ConcurrentHTTP Server - Design Document

Sistemas Operativos 2025/2026 - Trabalho Prático 2

Autores: Martim Dias 125925, Nuno Costa 125120

1. Introduction

1.1 Purpose

The purpose of this document is to detail the design and architecture of the **ConcurrentHTTP Server**, a high-performance web server developed for the Operating Systems course project. The design emphasizes concurrent request handling, robust inter-process communication (IPC), and thread safety through the disciplined use of POSIX synchronization primitives.

1.2 Scope

The ConcurrentHTTP Server is a multi-process, multi-threaded application implemented entirely in C using standard POSIX APIs.

The Server Implements The following core capabilities:

- **Architecture:** Master-Worker process model, where workers utilize internal thread pools for concurrency.
- **HTTP/1.1 Protocol:** Support for GET and HEAD methods, static file serving, MIME type detection, and standard status codes.
- **Resource Management:** Thread-safe logging, shared statistics tracking, and a per-worker thread-safe LRU file cache
- **Configuration:** All runtime parameters are loaded from a server.conf file.

1.3 Overview

The ConcurrentHTTP Server utilizes a modular design, as demonstrated by the file structure. The core components are:

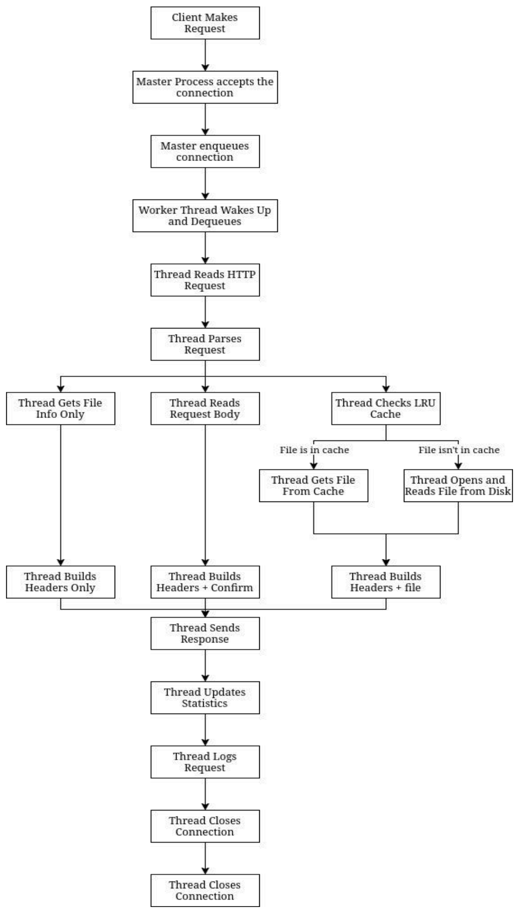
- **Master Process:** Responsible for socket setup , IPC initialization, worker process spawning, and server monitoring.

- **Worker Processes:** Each worker process handles client connections directly using a Shared Port Accept pattern, where the kernel distributes incoming connections among the workers.
- **Thread Pool:** Each worker creates a fixed number of worker threads to handle concurrent requests.
- **IPC Layer:** Manages named POSIX semaphores and a shared memory segment for statistics and logging synchronization between all processes.

---

2. System Architecture

The Concurrent HTTP Server employs a Master-Worker, Multi-Threaded Architecture designed for high throughput and fault isolation. The system leverages the Shared Port Accept pattern to distribute client connections efficiently across worker processes.



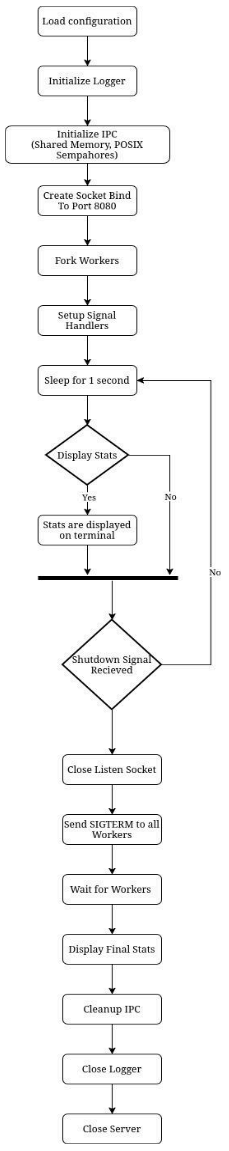
Client Makes Request
Master Process accepts the connection
Master enqueues connection
Worker Thread Wakes Up and Dequeues
Thread Reads HTTP Request
Thread Parses Request
Thread Gets File Info Only
Thread Reads Request Body
Thread Checks LRU Cache
File is in cache
File isn't in cache
Thread Gets File From Cache
Thread Opens and Reads File from Disk
Thread Builds Headers Only
Thread Builds Headers + Confirm
Thread Builds Headers + file
Thread Sends Response
Thread Updates Statistics
Thread Logs Request
Thread Closes Connection
Thread Closes Connection

1. Architecture Diagram

2.1 Master Process Responsibilities

The Master Process serves as the system supervisor and initializer. Its primary responsibilities are:

- **Initialization:** Creating and linking all named POSIX resources, including the shared memory segment (shm_open) and named semaphores (sem_open) for logging and statistics.
- **Socket Setup:** Setting up the listening socket and marking it with SO_REUSEADDR before the first fork.
- **Worker Management:** Spawning a pre-configured number of Worker Processes using fork().
- **System Monitoring:** Trapping termination signals (SIGINT and SIGTERM) and managing the graceful shutdown sequence.



2. Master Process Life cycle FlowChart

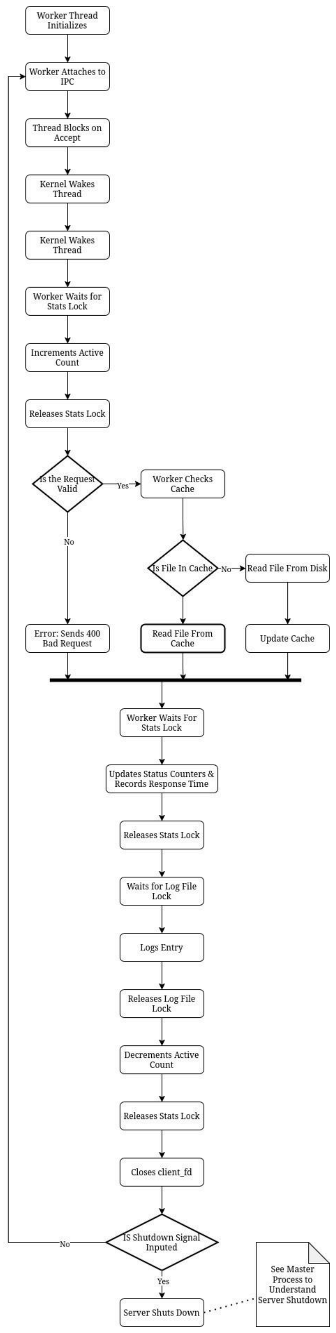
2.2 Worker Process and Thread Pool Model

The Worker Processes (worker.c) are the core execution units of the server. Each worker operates independently and manages its own concurrency through a local Thread Pool.

- **Shared Port Acceptance:** Each Worker Process inherits the listening socket file descriptor from the Master. The operating system kernel is responsible for distributing incoming client connections among the available Workers. This design choice eliminates the need for an Inter-Process Communication (IPC) connection queue, reducing IPC overhead per request.
- **Thread Pool:** Each Worker launches a fixed-size pool of dedicated Worker Threads. These threads execute worker_thread_fn, which immediately enters a blocking accept() call, waiting for a connection assigned by the kernel.
- **Intra-Process Synchronization:** Threads within a Worker share two critical data structures: the LRU File Cache and the Thread Pool Queue (if any internal queues are used). These are protected by Pthread Reader-Writer Locks and Pthread Mutexes/Condition Variables, respectively.

3. Worker Process Request Handling Flow

This flow details the complete lifecycle of a client request within a Worker Thread, from acceptance to cleanup, highlighting the strict discipline required for updating shared statistics and logs via Inter-Process Communication (IPC) semaphores.



3. Worker Process Request Handling FlowChart

The Worker Thread, upon being woken by a connection, performs the following critical sequence:

1. **Acquire Statistical Lock (sem_stats):** The thread must first acquire the sem_stats semaphore to ensure exclusive access to the shared server_stats_t structure.
2. **Increment Active Count:** The thread atomically increments the active_connections counter, allowing the Master Process to report real-time load.

3. **Release Statistical Lock:** The thread releases the sem_stats semaphore, allowing other processes/threads to update the statistics.
4. **Process Request:** The thread reads and parses the HTTP request headers.
5. **Serve Data:** The thread enters the most complex phase, involving the LRU Cache check and file serving (detailed in Section 4).
    - If the request is valid, the thread proceeds to check the cache.
    - If the request is invalid (e.g., 400 Bad Request), the thread sends the error response and bypasses the cache/disk logic, but **must** rejoin the main flow for accounting.
6. **Update Final Metrics:** After the response is sent, the thread re-acquires the sem_stats lock to update final metrics (status code counters, total bytes transferred, and response time) before releasing the lock.
7. **Log Transaction:** The thread acquires the dedicated sem_log semaphore to gain exclusive access to the shared log file. It writes the completed transaction log entry and then releases sem_log.
8. **Decrement Active Count:** The thread re-acquires sem_stats, decrements the active_connections counter, and releases the lock.
9. **Cleanup:** The thread closes the client file descriptor (client_fd) and returns to block on the accept() call, awaiting the next connection.

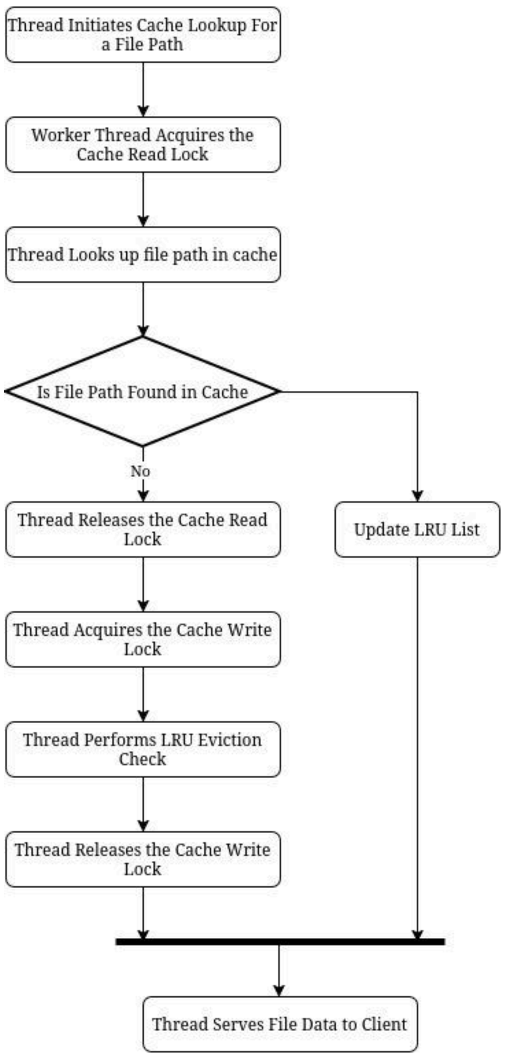
---

4. LRU Cache Synchronization Design

The LRU Cache is shared by all threads within a single Worker Process and is protected by a Pthread Reader-Writer Lock (pthread_rwlock_t) to facilitate high-speed, concurrent access.

4.1 Cache Access Flow

The cache access logic is designed to optimize for high concurrency during read operations (Cache Hits) while strictly enforcing mutual exclusion during structural modifications (Cache Misses).


Thread Initiates Cache Lookup For a File Path
Worker Thread Acquires the Cache Read Lock
Thread Looks up file path in cache
Is File Path Found in Cache
No
Thread Releases the Cache Read Lock
Update LRU List
Thread Acquires the Cache Write Lock
Thread Performs LRU Eviction Check
Thread Releases the Cache Write Lock
Thread Serves File Data to Client

4. LRU Cache Synchronization FlowChart

The thread must follow this protocol for every file lookup:

- **Acquire Read Lock:** The Worker Thread calls pthread_rwlock_rdlock() to acquire the read lock. Multiple threads may hold this lock simultaneously.
- **Cache Lookup:** The thread performs a fast lookup of the file path in the cache's hash map.
- Decision: Cache Hit (YES):
    - The file data is retrieved from memory.
    - The thread performs a localized **LRU Update** (moving the entry to the MRU position), which necessitates a brief switch from Read Lock to Write Lock (or an atomic function protected internally by a write lock).
    - The thread serves the data and continues to the cleanup phase.

- Decision: Cache Miss (NO):
    - The thread must **release the Read Lock** (pthread_rwlock_unlock()).
    - The thread then attempts to acquire the exclusive Write Lock (pthread_rwlock_wrlock()). This blocks all other readers and writers.
    - **Disk I/O:** The thread performs the slow action: reading the file data from the physical disk.
    - **Cache Update:** The thread performs the LRU eviction check and inserts the new file data into the cache (a write operation).
    - The thread releases the Write Lock and serves the data.

---

5. Graceful Shutdown and IPC Cleanup

The server ensures a clean termination to prevent resource leaks and orphaned processes.

5.1 Termination Mechanism

Signal Interception: The Master Process registers signal handlers for SIGINT and SIGTERM to initiate the controlled shutdown sequence. Worker Processes use a simple signal handler to set a global flag (`g_stop`), allowing them to exit their main loop naturally.

5.2 Cleanup Sequence

The Master Process executes the following critical sequence upon receiving a termination signal:

1. **Terminate Workers:** The Master sends the SIGTERM signal to all active Worker Processes using kill().
2. **Reap Workers:** The Master waits for all Workers to terminate by calling waitpid() repeatedly until all child processes are reaped, preventing zombie processes.
3. **IPC Resource Unlink (Destruction):** This is the mandatory final step to destroy the named POSIX resources that persist in the file system space. The Master explicitly calls sem_unlink() and shm_unlink():
    a. sem_unlink(SEM_STATS_NAME)
    b. sem_unlink(SEM_LOG_NAME)

    c. shm_unlink(SHM_NAME)

4. **Master Exit:** The Master closes the listening socket and exits cleanly.

---

6. Conclusion

This Design Document specifies a robust, high-concurrency HTTP server based on a Master-Worker architecture utilizing the Shared Port Accept model. The design successfully addresses the core challenges of concurrent systems programming:

- **Inter-Process Communication (IPC):** Shared statistics and logging are strictly protected via named POSIX Semaphores.
- **Intra-Process Synchronization:** Performance-critical structures like the LRU File Cache are thread-safe and optimized for concurrent reads through Pthread Reader-Writer Locks.
- **Reliability:** A complete graceful shutdown procedure ensures clean termination of all worker processes and the proper unlinking of all named IPC resources (shared memory and semaphores).

With the system architecture and synchronization protocols defined, the project is ready to proceed to implementation and performance testing.