universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Sistemas Operativos **Ano lectivo 2025/2026**

Frequently Asked Questions[1]

Multi-Threaded Web Server Project

Student Support Guide

This FAQ document addresses the most common questions students have when building the concurrent web server. Read through all sections before starting your implementation. If you don't find your answer here, please attend office hours or post on the course forum.

Getting Started

Q1: What should I implement first?

Start with a single-threaded, single-process HTTP server that can serve static files. Once that works, add the master-worker architecture, then add threading, and finally add all synchronization mechanisms. Building incrementally makes debugging much easier.

Recommended order:

1. Basic socket server (listen, accept, read, write)
2. HTTP request parsing and response generation
3. File serving with proper MIME types
4. Master process with worker processes (fork)
5. Thread pool in each worker
6. Shared memory and semaphores
7. Statistics, logging, and caching

Q2: How do I test if my server is working?

Use a web browser! Navigate to http://localhost:8080. You can also use curl or wget from the command line:

```
curl -v http://localhost:8080/index.html
wget http://localhost:8080/test.css
```

Q3: What development environment should I use?

Use a Linux environment (Ubuntu, Fedora, or WSL2 on Windows). POSIX semaphores and shared memory work best on Linux. You can develop in any editor (VS Code, Vim, Emacs), but make sure you test on Linux.

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

Sistemas Operativos **Ano lectivo 2025/2026**

Process and Thread Architecture

Q4: Why do we need both processes and threads?

This architecture demonstrates both IPC (inter-process) and thread synchronization. Processes provide isolation and fault tolerance - if one worker crashes, others continue. Threads within a process share memory efficiently and handle concurrent connections. This mirrors production servers like Apache MPM and Nginx.

Q5: How does the master process distribute connections to workers?

The master accepts connections and places socket file descriptors into a shared memory queue. Worker processes dequeue these descriptors and pass them to their thread pools. Use POSIX semaphores to coordinate access to this shared queue.

Q6: How do I pass a socket descriptor between processes?

After fork(), both parent and child share the same file descriptor table. You can store the accepted socket descriptor in shared memory, and the worker process can read and use it directly. Alternatively, use Unix domain sockets with sendmsg/recvmsg to pass descriptors (advanced).

Q7: How many workers and threads should I create?

Make these configurable in your server.conf file. Recommended defaults: 4 worker processes, 10 threads per worker. This gives you 40 concurrent connections. On a machine with N CPU cores, use N or N+1 workers for optimal performance.

Sistemas Operativos **Ano lectivo 2025/2026**

Synchronization Mechanisms

Q8: When should I use semaphores vs. mutexes?

Use POSIX semaphores for inter-process synchronization (protecting shared memory between master and workers). **Use pthread mutexes for intra-process synchronization** (protecting shared data between threads within a worker).

Mechanism	Scope	Use Case
POSIX Semaphore	Inter-process	Shared memory queue, statistics
Pthread Mutex	Intra-process (threads)	Thread pool queue, cache
Condition Variable	Intra-process (threads)	Thread waiting/signaling

Q9: How do I implement a bounded buffer with semaphores?

Use three semaphores: empty_slots (initialized to buffer size), filled_slots (initialized to 0), and mutex (initialized to 1). Producer decrements empty, increments filled. Consumer does the opposite.

```
// Producer
sem_wait(&empty_slots);
sem_wait(&mutex);
// Add item to buffer
sem_post(&mutex);
sem_post(&filled_slots);
```

Q10: What's the difference between named and unnamed semaphores?

Named semaphores (sem_open) can be shared between unrelated processes - use these for master-worker IPC. **Unnamed semaphores** (sem_init) exist in shared memory and are faster, but both processes must have access to that memory.

Q11: My threads are not waking up from pthread_cond_wait. Why?

Always use pthread_cond_wait in a while loop, not an if statement. The thread might wake spuriously or the condition might change between signal and wakeup:

```
pthread_mutex_lock(&mutex);
while (queue_empty) {
    pthread_cond_wait(&cond, &mutex);
}
// Process item
pthread_mutex_unlock(&mutex);
```

Sistemas Operativos **Ano lectivo 2025/2026**

HTTP and Networking

Q12: How do I parse an HTTP request?

Read until you find \r\n\r\n (end of headers). Parse the first line to extract method, path, and version. Example request:

```
GET /index.html HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.68.0
```

Use strtok or sscanf to parse. Store method, path, and any headers you need.

Q13: What's the format for an HTTP response?

Status line + headers + blank line + body:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234
Server: ConcurrentHTTP/1.0

<html>...</html>
```

Q14: How do I determine the MIME type of a file?

Check the file extension. Create a lookup table:

```
const char* get_mime_type(const char* path) {
    if (ends_with(path, ".html")) return "text/html";
    if (ends_with(path, ".css")) return "text/css";
    if (ends_with(path, ".js")) return "application/javascript";
    if (ends_with(path, ".jpg")) return "image/jpeg";
    if (ends_with(path, ".png")) return "image/png";
    return "application/octet-stream";
}
```

Q15: How do I handle partial reads/writes?

Always loop until you've read/written all data. recv() and send() may return less than requested:

```
ssize_t send_all(int sock, const char* buf, size_t len) {
    size_t total = 0;
    while (total < len) {
```

```
            ssize_t n = send(sock, buf + total, len - total, 0);
            if (n <= 0) return -1;
            total += n;
        }
        return total;
    }
```

Q16: What status code should I return for different errors?

- **404 Not Found:** File doesn't exist
- **403 Forbidden:** File exists but can't be read (permissions)
- **400 Bad Request:** Malformed HTTP request
- **500 Internal Server Error:** Server-side errors (malloc fails, etc.)
- **503 Service Unavailable:** Queue full, can't accept request

Sistemas Operativos **Ano lectivo 2025/2026**

Debugging and Common Issues

Q17: My server says 'Address already in use'. How do I fix this?

Set the SO_REUSEADDR socket option before bind():

```
int opt = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

This allows immediate rebinding after the previous server exits. Also wait a few seconds after killing the server before restarting.

Q18: How do I debug race conditions?

Use Helgrind (part of Valgrind):

```
valgrind --tool=helgrind ./server
```

It will report data races, lock ordering issues, and other threading problems. Also add printf statements (but protect them with locks!) to see execution order.

Pro tip: Add sleep() calls in critical sections to make race conditions more likely to occur during testing.

Q19: My program deadlocks. How do I find the problem?

Deadlocks occur when threads wait circularly for locks. Common causes:

- Acquiring locks in different order (Thread A: lock1→lock2, Thread B: lock2→lock1)
- Forgetting to unlock a mutex
- Waiting on condition variable without proper signaling

To debug: Attach gdb to running process and check thread stacks:

```
ps aux | grep server  # Get PID
gdb -p <PID>
(gdb) info threads
(gdb) thread apply all bt
```

Q20: How do I check for memory leaks?

Use Valgrind with memcheck:

```
valgrind --leak-check=full --show-leak-kinds=all ./server
```

Common leaks in this project:

- Not freeing allocated buffers for HTTP requests/responses
- Not closing file descriptors (sockets)
- Not cleaning up thread-local storage
- Not unlinking named semaphores and shared memory

Q21: My log file has garbled entries. What's wrong?

Multiple threads are writing simultaneously without synchronization. Protect all file writes with a semaphore:

```
sem_wait(&log_sem);
fprintf(log_file, "...");
fflush(log_file);  // Important!
sem_post(&log_sem);
```

Performance and Testing

Q22: How do I test with many concurrent connections?

Use Apache Bench (ab):

```
ab -n 10000 -c 100 http://localhost:8080/
```

-n 10000 = 10,000 total requests, -c 100 = 100 concurrent requests

You can also write a simple multi-threaded client in C to have more control over the test patterns.

Q23: My server is slow. How do I optimize it?

Common bottlenecks:

- **Excessive locking:** Hold locks for minimal time. Don't do I/O while holding locks.
- **No caching:** Cache small files to avoid repeated disk reads.
- **Too few threads:** Increase thread pool size (but not too high - context switching overhead).
- **Blocking on semaphores:** Profile with strace to see where threads are waiting.

Q24: How do I know if my statistics are correct?

Run a controlled test with a known number of requests and compare:

```
# Send exactly 100 requests
ab -n 100 -c 1 http://localhost:8080/
```

Check that your server reports exactly 100 requests. If numbers don't match, you have a synchronization bug (lost updates to shared counters).

Shared Memory and IPC

Q25: How do I create shared memory between processes?

Use shm_open and mmap:

```
int shm_fd = shm_open("/my_shm", O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, sizeof(struct shared_data));
struct shared_data* data = mmap(NULL, sizeof(struct shared_data),
    PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

Clean up on exit:

```
munmap(data, sizeof(struct shared_data));
close(shm_fd);
```

```
shm_unlink("/my_shm");
```

Q26: Can I put pthread mutexes in shared memory?

Yes, but you must set the PTHREAD_PROCESS_SHARED attribute:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shm_data->mutex, &attr);
```

However, POSIX semaphores are usually simpler for inter-process synchronization.

Configuration and Best Practices

Q27: How should I structure my code?

Modular design is critical for maintainability:

- **master.c/h:** Master process logic
- **worker.c/h:** Worker process logic
- **http.c/h:** HTTP parsing and response generation
- **thread_pool.c/h:** Thread pool management
- **cache.c/h:** File caching
- **logger.c/h:** Thread-safe logging
- **stats.c/h:** Statistics collection

Q28: Should I use global variables?

Minimize global variables. Use them only for truly global state (like shared memory pointers, configuration). Pass most data through function parameters. This makes your code easier to test and debug.

Q29: How detailed should my comments be?

Every function should have a comment describing its purpose, parameters, and return value. Comment complex algorithms and synchronization logic. Don't comment obvious code. Focus on the why, not the what.

Sistemas Operativos **Ano lectivo 2025/2026**

Submission and Grading

Q30: What should be in my design document?

- Architecture diagram (processes, threads, data flow)
- Synchronization strategy (which semaphores/mutexes protect what)
- Data structures (queues, cache, statistics)
- Flowcharts for key operations (request handling, connection distribution)
- Discussion of design choices and alternatives considered

Q31: What should be in my technical report?

- Implementation details (how you implemented each feature)
- Challenges faced and how you solved them
- Testing methodology and results
- Performance analysis (throughput, latency under load)
- Lessons learned and potential improvements

Q32: My code compiles but doesn't pass all tests. Will I still get credit?

Partial credit is available. The grading rubric considers functionality (35%), code quality (20%), documentation (25%), testing (10%), and design (10%). Even if functionality is incomplete, well-documented, well-structured code with good design will earn substantial points.

Q33: Can I use ChatGPT or other AI tools?

You may use AI tools for learning and understanding concepts, but all submitted code must be your own. Using AI-generated code without understanding it is considered plagiarism. If you use AI to help debug or explain concepts, mention it in your report. The goal is to learn, not just to get working code.

Additional Resources

- **Office Hours:** See course website for schedule
- **Course Forum:** Post questions and help classmates
- **Code Templates:** See separate template document
- **Network Programming Tutorial:** Step-by-step guide (separate document)
- **Debugging Guide:** Advanced debugging techniques (separate document)

Remember: Start early, test frequently, and ask for help when you need it!

This FAQ will be updated throughout the project. Check for the latest version on the course website.