Sistemas Operativos **Ano lectivo 2025/2026**

Trabalho Prático 2

Multi-Threaded Web Server with IPC and Semaphores – Announcement of Project Proposal[1]

Multi-Threaded Web Server with IPC and Semaphores

Build a production-grade concurrent web server!

Master process synchronization, thread management, and IPC in a real-world application

Introduction

Welcome to the most challenging and rewarding project of this semester! You will design and implement a multi-threaded web server that handles concurrent HTTP requests using advanced synchronization mechanisms. This project simulates real-world server architecture where multiple processes and threads must coordinate efficiently to serve thousands of clients simultaneously.

Your server will demonstrate mastery of Inter-Process Communication (IPC), POSIX semaphores, pthread mutex locks, and condition variables to manage shared resources, prevent race conditions, and ensure optimal performance under load.

Project Overview

Project Name: ConcurrentHTTP Server

Duration: 3 weeks (21 days)

Difficulty Level: Advanced (requires strong C programming and OS concepts)

Team Size: Individual or pairs (instructor's discretion)

[1] The text of this project proposal had AI contributions to its completion.

Sistemas Operativos **Ano lectivo 2025/2026**

What You Will Build

You will create a multi-process, multi-threaded HTTP/1.1 web server with the following architecture:

Master Process	Worker Processes
• Accepts incoming connections • Manages worker processes • Coordinates via IPC • Collects statistics	• Thread pool per process • Handle HTTP requests • Serve static files • Share resources safely

Learning Objectives

1. Implement multi-process architecture using fork() and process coordination
2. Create thread pools using pthreads for concurrent request handling
3. Use POSIX semaphores for inter-process synchronization
4. Apply mutex locks and condition variables for thread synchronization
5. Implement shared memory for IPC between processes
6. Handle concurrent access to shared resources (logs, statistics, file cache)
7. Prevent race conditions, deadlocks, and resource starvation
8. Design and implement a producer-consumer pattern for request queue management

Sistemas Operativos **Ano lectivo 2025/2026**

Technical Specifications

System Architecture

1. Master Process

The master process is responsible for:

- **Socket Initialization:** Create and bind socket, listen on configurable port (default 8080)
- **Worker Management:** Fork N worker processes (configurable, default 4)
- **Connection Distribution:** Accept connections and distribute to workers via shared queue
- **IPC Setup:** Initialize shared memory segments and semaphores
- **Signal Handling:** Graceful shutdown on SIGINT/SIGTERM
- **Statistics Monitoring:** Collect and display server statistics periodically

2. Worker Processes

Each worker process maintains:

- **Thread Pool:** Create M threads (configurable, default 10 per worker)
- **Request Queue:** Bounded buffer for incoming connections (producer-consumer)
- **Thread Synchronization:** Mutex and condition variables for queue access
- **File Cache:** LRU cache for frequently accessed files (thread-safe)
- **Local Statistics:** Track requests served, bytes transferred, errors

Required Synchronization Mechanisms

Mechanism	Purpose	API
POSIX Semaphores	Inter-process synchronization for shared memory access	sem_open, sem_wait, sem_post, sem_close
Pthread Mutex	Thread-level mutual exclusion for critical sections	pthread_mutex_lock, pthread_mutex_unlock
Condition Variables	Thread signaling for queue operations (empty/full)	pthread_cond_wait, pthread_cond_signal
Shared Memory	IPC for statistics and connection queue	shm_open, mmap, shm_unlink

Sistemas Operativos **Ano lectivo 2025/2026**

HTTP Server Requirements

Your server must implement the following HTTP/1.1 features:

1. **HTTP Methods:** Support GET and HEAD requests
2. **Static File Serving:** Serve HTML, CSS, JS, images from document root
3. **Status Codes:** 200 OK, 404 Not Found, 403 Forbidden, 500 Internal Server Error
4. **Response Headers:** Content-Type, Content-Length, Server, Date
5. **MIME Types:** Detect and set correct Content-Type (html, css, js, png, jpg, etc.)
6. **Directory Index:** Serve index.html for directory requests
7. **Error Pages:** Custom error pages for 4xx and 5xx errors
8. **Logging:** Thread-safe logging to file with timestamps

Sistemas Operativos **Ano lectivo 2025/2026**

Required Features Implementation

Feature 1: Connection Queue (Producer-Consumer)

Implementation Details:

- Master process accepts connections (producer)
- Worker threads dequeue and process connections (consumers)
- Bounded circular buffer in shared memory (size: 100 connections)
- Use semaphores: empty_slots, filled_slots, mutex
- Handle queue full scenario gracefully (reject with 503 Service Unavailable)

Feature 2: Thread Pool Management

Implementation Details:

- Each worker process creates fixed number of threads at startup
- Threads block on condition variable waiting for work
- Use mutex to protect shared request queue within process
- Implement proper cleanup on shutdown (join all threads)

Feature 3: Shared Statistics

Track the following metrics:

- Total requests served
- Bytes transferred
- Requests per HTTP status code (200, 404, 500, etc.)
- Active connections count
- Average response time

Synchronization Requirements:

- Store in shared memory accessible by all processes
- Use POSIX semaphores for atomic updates
- Master process displays statistics every 30 seconds

Feature 4: Thread-Safe File Cache

Implementation Details:

- LRU cache for small files (< 1MB) within each worker process
- Maximum cache size: 10MB per worker
- Use reader-writer locks (pthread_rwlock) for cache access
- Multiple threads can read simultaneously
- Exclusive access for cache modifications

Feature 5: Thread-Safe Logging

Log Format (Apache Combined Log Format):

```
127.0.0.1 - - [10/Nov/2025:13:55:36 -0800] "GET /index.html HTTP/1.1"
200 2048
```

Sistemas Operativos **Ano lectivo 2025/2026**

Synchronization Requirements:

- Single log file shared by all processes and threads
- Use POSIX semaphore for atomic file writes
- Buffer log entries in memory and flush periodically
- Implement log rotation when file exceeds 10MB

Sistemas Operativos **Ano lectivo 2025/2026**

Configuration and Build System

Configuration File (server.conf)

Create a text-based configuration file with the following parameters:

```
PORT=8080
DOCUMENT_ROOT=/var/www/html
NUM_WORKERS=4
THREADS_PER_WORKER=10
MAX_QUEUE_SIZE=100
LOG_FILE=access.log
CACHE_SIZE_MB=10
TIMEOUT_SECONDS=30
```

Makefile Requirements

Your Makefile must support the following targets:

- **make all:** Build the server executable
- **make clean:** Remove object files and executables
- **make run:** Build and start the server
- **make test:** Run automated tests
- **Compiler flags:** -Wall -Wextra -pthread -lrt

Project Structure

```
concurrent-http-server/
├── src/
│   ├── main.c
│   ├── master.c/h
│   ├── worker.c/h
│   ├── http.c/h
│   ├── thread_pool.c/h
│   ├── cache.c/h
│   ├── logger.c/h
│   ├── stats.c/h
│   └── config.c/h
├── www/
│   └── index.html
├── tests/
│   ├── test_load.sh
│   └── test_concurrent.c
```

Sistemas Operativos **Ano lectivo 2025/2026**

```
├── docs/
│   ├── design.pdf
│   ├── report.pdf
│   └── user_manual.pdf
├── Makefile
├── server.conf
└── README.md
```

Testing Requirements

Functional Tests

9. Test GET requests for various file types (HTML, CSS, JS, images)
10. Verify correct HTTP status codes (200, 404, 403, 500)
11. Test directory index serving
12. Verify correct Content-Type headers

Concurrency Tests

13. Use Apache Bench (ab) to test concurrent requests: ab -n 10000 -c 100
14. Verify no dropped connections under load
15. Test with multiple clients using curl or wget in parallel
16. Verify statistics accuracy under concurrent load

Synchronization Tests

17. Use Helgrind or Thread Sanitizer to detect race conditions
18. Verify log file integrity (no interleaved log entries)
19. Test cache consistency across threads
20. Verify statistics counters match actual requests (no lost updates)

Stress Tests

21. Run for 5+ minutes with continuous load
22. Monitor for memory leaks using Valgrind
23. Test graceful shutdown under load
24. Verify no zombie processes remain after shutdown

Project Deliverables

1. **Source Code:** Complete C implementation with modular design
2. **Makefile:** Build system with all required targets
3. **Configuration File:** server.conf with all parameters
4. **Design Document:** 5-7 pages with architecture diagrams, flowcharts, synchronization analysis
5. **Technical Report:** 8-12 pages covering implementation details, challenges, solutions, performance analysis
6. **User Manual:** Compilation, configuration, execution instructions
7. **Test Suite:** Automated tests with results and logs
8. **Demo Website:** Sample HTML pages to test server functionality
9. **Presentation:** 10-minute demo + 5 minutes Q&A

Grading Rubric (Total: 100 Percentage Points)

Component	Criteria	Points
Functionality (35%)	• Correct multi-process architecture (master-worker) • Thread pool working correctly • HTTP server handles GET/HEAD requests properly • All synchronization mechanisms working (semaphores, mutexes) • No race conditions or deadlocks • Proper file serving with correct status codes	**35**
Code Quality (20%)	• Clean, modular code structure (separate files for components) • Comprehensive error handling for all system calls • Proper memory management (no leaks verified with Valgrind) • Well-commented code explaining complex logic • Follows coding standards, compiles	**20**

Component	Criteria	Points
	with -Wall -Wextra -Werror	
Documentation (25%)	• Design document: architecture diagrams, flowcharts, synchronization analysis (5-7 pages) • Technical report: implementation details, challenges, solutions (8-12 pages) • User manual: compilation and execution instructions with examples • README with quick start guide • Code comments and inline documentation	**25**
Testing (10%)	• Comprehensive test suite covering all features • Load testing with Apache Bench (10,000 requests minimum) • Concurrency verification with multiple simultaneous clients • Test documentation with results and analysis • Edge case handling demonstrated	**10**

Sistemas Operativos **Ano lectivo 2025/2026**

Component	Criteria	Points
Design & Creativity (10%)	• Problem complexity and architectural sophistication • Innovative synchronization solutions • Performance optimizations (caching, efficient data structures) • Additional features beyond minimum requirements • Thoughtful design decisions with clear rationale	**10**
TOTAL		**100**

Bonus Features (Optional - Up to +15 Extra Points)

Students may earn extra credit by implementing advanced features:

- **HTTP Keep-Alive (Persistent Connections):** +3 points
- **Range Requests (Partial Content / HTTP 206):** +3 points
- **CGI Script Execution:** +5 points
- **Virtual Host Support:** +4 points
- **HTTPS/SSL Support:** +7 points
- **Real-time Web Dashboard (Statistics Viewer):** +5 points
- **WebSocket Support:** +6 points

Sistemas Operativos **Ano lectivo 2025/2026**

Constraints and Rules

- **Pure C implementation:** No C++ features
- **Standard libraries only:** Use POSIX APIs, no third-party libraries
- **No busy-waiting:** All threads must block on condition variables or semaphores
- **Memory management:** No memory leaks (verified with Valgrind)
- **Error handling:** Check all system call return values
- **Compilation:** Must compile with -Wall -Wextra -Werror
- **Academic integrity:** All code must be original (plagiarism will result in zero)

Recommended Resources

Required Reading

- POSIX Threads Programming: https://computing.llnl.gov/tutorials/pthreads/
- HTTP/1.1 RFC 2616: https://www.rfc-editor.org/rfc/rfc2616
- Beej's Guide to Network Programming
- The Linux Programming Interface (Kerrisk) - Chapters 30, 53-54

Tools

- **Apache Bench (ab):** Load testing
- **Valgrind:** Memory leak detection
- **Helgrind:** Thread error detection
- **GDB:** Debugging (supports multi-threaded debugging)
- **strace:** System call tracing
- **curl/wget:** Testing HTTP requests

Important Dates

Milestone	Due Date
Design Document	End of Week 1
Code Checkpoint (Demo)	End of Week 2
Final Submission	End of Week 3 – December 12, 2025
Presentations	Following week (December 15-19)

Sistemas Operativos **Ano lectivo 2025/2026**

Submission Guidelines

Submit a single compressed archive (tar.gz or zip) named:

• Name of the file: SO-2526-T2-Px-Gy-11111-22222.tgz, where:

o Px is the number of the practical class (P1..P6);

o Gy is the number of the group within the practical class;

o 11111 and 22222 are the mec. numbers of the students in the group.

Deadline: December 12, 2025

Archive must contain:

- Complete source code with proper directory structure
- Makefile that builds successfully
- All documentation (PDFs in docs/ directory)
- Test scripts and sample website
- README.md with quick start instructions

Final Notes

This project represents the culmination of your operating systems education this semester. It combines process management, thread synchronization, inter-process communication, and practical systems programming in a real-world application. You will face challenges with concurrency bugs, deadlocks, and performance optimization - these are the same challenges that software engineers deal with daily in production systems.

Start early. Test frequently. Debug systematically. Document thoroughly.

Office hours and TA support are available throughout the project. Don't hesitate to ask questions - debugging concurrent programs is difficult, and we're here to help you learn.

Good luck! We look forward to seeing your web servers in action!

For questions or clarifications, please contact Prof. Pedro Azevedo Fernandes (paf@ua.pt) or Prof. Nuno Lau (nunolau@ua.pt).