# Multi-Threaded Web Server with IPC and Semaphores

SO 25/26

# Design Document

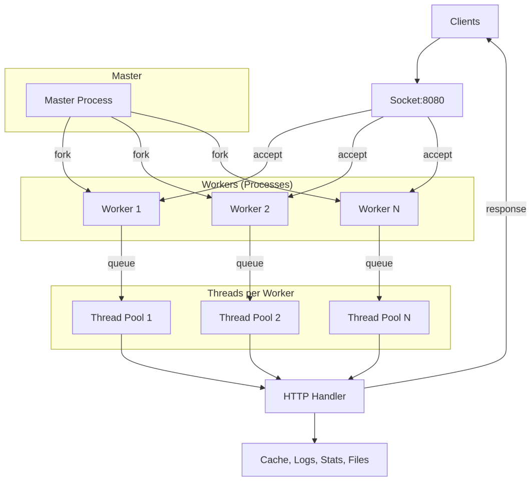
Department of Electronics, Telecommunications and Informatics

Bernardo Mota Coelho - 125059
Tiago Francisco Crespo do Vale - 125913

9/12/2025

universidade
de aveiro

# 1. Architecture diagram



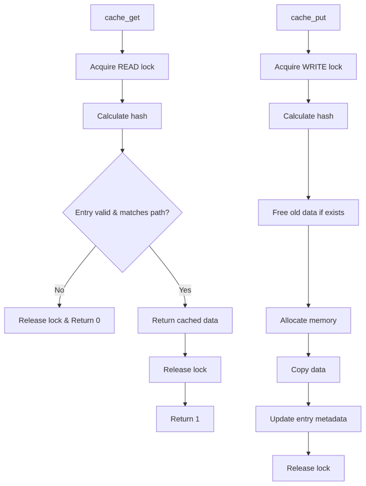
# 2. Flowcharts

### Flowcharts

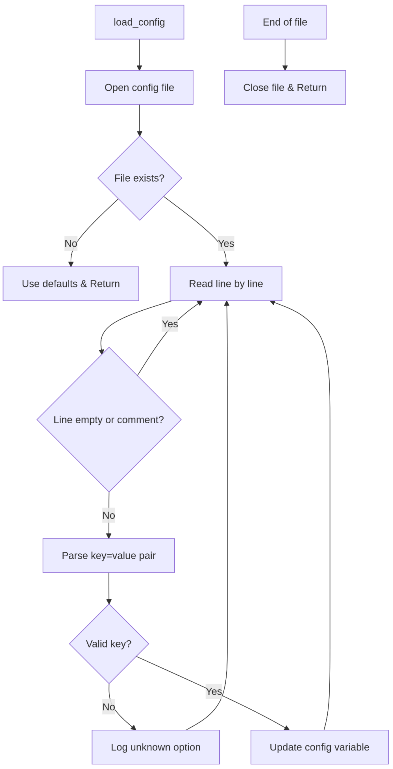
This document includes the following specific flowcharts:

3. Cache Flowchart
4. Config Flowchart
5. HTTP Flowchart
6. Main Flowchart
7. Master Flowchart
8. Semaphores Flowchart
9. Shared Memory Flowchart
10. Stats Flowchart
11. ThreadPool Flowchart
12. Worker Flowchart

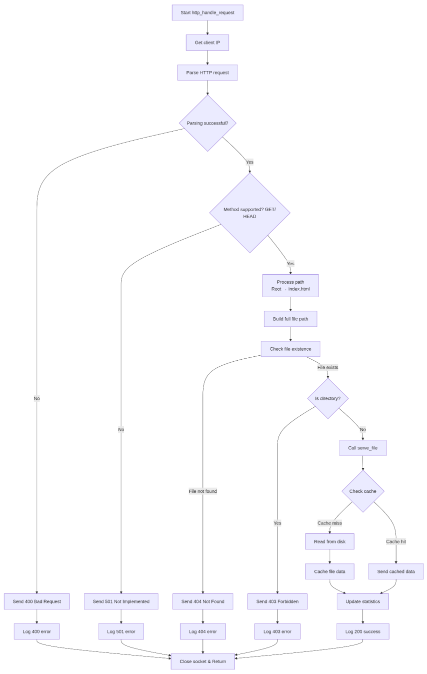
## Cache FlowChart:



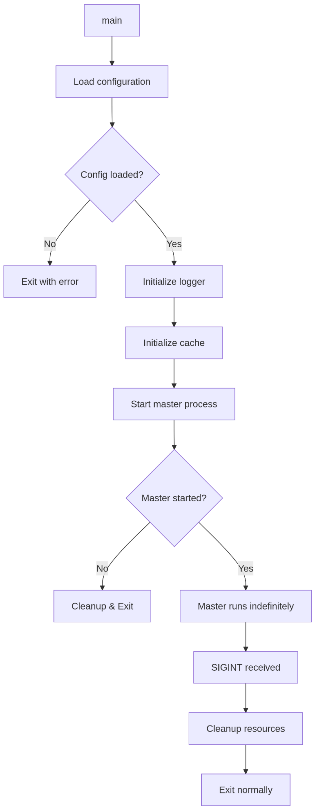
# Config FlowChart:



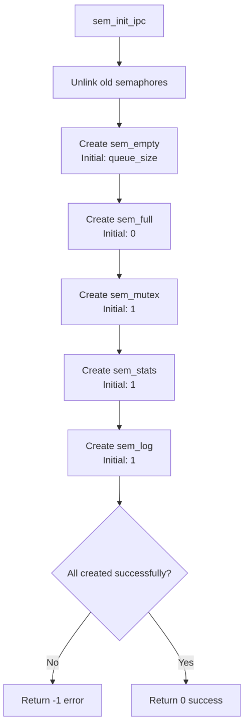
# HTTP FlowChart:



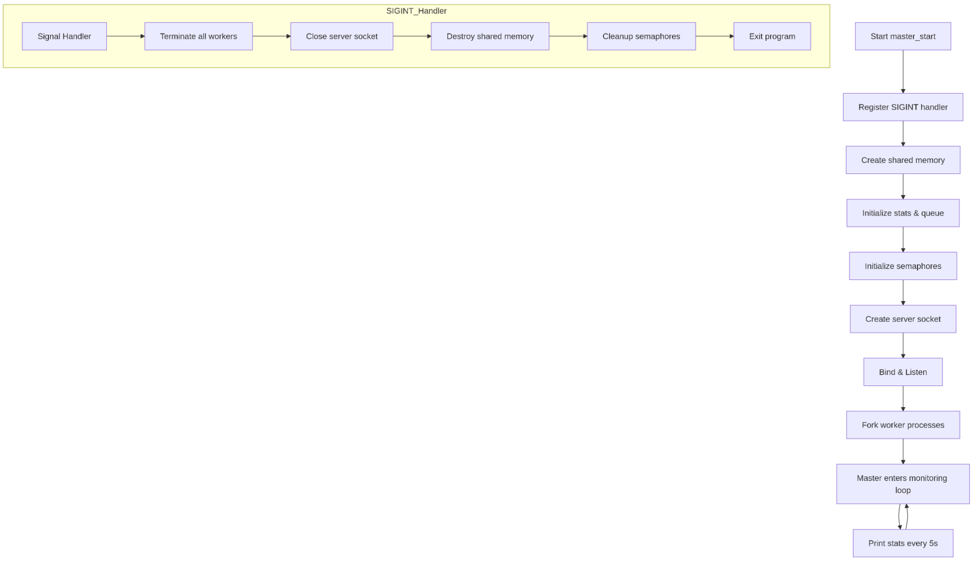
# Main FlowChart:



# Semaphores FlowChart:

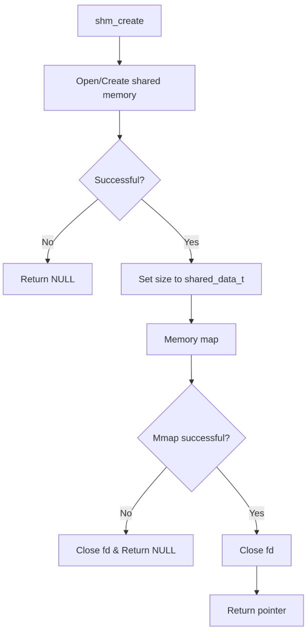


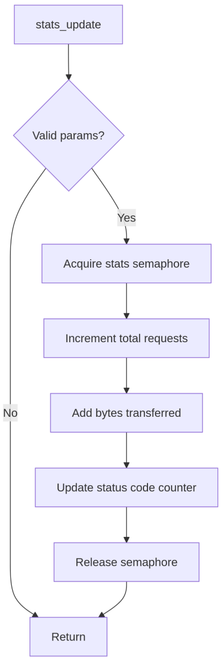
# Master FlowChart:



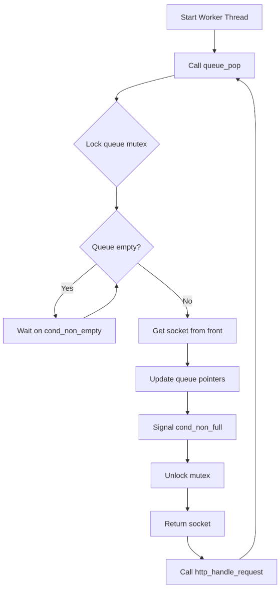
# Shared Memory FlowChart:

# Stats FlowChart:

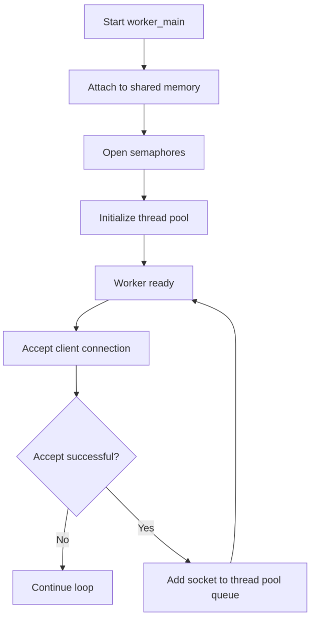




ThreadPool FlowChart:



Worker FlowChart:



# 3. Synchronization Analysis

This HTTP server uses multiple synchronization mechanisms across processes and threads to ensure data consistency and avoid race conditios.

## Process-Level Syncronization (IPC)

Shared Memory (shm_data)

```
// Master creates, workers attach
shared_data_t* shm_data = shm_create();  // master.c
shared_data_t* shm_data = shm_create();  // worker.c (attaches)
```

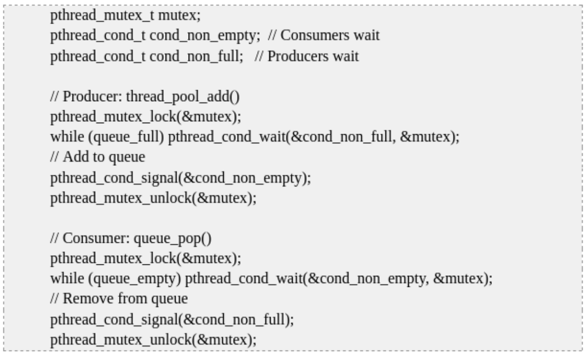
Data shared:

| | |
|---|---|
| server_stats_t stats | - Request Statistics |
| shared_queue_t queue | - Work queue (if used) |

Protection: POSIX named semaphores

### Thread-Level Synchronization

A. Thread Pool Queue (thread_pool.c)

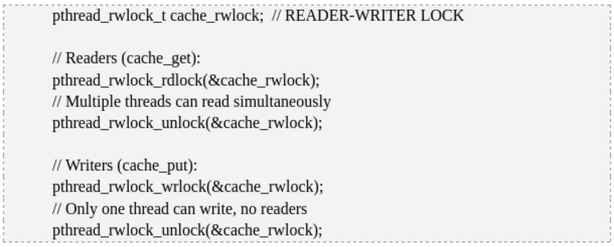


```
pthread_mutex_t mutex;
pthread_cond_t cond_non_empty;  // Consumers wait
pthread_cond_t cond_non_full;   // Producers wait

// Producer: thread_pool_add()
pthread_mutex_lock(&mutex);
while (queue_full) pthread_cond_wait(&cond_non_full, &mutex);
// Add to queue
pthread_cond_signal(&cond_non_empty);
pthread_mutex_unlock(&mutex);

// Consumer: queue_pop()
pthread_mutex_lock(&mutex);
while (queue_empty) pthread_cond_wait(&cond_non_empty, &mutex);
// Remove from queue
pthread_cond_signal(&cond_non_full);
pthread_mutex_unlock(&mutex);
```

B. Cache System (cache.c)



```
pthread_rwlock_t cache_rwlock;  // READER-WRITER LOCK

// Readers (cache_get):
pthread_rwlock_rdlock(&cache_rwlock);
// Multiple threads can read simultaneously
pthread_rwlock_unlock(&cache_rwlock);

// Writers (cache_put):
pthread_rwlock_wrlock(&cache_rwlock);
// Only one thread can write, no readers
pthread_rwlock_unlock(&cache_rwlock);
```

A: Pattern: Producer-Consumer with bounded buffer
B: Advantage: Better performance for read-heavy workload (web server cache)

## Critical Sections Analysis

Section 1: Statistics Updates

```
// stats.c - stats_update()
sem_wait(sem_stats);  //  ENTER CRITICAL SECTION
stats->total_requests++;
stats->bytes_transferred += bytes;
// ... update status counters
sem_post(sem_stats);  //  EXIT CRITICAL SECTION
```

**Risk:** Without semaphore, race condition on counters when multiple threads update simultaneously

Section 2:

```
// logger.c - logger_log()
// Implicit semaphore via sem_ws_log
fprintf(log_fp, "[%s] %s \"%s %s\" %d %ld\n", ...);
fflush(log_fp);
```

**Risk:** Log entries could interleave without synchronization.

Section 3:

```
// cache.c - cache_put()
pthread_rwlock_wrlock(&cache_rwlock);
if (e->valid && e->data) free(e->data);  // Free old
e->data = malloc(size);                  // Allocate new
memcpy(e->data, data, size);             // Copy data
pthread_rwlock_unlock(&cache_rwlock);
```

**Risk:** Without RW-lock, cache corruption or double-free

## Potential Deadlocks

Scenario 1: Nested Locks

```
// Thread 1:
pthread_rwlock_wrlock(&cache_lock);   // LOCK A
sem_wait(sem_stats);                  // LOCK B

// Thread 2:
sem_wait(sem_stats);                  // LOCK B
pthread_rwlock_wrlock(&cache_lock);   // LOCK A
```

**Analysis:** Potential deadlock if threads acquire locks in different order.

**Solution:** Always acquire locks in consistent order:

1. Process semaphores first (sem_stats, sem_log)
2. Then thread mutexes/RW-locks

Scenario 2: Conditional Wait Timeout

```
// Thread 1:
pthread_rwlock_wrlock(&cache_lock);   // LOCK A
sem_wait(sem_stats);                  // LOCK B

// Thread 2:
sem_wait(sem_stats);                  // LOCK B
pthread_rwlock_wrlock(&cache_lock);   // LOCK A
```

**Prevention:** Timeout prevents permanent deadlock if queue stays full.

### Race Condition Analysis

Race 1: Cache Read While Writing

```
// Thread A (Writer):
free(old_data);        // ← Could free while Thread B reads
malloc(new_size);
memcpy(new_data, ...);

// Thread B (Reader):
char* data = e->data;  // ← Could read freed memory
size_t size = e->size;
```

**Prevention:** RW-lock ensures writers have exclusive access

Race 2: Statistics Counter Increment

```
// Without semaphore:
Thread A: read total (100)
Thread B: read total (100)
Thread A: increment (101)
Thread B: increment (101)  // Lost update!
Final: 101 (should be 102)
```

**Prevention:** Semaphore ensures atomic increment.

Race 3: File Serving with Cache

```
// Thread A: Cache miss, reading file
read(file_fd, file_data, st.st_size);  // Reading
cache_put(path, file_data, size);      // Caching

// Thread B: Same request arrives
cache_get(path, &data, &size);         // Might get partial data
```

**Prevention:** Cache put is atomic under write lock