

Concurrent HTTP Server: Design Document

Inês Batista, 124877 & Maria Quinteiro, 124996 (P3G4)

1. Introduction

This document describes the architecture and synchronization methods for the ConcurrentHTTP Server, a multi-threaded, multi-process HTTP/1.1 server created for the Operating Systems course. The objective of the project is to develop a production-level web server that can effectively manage large numbers of simultaneous client connections, replicating real-world obstacles in concurrent systems programming. The main goal is to showcase proficiency in essential operating systems principles, such as process management, thread synchronization, and inter-process communication. The major technical hurdle involves handling simultaneous access to shared resources while avoiding race conditions, deadlocks, and data corruption. This design document outlines the server's master-worker framework, the synchronization methods used to manage processes and threads, and the data structures that facilitate effective request processing. This project examines the essential principles that support contemporary, high-performance web infrastructure.

2. System Architecture

Following the high-level objectives outlined in the introduction, this section explores the specific architectural design of the ConcurrentHTTP Server. The structure is a multi-threaded, multi-process framework that purposefully divides concerns to attain stability, scalability, and effective resource usage.

2.1. High-Level Overview

The system is built upon a master-worker pattern with a shared task queue, creating a clear producer-consumer relationship between its components. The most effective way to understand the overall data flow and component interaction is through the following architectural diagram:

This diagram illustrates the main data path and core components. The Master Process acts as the producer, accepting incoming socket connections and enqueueing them into a bounded Shared Memory Queue. Multiple Worker Processes, forked by the master, act as consumers. Each worker runs a Thread Pool where individual threads (T) dequeue connection descriptors, process the HTTP requests, and generate responses. To optimize performance, each worker maintains a local LRU Cache for frequently accessed files. Crucially, all components interact with shared resources: the queue and Global Statistics reside in shared memory, while all threads and processes write to a common Log File. The entire system is coordinated by synchronization primitives to ensure data consistency and prevent race conditions. This architecture effectively decouples connection acceptance from request processing, allowing the system to scale and handle a high volume of concurrent clients efficiently.

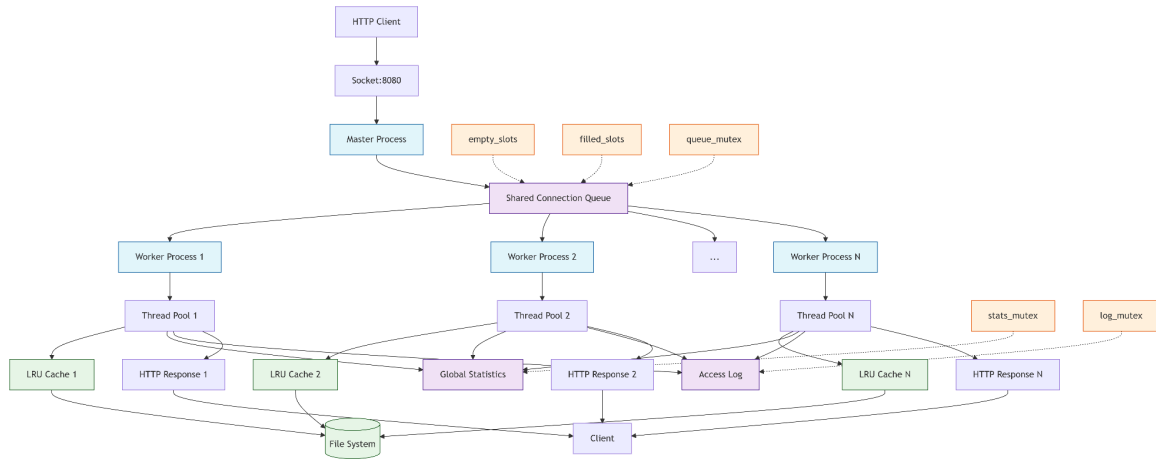


Figure 1: High-level master-worker architecture overview

3. Main Components

3.1. Master Process

The Master Process serves as the central orchestrator and entry point of the system. Its primary role begins with network initialization: creating a socket, binding it to a configurable port (default 8080), and placing it in a listening state. Once initialized, the master forks a specified number of Worker Processes and initializes the shared memory segments and semaphores required for inter-process communication. Acting as the producer in the producer-consumer pattern, the master accepts incoming connection file descriptors and places them into the bounded shared queue. It also manages the global statistics structure in shared memory and implements robust signal handling (SIGINT/SIGTERM) to ensure graceful server shutdown, coordinating the termination of all workers and cleaning up all shared resources.

3.2. Worker Process and Synchronization Mechanisms

Each Worker Process is a self-contained unit designed for HTTP request processing. Upon startup, a worker initializes a static Thread Pool, creating fixed threads that remain dormant until work is available. As a consumer, it continuously retrieves connection descriptors from the shared queue and dispatches them to available threads. Each worker maintains a thread-safe LRU File Cache that stores frequently accessed files (<1MB) in memory, significantly reducing disk I/O. Worker threads handle the core HTTP protocol: parsing requests, serving static files or generating errors, and sending responses.

The system employs a layered synchronization strategy to coordinate these components. For Inter-Process Communication (IPC), POSIX Named Semaphores manage the shared connection queue (using `empty_slots`, `filled_slots`, and `queue_mutex`) and protect global statistics (using `stats_mutex`). Within each worker, Thread-Level Synchronization uses pthread mutexes and condition variables to protect internal request queues, while pthread read-writer locks (`pthread_rwlock_t`) allow concurrent cache reads but exclusive writes. Shared Memory segments, created via `shm_open` and mapped with `mmap`, host the connection queue and statistics, enabling fast data exchange between processes. This multi-faceted approach ensures data integrity, prevents race conditions and deadlocks, and allows the system to scale effectively under concurrent load.

4. Synchronization Diagrams

4.1. Connection Queue (Producer-Consumer Pattern)

The connection queue implementation follows the classic producer-consumer paradigm using a bounded buffer in shared memory, synchronized with POSIX semaphores. The sequence is precisely coordinated to prevent race conditions and ensure efficient resource utilization.

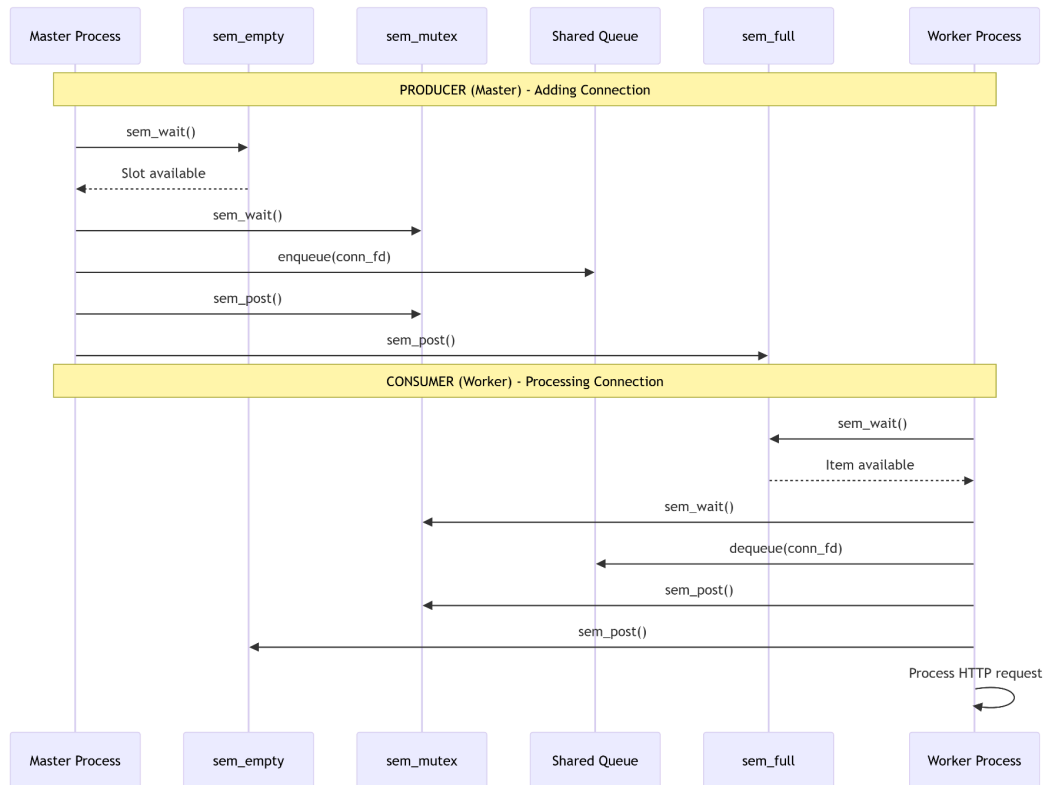


Figure 2: Producer-consumer protocol for connection queue

The Master Process (Producer) follows this sequence: First, `sem_wait(empty_slots)` ensures buffer space is available. Then `sem_wait(queue_mutex)` provides exclusive queue access for the enqueue operation. After adding the connection descriptor, `sem_post(queue_mutex)` releases the lock and `sem_post(filled_slots)` signals consumers.

The Worker Process (Consumer) operates symmetrically: `sem_wait(filled_slots)` waits for available work, `sem_wait(queue_mutex)` locks the queue for dequeue, then `sem_post(queue_mutex)` and `sem_post(empty_slots)` complete the handshake. This disciplined protocol guarantees no lost connections or corrupted queue state.

4.2. Cache Access (Read-Write Locks)

The LRU file cache employs pthread read-write locks to maximize parallelism for read operations while ensuring consistency during modifications.

For **Reader Threads**, multiple threads can simultaneously acquire read locks (`pthread_rwlock_rdlock`) and access cached data concurrently. This allows high through put

when serving popular static files. Each reader releases the lock with `pthread_rwlock_unlock` when finished.

For **Writer Threads**, an exclusive write lock (`pthread_rwlock_wrlock`) is required for cache modifications. This call blocks until all active readers release their locks, ensuring no threads are accessing the cache during structural changes. After completing the update (insertion or eviction), the writer releases the lock with `pthread_rwlock_unlock`, allowing waiting readers or writers to proceed.

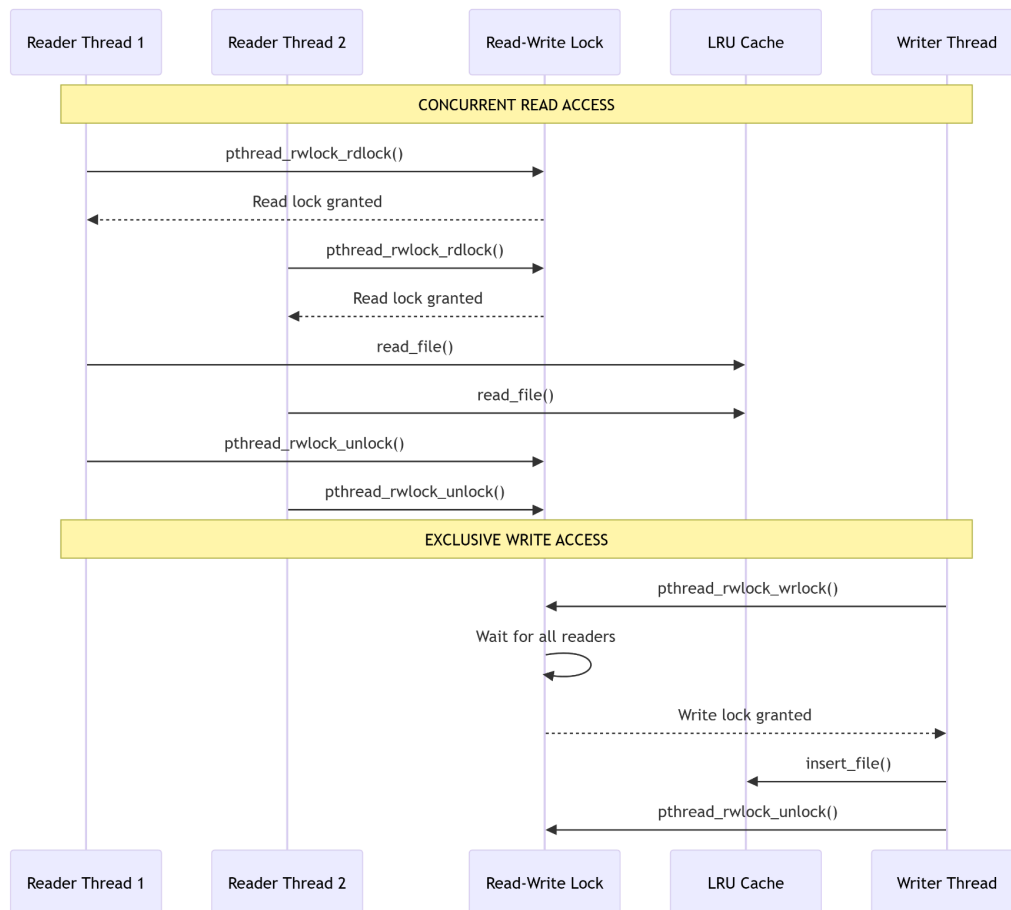


Figure 3: LRU cache concurrency control mechanism

5. Data Structures and Algorithms

5.1. Queue Structure

The connection queue is implemented as a bounded circular buffer in shared memory, allowing efficient producer-consumer communication between the master and worker processes. The data structure includes synchronization primitives to ensure thread-safe operations.

The queue operations follow the circular buffer algorithm with modulo arithmetic for index management. The three semaphores implement the classic solution to the producer-consumer problem, preventing buffer overflow, underflow, and race conditions during enqueue and dequeue operations.

```
typedef struct {
    int conn_fd[MAX_QUEUE_SIZE]; // Circular buffer for connection descriptors
    int front;                    // Dequeue position
    int rear;                    // Enqueue position
    int count;                   // Current number of elements
    sem_t *empty;                // Semaphore counting empty slots
    sem_t *full;                 // Semaphore counting filled slots
    sem_t *mutex;                // Binary semaphore for mutual exclusion
} conn_queue_t;
```

5.2. LRU Cache Implementation

The file cache employs a combination of a doubly-linked list and hash map to achieve $O(1)$ complexity for both access and eviction operations. While the detailed structure is omitted here for brevity, the algorithm maintains files in access order: recently accessed files are moved to the head of the list, while the Least Recently Used files are evicted from the tail when the cache exceeds its 10MB limit.

5.3. Shared Memory Statistics

Global statistics are maintained in shared memory to provide real-time server metrics across all processes. Atomic updates are ensured through semaphore protection.

```
typedef struct {
    int total_requests;          // Total requests served
    int total_bytes;             // Total bytes transmitted
    int status_200, status_404; // HTTP status counters
    int status_403, status_500; // Additional error counters
    sem_t *lock;                 // Semaphore for atomic updates
} stats_t;
```

When a worker process completes a request, it follows a strict sequence: `sem_wait(stats.lock)` to acquire exclusive access, updates the relevant counters, and finally `sem_post(stats.lock)` to release the lock. This approach maintains data consistency while allowing the master process to periodically read accurate aggregate statistics.

6. Execution Flow

6.1. Server Startup and initialization

The server initialization follows a precise sequence to establish all necessary components for handling concurrent connections. The Master Process begins by reading the configuration file (`server.conf`) to obtain parameters such as port number, document root, number of workers, and thread pool sizes. Following configuration parsing, the master engages in critical system resource setup: it creates and initializes all POSIX named semaphores required for inter-process synchronization, including `empty_slots`, `filled_slots`, and `queue_mutex` for the connection queue, along with `stats_mutex` for statistics protection. Subsequently, it allocates and maps the shared memory segments that will host the global connection queue and statistics structure. With the communication infrastructure in place, the master process executes a series of `fork()` system calls to create the specified number of Worker Processes. Each child worker process inherits the open shared memory descriptors and semaphores. During their initialization phase, the workers

create their respective thread pools, where each thread is launched and immediately begins waiting on a condition variable for incoming connection descriptors. This comprehensive startup sequence ensures that when the master process finally creates the server socket, binds to the designated port, and transitions to a listening state, the entire ecosystem of workers and threads is prepared and waiting to process incoming client requests.

6.2. Request Processing Lifecycle

The lifecycle of an HTTP request is a coordinated journey through the server's concurrent architecture, as visualized in the following sequence diagram:

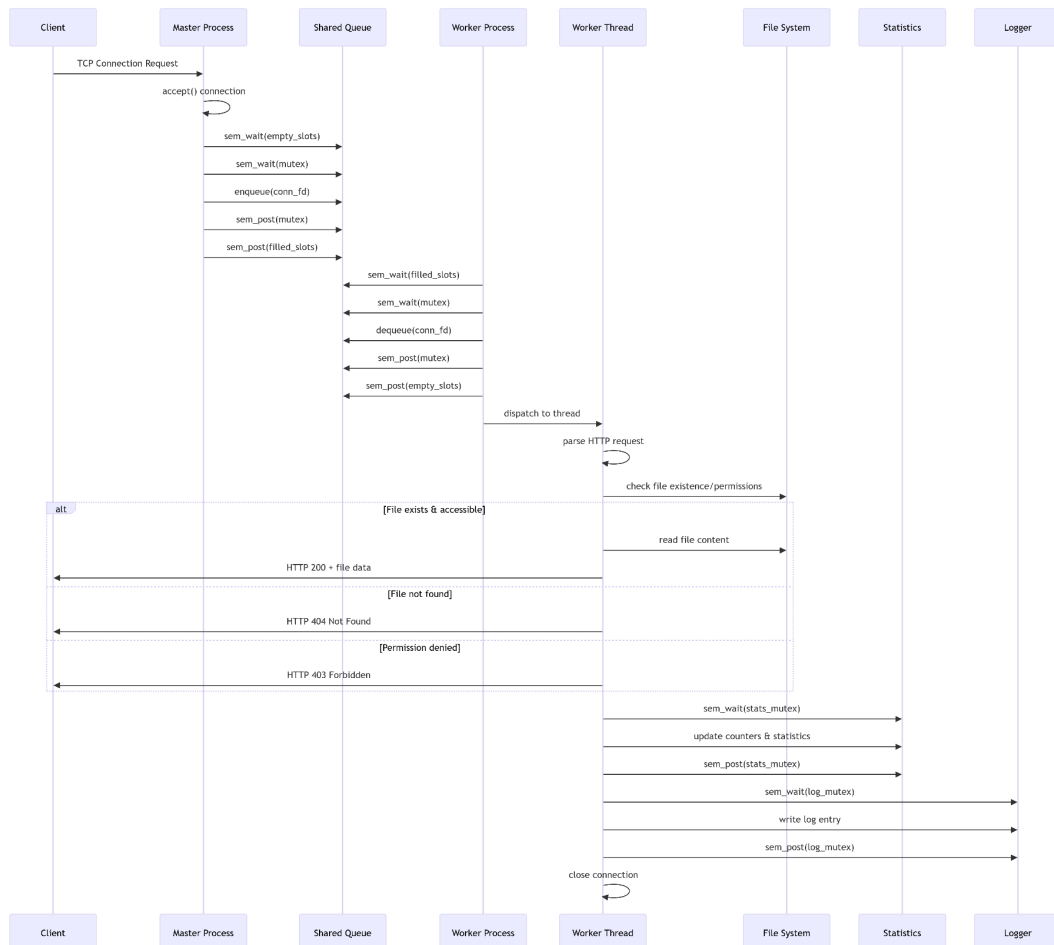


Figure 4: HTTP request processing lifecycle

7. Concurrency and synchronization analysis

The server's design requires careful handling of simultaneous access to common resources to maintain data integrity and avoid system crashes. Regions that need protection are the Shared Connection Queue, Global Statistics, LRU File Cache, and Log File. Each utilizes customized synchronization methods: the queue adopts a semaphore trio for producer-consumer management; statistics use a specific mutex for atomic counter increments; the cache implements reader-writer locks to permit simultaneous reads while ensuring exclusive writes;

and logging performs buffered writes with mutex safeguards to avoid interleaving while preserving efficiency.

To reduce concurrency risks, the design utilizes a tiered method for synchronization. Race conditions are avoided by thoroughly protecting all accesses to shared variables using suitable primitives—POSIX semaphores for inter-process coordination and read methods for thread-level synchronization. Deadlock avoidance is attained by implementing a rigid, consistent order for obtaining locks across the codebase, ensuring that the queue mutex always has priority when multiple resources need concurrent access. Starvation avoidance is achieved via FIFO-semaphore actions that ensure fair access to the connection queue for every worker process, upholding system equity during high load.

8. Performance Considerations

The design of the server includes various enhancements to enhance throughput and resource efficiency. Architectural scalability is realized via the multi-process, multi-threaded framework, which separates connection handling from request execution and facilitates efficient use of multi-core systems. Efficiency improves through caching techniques that lower disk input/output and reader-writer locks that decrease synchronization costs for typical read tasks. Buffered logging performs I/O optimization by grouping writes to minimize system calls, while graceful degradation methods maintain stability during high load by sending HTTP 503 responses to new connections when system thresholds are hit, averting failure and allowing recovery.