Multi-Threaded Web Server with IPC and Semaphores

Sistems Operativos – TP2

A production-grade concurrent HTTP/1.1 web server implementing advanced process and thread synchronization using POSIX semaphores, shared memory, and thread pools.

Table of Contents

Overview

This project implements a multi-process, multi-threaded HTTP/1.1 web server that demonstrates:

- **Process Management**: Master-worker architecture using `fork()`
- **Inter-Process Communication**: Shared memory and POSIX semaphores
- **Thread Synchronization**: Pthread mutexes, condition variables, reader-writer locks
- **Concurrent Request Handling**: Thread pools with producer-consumer pattern
- **HTTP Protocol**: Full HTTP/1.1 support including GET and HEAD methods
- **Resource Management**: Thread-safe LRU file cache and statistics tracking

Architecture

```
Master Process
├── Accepts TCP connections (port 8080)
├── Manages shared memory and semaphores
├── Distributes connections to workers
└── Monitors server statistics
```

```
Worker Processes (4 workers)
├── Each maintains a thread pool (10 threads)
├── Threads process HTTP requests
├── Thread-safe LRU file cache
└── Update shared statistics
```

✨ Features

Core Features

- ✅ **Multi-Process Architecture**: 1 master + N workers (default: 4)
- ✅ **Thread Pools**: M threads per worker (default: 10)
- ✅ **HTTP/1.1 Support**: GET and HEAD methods
- ✅ **Status Codes**: 200, 404, 403, 500, 503
- ✅ **MIME Types**: HTML, CSS, JavaScript, images (PNG, JPG), PDF
- ✅ **Directory Index**: Automatic index.html serving
- ✅ **Custom Error Pages**: Branded 404 and 500 pages

Synchronization Features

- ✅ **POSIX Semaphores**: Inter-process synchronization
- ✅ **Pthread Mutexes**: Thread-level mutual exclusion
- ✅ **Condition Variables**: Producer-consumer queue signaling
- ✅ **Reader-Writer Locks**: Thread-safe file cache access

Advanced Features

- ✅ **Thread-Safe LRU Cache**: 10MB cache per worker with intelligent eviction
- ✅ **Apache Combined Log Format**: Industry-standard logging
- ✅ **Shared Statistics**: Real-time request tracking across all workers
- ✅ **Configuration File**: Flexible server.conf for easy customization
- ✅ **Log Rotation**: Automatic rotation at 10MB
- ✅ **Graceful Shutdown**: Proper cleanup on SIGINT/SIGTERM

Bonus Features (if implemented)

- ⭐ **HTTP Keep-Alive**: Persistent connections
- ⭐ **CGI Support**: Dynamic script execution
- ⭐ **Virtual Hosts**: Multiple domains on one server
- ⭐ **HTTPS Support**: SSL/TLS encryption
- ⭐ **Real-time Dashboard**: Web-based statistics viewer

System Requirements

Required Software

- **Operating System**: Linux (Ubuntu 20.04+ or similar)
- **Compiler**: GCC 9.0+ with C11 support
- **Libraries**:
  - POSIX threads (`pthread`)
  - POSIX real-time extensions (`rt`)
  - Standard C library

Development Tools

- **Build System**: GNU Make
- **Testing**: Apache Bench (`ab`), curl, wget
- **Debugging**: GDB, Valgrind, Helgrind
- **Version Control**: Git (recommended)

Installation of Prerequisites

```
# Ubuntu/Debian
sudo apt-get update
sudo apt-get install -y build-essential gcc make
sudo apt-get install -y apache2-utils curl valgrind

# Verify installation
gcc --version
ab -V
valgrind --version
```

Project Structure

```
concurrent-http-server/
│
├── src/                          # Source code
│   ├── main.c                    # Program entry point
│   ├── master.c/h                # Master process implementation
│   ├── worker.c/h                # Worker process implementation
│   ├── http.c/h                  # HTTP request/response handling
│   ├── thread_pool.c/h           # Thread pool management
│   ├── cache.c/h                 # LRU cache implementation
│   ├── logger.c/h                # Thread-safe logging
│   ├── stats.c/h                 # Shared statistics
│   └── config.c/h                # Configuration file parser
│
```

```
├── www/                          # Document root
│   ├── index.html               # Main page
│   ├── style.css                # Stylesheets
│   ├── script.js                # JavaScript files
│   ├── images/                  # Image assets
│   └── errors/                  # Custom error pages
│       ├── 404.html
│       └── 500.html
│
├── tests/                        # Test suite
│   ├── test_load.sh             # Load testing script
│   ├── test_concurrent.c        # Concurrency tests
│   └── README.md                # Test documentation
│
├── docs/                         # Documentation
│   ├── design.pdf               # Architecture and design
│   ├── report.pdf               # Technical report
│   └── user_manual.pdf          # User guide
│
├── Makefile                      # Build system
├── server.conf                   # Configuration file
├── README.md                     # This file
└── .gitignore                    # Git ignore rules
```

🔨 Compilation

Quick Start

```
# Build the server
make


# Clean build artifacts
make clean


# Rebuild from scratch
make clean && make


# Build and run
make run
```

Build Targets

```
make all          # Build server executable (default)
make clean        # Remove object files and executable
make run          # Build and start server
make test         # Build and run test suite
make debug        # Build with debug symbols (-g)
make release      # Build with optimizations (-O3)
make valgrind     # Build and run under Valgrind
make helgrind     # Build and run under Helgrind
```

Manual Compilation

```
# Compile all source files
gcc -Wall -Wextra -pthread -lrt -o server \
    src/main.c \
    src/master.c \
    src/worker.c \
    src/http.c \
    src/thread_pool.c \
    src/cache.c \
    src/logger.c \
    src/stats.c \
    src/config.c

# Run the server
./server
```

Compiler Flags Explained

- `-Wall -Wextra`: Enable all warnings
- `-pthread`: Link pthread library
- `-lrt`: Link POSIX real-time extensions (for semaphores)
- `-O3`: Optimization level 3 (for release builds)
- `-g`: Debug symbols (for debugging)
- `-fsanitize=thread`: Thread sanitizer (for race detection)

⚙ Configuration

Configuration File (server.conf)

The server reads configuration from `server.conf` in the current directory:

```
# Server Configuration File

# Network settings
PORT=8080                          # Port to listen on
TIMEOUT_SECONDS=30                 # Connection timeout

# File system
DOCUMENT_ROOT=/var/www/html        # Root directory for serving files

# Process architecture
NUM_WORKERS=4                      # Number of worker processes
THREADS_PER_WORKER=10              # Threads per worker

# Queue management
MAX_QUEUE_SIZE=100                 # Connection queue size

# Caching
CACHE_SIZE_MB=10                   # Cache size per worker (MB)

# Logging
LOG_FILE=access.log                # Access log file path
LOG_LEVEL=INFO                     # Log level: DEBUG, INFO, WARN, ERROR
```

Configuration Parameters

Parameter	Default	Description
PORT	8080	TCP port for HTTP server
DOCUMENT_ROOT	./www	Root directory for serving files
NUM_WORKERS	4	Number of worker processes
THREADS_PER_WORKER	10	Thread pool size per worker
MAX_QUEUE_SIZE	100	Connection queue capacity
CACHE_SIZE_MB	10	Maximum cache size per worker
LOG_FILE	access.log	Path to access log
TIMEOUT_SECONDS	30	Connection timeout

Environment Variables

```
# Override configuration via environment
export HTTP_PORT=8080
export HTTP_WORKERS=4
export HTTP_THREADS=10

./server
```

🚀 Usage

Starting the Server

```
# Start with default configuration
./server

# Start with custom configuration file
./server -c /path/to/server.conf

# Start on specific port
./server -p 9090

# Start with verbose logging
./server -v

# Start in background (daemon mode)
./server -d
```

Command-Line Options

```
Usage: ./server [OPTIONS]

Options:
  -c, --config PATH      Configuration file path (default: ./server.conf)
  -p, --port PORT        Port to listen on (default: 8080)
  -w, --workers NUM      Number of worker processes (default: 4)
  -t, --threads NUM      Threads per worker (default: 10)
  -d, --daemon           Run in background
  -v, --verbose          Enable verbose logging
  -h, --help             Show this help message
  --version              Show version information
```

Accessing the Server

```
# Open in browser
firefox http://localhost:8080

# Using curl
curl http://localhost:8080/index.html

# View headers only (HEAD request)
curl -I http://localhost:8080/index.html

# Download file
wget http://localhost:8080/document.pdf
```

Stopping the Server

```
# Graceful shutdown (from server terminal)
Ctrl+C

# Send SIGTERM
kill $(pgrep -f "./server")

# Force kill (not recommended)
kill -9 $(pgrep -f "./server")
```

Viewing Logs

```
# Follow access log in real-time
tail -f access.log

# View last 100 entries
tail -n 100 access.log

# Search for errors
grep "500\|404" access.log

# Count requests by status code
awk '{print $9}' access.log | sort | uniq -c
```

Monitoring Statistics

Statistics are displayed every 30 seconds on stdout:

```
========================================
SERVER STATISTICS
========================================
Uptime: 120 seconds
Total Requests: 1,542
Successful (2xx): 1,425
Client Errors (4xx): 112
Server Errors (5xx): 5
Bytes Transferred: 15,728,640
Average Response Time: 8.3 ms
Active Connections: 12
Cache Hit Rate: 82.4%
========================================
```

Testing

Functional Tests

```
# Run all tests
make test

# Basic functionality test
curl http://localhost:8080/index.html

# Test HEAD method
curl -I http://localhost:8080/index.html

# Test 404 error
curl http://localhost:8080/nonexistent.html

# Test different file types
curl http://localhost:8080/style.css        # CSS
curl http://localhost:8080/script.js        # JavaScript
curl http://localhost:8080/image.png        # Image
```

Load Testing

```
# Basic load test (1000 requests, 10 concurrent)
ab -n 1000 -c 10 http://localhost:8080/index.html

# High concurrency test (10000 requests, 100 concurrent)
ab -n 10000 -c 100 http://localhost:8080/index.html

# Sustained load test (5 minutes)
ab -t 300 -c 50 http://localhost:8080/

# Test multiple files
for file in index.html style.css script.js; do
    ab -n 1000 -c 50 http://localhost:8080/$file
done
```

Concurrency Testing

```
# Parallel requests with curl
for i in {1..100}; do
    curl -s http://localhost:8080/index.html &
done
wait

# Parallel requests with different files
for i in {1..50}; do
    curl -s http://localhost:8080/page$((i % 10)).html &
done
wait
```

Memory Leak Detection

```
# Run server under Valgrind
make valgrind

# Or manually:
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --log-file=valgrind.log \
         ./server

# In another terminal, generate traffic
ab -n 1000 -c 50 http://localhost:8080/

# Stop server and check valgrind.log
```

Race Condition Detection

```
# Run server under Helgrind
make helgrind

# Or manually:
valgrind --tool=helgrind \
         --log-file=helgrind.log \
         ./server

# Generate concurrent traffic
ab -n 5000 -c 100 http://localhost:8080/

# Check helgrind.log for data races
```

Performance Testing

```
# Measure cache effectiveness
# First request (cache miss)
time curl -s http://localhost:8080/large.html > /dev/null

# Subsequent requests (cache hit)
for i in {1..10}; do
    time curl -s http://localhost:8080/large.html > /dev/null
done

# Monitor server resource usage
top -p $(pgrep -f "./server")

# Monitor worker processes
watch -n 1 'pgrep -P $(pgrep -f "./server") | xargs ps -o pid,ppid,nlwp,cmd'
```

🏗️ Implementation Details

Master Process

Responsibilities:

- Initialize listening socket on configured port
- Create shared memory segments for connection queue and statistics
- Initialize POSIX semaphores for synchronization
- Fork N worker processes
- Accept incoming TCP connections
- Enqueue connections in shared queue
- Handle signals (SIGINT, SIGTERM) for graceful shutdown
- Display statistics periodically

Key Functions:

```
int master_init(config_t *config);
void master_accept_loop(int listen_fd);
void master_signal_handler(int signum);
void master_cleanup(void);
```

Worker Processes

Responsibilities:

- Create thread pool with M threads
- Initialize thread-safe LRU cache
- Threads dequeue connections from shared queue
- Parse HTTP requests
- Serve files from document root
- Update shared statistics
- Handle graceful shutdown

Key Functions:

```
void worker_main(int worker_id);
void* worker_thread(void *arg);
void worker_process_request(int client_fd);
```

Thread Pool

Implementation:

- Fixed-size thread pool (no dynamic creation)
- Condition variables for empty/full queue
- Mutex protection for queue access
- Threads block when no work available

Queue Operations:

```
void thread_pool_init(thread_pool_t *pool, int num_threads);
void thread_pool_enqueue(thread_pool_t *pool, int client_fd);
int thread_pool_dequeue(thread_pool_t *pool);
void thread_pool_shutdown(thread_pool_t *pool);
```

HTTP Request Handling

Supported Methods:

- **GET**: Retrieve resource, return headers + body
- **HEAD**: Retrieve headers only, no body

Request Parsing:

```
http_request_t* http_parse_request(char *buffer);
char* http_build_response(http_request_t *req, int *status_code);
void http_send_response(int fd, char *response, size_t size);
```

MIME Type Detection:

```
const char* get_mime_type(const char *filename);
// .html → text/html
// .css  → text/css
// .js   → application/javascript
// .png  → image/png
// .jpg  → image/jpeg
// .pdf  → application/pdf
```

LRU Cache Implementation

Algorithm:

- Doubly-linked list for LRU ordering
- Hash table for O(1) lookup
- Reader-writer lock for thread safety
- Maximum size enforced (10MB per worker)

Cache Operations:

```
cache_t* cache_init(size_t max_size_mb);
cache_entry_t* cache_get(cache_t *cache, const char *key);
void cache_put(cache_t *cache, const char *key, void *data, size_t size);
void cache_evict_lru(cache_t *cache);
void cache_destroy(cache_t *cache);
```

Synchronization:

```
pthread_rwlock_rdlock(&cache->lock);  // Multiple readers
// Read cached file
pthread_rwlock_unlock(&cache->lock);

pthread_rwlock_wrlock(&cache->lock);  // Exclusive writer
// Update cache
pthread_rwlock_unlock(&cache->lock);
```

Logging

Format: Apache Combined Log

```
127.0.0.1 - - [22/Nov/2025:14:30:45 -0800] "GET /index.html HTTP/1.1" 200
2048
```

Thread-Safe Implementation:

```
sem_wait(&log_semaphore);     // Acquire lock
fprintf(log_file, ...);        // Write log entry
fflush(log_file);             // Ensure written
sem_post(&log_semaphore);     // Release lock
```

Log Rotation:

```
if (log_size > 10 * 1024 * 1024) {
    rename("access.log", "access.log.1");
    log_file = fopen("access.log", "a");
}
```

Shared Statistics

Tracked Metrics:

```
typedef struct {
    uint64_t total_requests;
    uint64_t bytes_transferred;
    uint32_t status_counts[600];  // Status code histogram
    uint32_t active_connections;
    double avg_response_time;
    time_t start_time;
} server_stats_t;
```

Atomic Updates:

```
sem_wait(&stats_semaphore);
stats->total_requests++;
stats->bytes_transferred += bytes;
stats->status_counts[status_code]++;
sem_post(&stats_semaphore);
```

Known Issues

Current Limitations

1. **No SSL/TLS Support**: HTTP only, no HTTPS (bonus feature)
2. **No Keep-Alive**: Connections close after each request (bonus feature)
3. **Static Files Only**: No CGI or dynamic content (bonus feature)
4. **Single Document Root**: No virtual hosts (bonus feature)

Platform-Specific Issues

- **macOS**: Limited support due to differences in POSIX semaphore implementation
- **Windows**: Not supported (requires POSIX APIs)

Performance Considerations

- **Large File Handling**: Files > 1MB not cached, may be slow
- **Many Small Files**: Cache thrashing possible with > 1000 distinct files
- **Queue Overflow**: 503 errors possible under extreme load (> 500 concurrent connections)

Authors

[Your Name]

- Student NMec: [NMec]
- Email: [email@ua.pt]
- GitHub: @username (https://github.com/username)

[Partner Name] (if pair project)

- Student NMec: [NMec]
- Email: [email@ua.pt]
- GitHub: @username (https://github.com/username)

Acknowledgments

References

- Official bibliographic references os the course
- POSIX Threads Programming
- HTTP/1.1 RFC 2616
- Beej's Guide to Network Programming
- The Linux Programming Interface (Michael Kerrisk)
- Operating Systems: Three Easy Pieces (Arpaci-Dusseau)
- Other documentation used…

Tools Used

- **Compiler**: GCC 11.4.0
- **Debugger**: GDB 12.1, Valgrind 3.19
- **Build System**: GNU Make 4.3
- **Editor**: [VS Code / Vim / etc.]
- **Version Control**: Git 2.34

Course Information

- **Course**: Sistemas Operativos (40381-SO)
- **Semester**: 1º semester 2025/2026
- **Instructor**: Prof. Pedro Azevedo Fernandes / Prof. Nuno Lau
- **Institution**: Universidade de Aveiro

License

This project is submitted as coursework for Sistemas Operativos at Universidade de Aveiro.

All code is original work by the authors unless otherwise noted in source file comments.

Support

Getting Help

1. **Read the documentation** (this README, user manual)
2. **Check course materials** (lecture slides, textbook)
3. **Attend office hours** (schedule on course website)
4. **Post on course forum** (for general questions)
5. **Email instructor** (for specific issues)

Submitting Issues

If you find bugs or have suggestions (for future semesters):

1. Describe the issue clearly
2. Provide steps to reproduce
3. Include server logs and error messages
4. Specify your environment (OS, compiler version)

Additional Documentation

- **Design Document**: Architecture diagrams and design decisions
- **Technical Report**: Implementation details and performance analysis
- **User Manual**: Complete usage guide

Submission Checklist

Before submitting your project, verify:

Code compiles without warnings (`make clean && make`)

All required features implemented (see project specification)

Passes basic functional tests

Handles load test with 10,000 requests successfully

No memory leaks (verified with Valgrind)

No race conditions (verified with Helgrind)

Proper error handling (all system calls checked)

Code documented with comments

Design document completed (5-7 pages)

Technical report completed (8-12 pages)

User manual completed (2-4 pages)

Test suite included with results

Demo website included in www/

README.md completed

Submission archive created correctly

Creating Submission Archive

```
# Create submission archive
tar -czf -czf SO-2526-T2-Px-Gy-11111-22222.tar.gz \
    concurrent-http-server/

# Verify archive contents
tar -tzf -czf SO-2526-T2-Px-Gy-11111-22222.tar.gz

# Check archive size
ls -lh -czf SO-2526-T2-Px-Gy-11111-22222.tar.gz
```

Last Updated: November 22, 2025
Version: 2.0
Project Status: 🚀 Ready for Submission

🎓 Learning Outcomes

By completing this project, you have demonstrated:

- ✅ Understanding of process management and IPC
- ✅ Proficiency with thread synchronization primitives
- ✅ Ability to design concurrent systems
- ✅ Knowledge of network programming
- ✅ Skills in debugging multi-threaded programs
- ✅ Experience with real-world systems programming

Congratulations on completing this challenging project! 🎉

For questions or issues, contact: [paf@ua.pt]