reader-writer locks for thread-safe access. Optimizes file serving by storing file contents and metadata, allowing concurrent reads and exclusive writes.
`config.c`	Handles server configuration management. Loads settings from a configuration file, applies defaults, and provides accessors for all configuration parameters (port, document root, workers, threads, queue size, log file, cache size, timeout).
`global.c`	Defines global variables and shared state used across the server, such as flags, counters, and configuration pointers. Facilitates inter-module communication and centralizes global data management.
`http.c`	Handles HTTP protocol logic, including parsing requests, building responses, managing headers, and serving files. Implements error handling and supports static file delivery from the document root.
`logger.c`	Manages server logging, including access logs and error logs. Provides thread-safe functions to write log entries, format messages, and handle log file rotation or flushing.
`main.c`	Entry point for the server application. Initializes all subsystems, parses command-line arguments, loads configuration, and starts the master process and worker threads. Coordinates server startup and shutdown.

`master.c`	Implements the master process logic, responsible for spawning and managing worker processes, handling signals, and coordinating inter-process communication. Oversees server lifecycle and resource management.
`semaphores.c`	Provides semaphore utilities for process and thread synchronization. Implements creation, destruction, and operations on POSIX semaphores to coordinate access to shared resources.
`shared_mem.c`	Manages shared memory segments for inter-process communication. Handles allocation, mapping, and cleanup of shared memory used for statistics, configuration, or other shared data.
`stats.c`	Collects and manages server statistics, such as request counts, errors, and performance metrics. Provides functions to update, retrieve, and reset statistics, supporting monitoring and reporting.
`thread_pool.c`	Implements a thread pool for efficient request handling. Manages worker threads, task queues, and synchronization primitives to process incoming connections concurrently.
`worker.c`	Contains the logic for worker processes/threads. Handles accepting connections, processing requests, interacting with the thread pool, and communicating with the master process. Executes the main request-processing loop.

2. Challenges

I. Inter-Process Communication and Synchronization:

- **Challenge:** Coordinating the Master and Worker processes using POSIX semaphores for inter-process communication (IPC).

- **Resolution Focus:** Achieving thread- and process-safety when accessing shared memory, specifically the request queue and statistics structure. Precise management of the `sem_empty` and `sem_full` semaphores was essential to correctly implement the Producer-Consumer model, thereby preventing race conditions and deadlocks.

II. Resource Management and Graceful Termination:

- **Challenge:** Implementing a reliable graceful shutdown mechanism to handle server interruptions (e.g., `SIGINT`). Forceful stops caused named semaphores and shared memory objects to persist in the operating system (`/dev/shm`), blocking server restarts.

- **Resolution Focus:** Developing a robust signal handler (`handle_sigint`) to guarantee that `shm_unlink` and `sem_unlink` were called to release resources, and that all worker processes were properly terminated using `kill()` before the Master process exited.

III. File Descriptor (FD) Leak Prevention:

- **Challenge:** Persistent file descriptor leaks, mainly related to sockets and file operations, especially following timeouts or errors. Unclosed FDs would eventually exhaust the process limit, preventing the server from accepting new connections.

- **Resolution Focus:** Ensuring absolute stability by verifying that every `open()` call had a corresponding `close()` across all execution pathways, particularly within error handling blocks.

IV. Multi-Process Debugging and Stability:

- **Challenge:** Diagnosing "Core Dumps" and segmentation faults (often stemming from improper pointer or string manipulation in `parse_request`) within the

multi-process architecture, as one crashing worker made pinpointing the source difficult.

- **Resolution Focus:** Strict and careful memory management, specifically when dealing with dynamic allocations required for the request buffers and the cache.

3. Solutions

1. Robust Inter-Process Communication (IPC) and Concurrency

- **Named Semaphores for IPC:** POSIX named semaphores (`sem_open`, `sem_wait`, `sem_post`) were used to resolve race conditions in the multi-process environment. A specific mutex semaphore (`sems.sem_stats`) protects critical sections, such as updating global statistics in shared memory, allowing only one worker to modify the data at a time.

- **High-Performance Caching with Read-Write Locks:** An in-memory, LRU-like cache was implemented to minimize disk I/O latency. To overcome the concurrency bottleneck, it utilizes `pthread_rwlock_t` instead of a standard mutex. This allows for simultaneous content reads by multiple workers/threads (`pthread_rwlock_rdlock`) while exclusively blocking access only when a new file is being written to the cache (`pthread_rwlock_wrlock`).

2. Resource Management and Stability

- **Graceful Shutdown Mechanism:** A dedicated signal handler (`handle_sigint`) in the Master process catches `SIGINT` (CTRL+C) to execute a strict cleanup routine and prevent "orphan" resources. This routine systematically:
    1. Terminates all worker processes using `kill()`.
    2. Closes the main server socket.
    3. Cleans up system resources by unlinking shared memory (`shm_unlink`) and semaphores (`sem_unlink`), ensuring the server can be restarted cleanly without "File exists" errors.

- **Mitigation of Leaks and Hanging Connections:** To prevent file descriptor leaks and "hanging" connections, socket timeouts (`SO_RCVTIMEO` and `SO_SNDTIMEO`) were configured using `setsockopt`. The HTTP handler enforces a strict cycle

where the client socket is explicitly closed (`close(client_socket)`) after every request or error, immediately returning file descriptors to the system pool.