Sistemas Operativos **Ano lectivo 2025/2026**

Code Templates and Starter Code[1]

Multi-Threaded Web Server Project

Student Guide

This document provides code templates and starter code for the key components of your web server. These templates demonstrate proper structure and synchronization patterns. You should understand every line before using them in your project.

Contents

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

Sistemas Operativos **Ano lectivo 2025/2026**

1. Configuration File Parser

Parse server.conf to load runtime parameters.

```
// config.h
#ifndef CONFIG_H
#define CONFIG_H

typedef struct {
    int port;
    char document_root[256];
    int num_workers;
    int threads_per_worker;
    int max_queue_size;
    char log_file[256];
    int cache_size_mb;
    int timeout_seconds;
} server_config_t;

int load_config(const char* filename, server_config_t* config);

#endif

// config.c
#include "config.h"
#include <stdio.h>
#include <string.h>

int load_config(const char* filename, server_config_t* config) {
    FILE* fp = fopen(filename, "r");
    if (!fp) return -1;

    char line[512], key[128], value[256];
    while (fgets(line, sizeof(line), fp)) {
        if (line[0] == '#' || line[0] == '\n') continue;

        if (sscanf(line, "%[^=]=%s", key, value) == 2) {
            if (strcmp(key, "PORT") == 0) config->port = atoi(value);
            else if (strcmp(key, "NUM_WORKERS") == 0) config->num_workers =
atoi(value);
            else if (strcmp(key, "THREADS_PER_WORKER") == 0)
                config->threads_per_worker = atoi(value);
            else if (strcmp(key, "DOCUMENT_ROOT") == 0)
                strncpy(config->document_root, value, sizeof(config-
>document_root));
            // Add other config parameters...
        }
```

```
        }
        fclose(fp);
        return 0;
    }
```

2. Shared Memory Setup

Create shared memory for connection queue and statistics.

```
// shared_mem.h
#ifndef SHARED_MEM_H
#define SHARED_MEM_H

#define MAX_QUEUE_SIZE 100

typedef struct {
    long total_requests;
    long bytes_transferred;
    long status_200;
    long status_404;
    long status_500;
    int active_connections;
} server_stats_t;

typedef struct {
    int sockets[MAX_QUEUE_SIZE];
    int front;
    int rear;
    int count;
} connection_queue_t;

typedef struct {
    connection_queue_t queue;
    server_stats_t stats;
} shared_data_t;

shared_data_t* create_shared_memory();
void destroy_shared_memory(shared_data_t* data);

#endif

// shared_mem.c
#include "shared_mem.h"
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define SHM_NAME "/webserver_shm"

shared_data_t* create_shared_memory() {
    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
```

```
    if (shm_fd == -1) return NULL;

    if (ftruncate(shm_fd, sizeof(shared_data_t)) == -1) {
        close(shm_fd);
        return NULL;
    }

    shared_data_t* data = mmap(NULL, sizeof(shared_data_t),
        PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    close(shm_fd);

    if (data == MAP_FAILED) return NULL;

    memset(data, 0, sizeof(shared_data_t));
    return data;
}

void destroy_shared_memory(shared_data_t* data) {
    munmap(data, sizeof(shared_data_t));
    shm_unlink(SHM_NAME);
}
```

3. Semaphore Initialization

```
// semaphores.h
#ifndef SEMAPHORES_H
#define SEMAPHORES_H

#include <semaphore.h>

typedef struct {
    sem_t* empty_slots;
    sem_t* filled_slots;
    sem_t* queue_mutex;
    sem_t* stats_mutex;
    sem_t* log_mutex;
} semaphores_t;

int init_semaphores(semaphores_t* sems, int queue_size);
void destroy_semaphores(semaphores_t* sems);

#endif

// semaphores.c
#include "semaphores.h"
#include <fcntl.h>

int init_semaphores(semaphores_t* sems, int queue_size) {
    sems->empty_slots = sem_open("/ws_empty", O_CREAT, 0666, queue_size);
    sems->filled_slots = sem_open("/ws_filled", O_CREAT, 0666, 0);
    sems->queue_mutex = sem_open("/ws_queue_mutex", O_CREAT, 0666, 1);
    sems->stats_mutex = sem_open("/ws_stats_mutex", O_CREAT, 0666, 1);
    sems->log_mutex = sem_open("/ws_log_mutex", O_CREAT, 0666, 1);

    if (sems->empty_slots == SEM_FAILED || sems->filled_slots == SEM_FAILED
||
        sems->queue_mutex == SEM_FAILED || sems->stats_mutex == SEM_FAILED
||
        sems->log_mutex == SEM_FAILED) {
        return -1;
    }
    return 0;
}

void destroy_semaphores(semaphores_t* sems) {
    sem_close(sems->empty_slots);
    sem_close(sems->filled_slots);
    sem_close(sems->queue_mutex);
    sem_close(sems->stats_mutex);
```

```
        sem_close(sems->log_mutex);

        sem_unlink("/ws_empty");
        sem_unlink("/ws_filled");
        sem_unlink("/ws_queue_mutex");
        sem_unlink("/ws_stats_mutex");
        sem_unlink("/ws_log_mutex");
    }
```

4. Master Process Template

```
// master.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

volatile sig_atomic_t keep_running = 1;

void signal_handler(int signum) {
    keep_running = 0;
}

int create_server_socket(int port) {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) return -1;

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

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        close(sockfd);
        return -1;
    }

    if (listen(sockfd, 128) < 0) {
        close(sockfd);
        return -1;
    }

    return sockfd;
}

void enqueue_connection(shared_data_t* data, semaphores_t* sems, int
client_fd) {
    sem_wait(sems->empty_slots);
    sem_wait(sems->queue_mutex);

    data->queue.sockets[data->queue.rear] = client_fd;
```

```
        data->queue.rear = (data->queue.rear + 1) % MAX_QUEUE_SIZE;
        data->queue.count++;

        sem_post(sems->queue_mutex);
        sem_post(sems->filled_slots);
    }
```

5. Thread Pool Implementation

```
// thread_pool.h
#ifndef THREAD_POOL_H
#define THREAD_POOL_H

#include <pthread.h>

typedef struct {
    pthread_t* threads;
    int num_threads;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int shutdown;
} thread_pool_t;

thread_pool_t* create_thread_pool(int num_threads);
void destroy_thread_pool(thread_pool_t* pool);

#endif

// thread_pool.c
#include "thread_pool.h"
#include <stdlib.h>

void* worker_thread(void* arg) {
    thread_pool_t* pool = (thread_pool_t*)arg;

    while (1) {
        pthread_mutex_lock(&pool->mutex);

        while (!pool->shutdown /* && queue is empty */) {
            pthread_cond_wait(&pool->cond, &pool->mutex);
        }

        if (pool->shutdown) {
            pthread_mutex_unlock(&pool->mutex);
            break;
        }

        // Dequeue work item and process

        pthread_mutex_unlock(&pool->mutex);
    }
    return NULL;
}
```

```
thread_pool_t* create_thread_pool(int num_threads) {
    thread_pool_t* pool = malloc(sizeof(thread_pool_t));
    pool->threads = malloc(sizeof(pthread_t) * num_threads);
    pool->num_threads = num_threads;
    pool->shutdown = 0;

    pthread_mutex_init(&pool->mutex, NULL);
    pthread_cond_init(&pool->cond, NULL);

    for (int i = 0; i < num_threads; i++) {
        pthread_create(&pool->threads[i], NULL, worker_thread, pool);
    }

    return pool;
}
```

6. HTTP Request Parser

```
typedef struct {
    char method[16];
    char path[512];
    char version[16];
} http_request_t;

int parse_http_request(const char* buffer, http_request_t* req) {
    char* line_end = strstr(buffer, "\r\n");
    if (!line_end) return -1;

    char first_line[1024];
    size_t len = line_end - buffer;
    strncpy(first_line, buffer, len);
    first_line[len] = '\0';

    if (sscanf(first_line, "%s %s %s", req->method, req->path, req->version)
!= 3) {
        return -1;
    }

    return 0;
}
```

7. HTTP Response Builder

```
void send_http_response(int fd, int status, const char* status_msg,
                        const char* content_type, const char* body, size_t
body_len) {
    char header[2048];
    int header_len = snprintf(header, sizeof(header),
        "HTTP/1.1 %d %s\r\n"
        "Content-Type: %s\r\n"
        "Content-Length: %zu\r\n"
        "Server: ConcurrentHTTP/1.0\r\n"
        "Connection: close\r\n"
        "\r\n",
        status, status_msg, content_type, body_len);

    send(fd, header, header_len, 0);
    if (body && body_len > 0) {
        send(fd, body, body_len, 0);
    }
}
```

8. Thread-Safe Logger

```
void log_request(sem_t* log_sem, const char* client_ip, const char* method,
                 const char* path, int status, size_t bytes) {
    time_t now = time(NULL);
    struct tm* tm_info = localtime(&now);
    char timestamp[64];
    strftime(timestamp, sizeof(timestamp), "%d/%b/%Y:%H:%M:%S %z", tm_info);

    sem_wait(log_sem);
    FILE* log = fopen("access.log", "a");
    if (log) {
        fprintf(log, "%s - - [%s] \"%s %s HTTP/1.1\" %d %zu\n",
                client_ip, timestamp, method, path, status, bytes);
        fclose(log);
    }
    sem_post(log_sem);
}
```

Usage Notes

- These templates provide the structure - you must fill in the details
- Add error checking for ALL system calls
- Implement proper cleanup in signal handlers
- Test each component individually before integration
- Compile with: gcc -Wall -Wextra -pthread -lrt -o server *.c

---

For complete examples, see the network programming tutorial and additional course materials.