Sistemas Operativos - 2025/2026

Trabalho Prático 2

Servidor Web Multi-Threaded com IPC

Autores:

Marcos Costa (NMec: 125882)
José Mendes (NMec: 114429)

Universidade de Aveiro
12 de Dezembro de 2025

Conteúdo

Resumo

Este relatório descreve o desenho e a implementação de um servidor web HTTP/1.1 concorrente em linguagem C. O projeto utiliza uma arquitetura híbrida de múltiplos processos e múltiplas threads para garantir alta disponibilidade e paralelismo. O sistema implementa mecanismos avançados de Sincronização e Comunicação entre Processos (IPC), nomeadamente Semáforos POSIX, Memória Partilhada e passagem de descritores de ficheiro via Sockets Unix. Adicionalmente, foi desenvolvida uma cache LRU *thread-safe* para otimizar o tempo de resposta a pedidos de ficheiros estáticos.

1 Introdução

O objetivo deste projeto foi desenvolver um servidor web robusto capaz de lidar com múltiplas conexões simultâneas, simulando um ambiente de produção real. A solução proposta baseia-se no modelo "Master-Worker", onde um processo mestre gere a aceitação de conexões e delega o processamento a um conjunto fixo de processos trabalhadores, cada um gerindo a sua própria *pool* de threads.

Os principais desafios abordados incluem a sincronização segura de recursos partilhados, a prevenção de *race conditions* e *deadlocks*, e a gestão eficiente de memória.

2 Arquitetura do Sistema

A arquitetura do servidor divide-se em três componentes principais:

2.1 Processo Master

O Master é o ponto de entrada do servidor. A sua responsabilidade é exclusiva: aceitar novas conexões TCP na porta configurada (padrão 8080). Ao contrário de um servidor iterativo simples, o Master não processa o pedido HTTP. Em vez disso, envia o descritor de ficheiro (*socket fd*) do cliente para um dos processos Worker disponíveis.

2.2 Processos Workers e IPC

O sistema inicia N processos Worker (configurável via `server.conf`). A comunicação entre o Master e os Workers é realizada através de:

- **SocketPair (Unix Domain Sockets):** Utilizado para passar fisicamente o descritor de ficheiro do cliente entre processos distintos, utilizando as mensagens de controlo `SCM_RIGHTS`.
- **Semáforos Nomeados:** Controlam o fluxo de trabalho (*backpressure*), garantindo que o Master não envia pedidos se os Workers estiverem saturados.

2.3 Thread Pool

Cada Worker mantém uma *Thread Pool* (padrão 10 threads). As threads operam num modelo Produtor-Consumidor interno:

1. O processo Worker (thread principal) recebe o FD do Master e coloca-o numa fila local.
2. Uma thread livre acorda, retira o FD da fila, processa o pedido HTTP e envia a resposta.

3 Detalhes de Implementação

3.1 Sincronização e Semáforos

A coordenação entre processos é feita através de semáforos POSIX nomeados, inicializados em `semaphores.c`. A Figura 1 ilustra a inicialização dos semáforos.

```c
int initialize_semaphores(semaphores_t * sems, int queue_size){

    // Abre ou cria semáforos nomeados em /dev/shm
    sems->empty_slots = sem_open( name: "/ws_empty",  oflag: O_CREAT, 0666, queue_size);
    sems->filled_slots = sem_open( name: "/ws_filled",  oflag: O_CREAT, 0666, 0);
    sems->queue_mutex = sem_open( name: "/ws_queue_mutex",  oflag: O_CREAT, 0666, 1);
    sems->stats_mutex = sem_open( name: "/ws_stats_mutex",  oflag: O_CREAT, 0666, 1);
    sems->log_mutex = sem_open( name: "/ws_log_mutex",  oflag: 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;
}
```

Figura 1: Excerto de código: Inicialização dos Semáforos POSIX

O par `empty_slots` / `filled_slots` implementa a lógica de buffer limitado, prevenindo sobrecarga do sistema.

3.2 Transferência de Descritores (Socket Passing)

Um dos pontos técnicos mais críticos foi a transferência do socket conectado do processo pai (Master) para o filho (Worker). Como os descritores são locais ao processo, utilizou-se a primitiva `sendmsg` com dados auxiliares (*Ancillary Data*).

```c
int send_fd(int socket, int fd_to_send) {
    struct msghdr msg = {0};
    char dummy = '$'; // Payload obrigatório
    struct iovec iov = { .iov_base = &dummy, .iov_len = 1 };
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;

    char cmsg_buf[CMSG_SPACE(sizeof(int))];
    memset(cmsg_buf, c:0, n:sizeof(cmsg_buf));
    msg.msg_control = cmsg_buf;
    msg.msg_controllen = CMSG_SPACE(sizeof(int));

    struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;
    cmsg->cmsg_len = CMSG_LEN(sizeof(int));
    *(int*)CMSG_DATA(cmsg) = fd_to_send;

    // MSG_NOSIGNAL: Evita crash se worker morrer
    // MSG_DONTWAIT: Retorna erro imediatamente se buffer cheio (EAGAIN)
    if (sendmsg(fd:socket, &msg, flags:MSG_NOSIGNAL | MSG_DONTWAIT) < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            return -2; // Buffer cheio
        }
        perror(s:"sendmsg error");
        return -1;
    }
    return 0;
}
```

Figura 2: Excerto de código: Envio de File Descriptor via SocketPair

3.3 Cache LRU Thread-Safe

Para reduzir a latência de I/O em disco, foi implementada uma cache em memória RAM.

- **Estrutura:** Tabela de Hash para acesso $O(1)$ combinada com uma Lista Duplamente Ligada para gerir a política de expiração LRU (*Least Recently Used*).
- **Concorrência:** Utilização de `pthread_rwlock_t` para permitir leituras paralelas.

3.4 Estatísticas Partilhadas

As estatísticas (Total de Pedidos, Bytes Transferidos, Conexões Ativas) são armazenadas num segmento de Memória Partilhada (`mmap`). O acesso a estes contadores é protegido pelo semáforo `stats_mutex`, garantindo atomicidade nas atualizações vindas de diferentes processos.

4 Desafios e Soluções

4.1 Gestão de Sinais e Processos Zombie

Desafio: Garantir que, ao encerrar o servidor (Ctrl+C), todos os processos filhos e recursos (memória partilhada, semáforos) fossem libertados corretamente.
Solução: Implementação de um *signal handler* para `SIGINT` no `main.c`. O handler envia `SIGTERM` a todos os PIDs dos workers, espera pela sua terminação (`wait`) e executa as funções de limpeza `shm_unlink` e `sem_unlink`.

4.2 Concorrência na Escrita de Logs

Desafio: Múltiplos processos a tentar escrever no ficheiro `access.log` resultavam em linhas intercaladas e corrompidas.
Solução: Introdução de um semáforo dedicado (`log_mutex`). Antes de qualquer escrita no ficheiro, a thread deve adquirir este semáforo, libertando-o logo após o `fflush`.

4.3 Fugas de Memória (Memory Leaks)

Desafio: A cache dinâmica, se mal gerida, consumia memória indefinidamente.
Solução: Validação rigorosa com a ferramenta **Valgrind**. Assegurou-se que a função `evict_entry` liberta corretamente a memória dos nós removidos e que o `destroy_thread_pool` limpa todas as estruturas alocadas antes do fecho do programa.

5 Testes e Validação

5.1 Metodologia

Os testes foram realizados utilizando scripts Bash automatizados e ferramentas padrão de benchmarking:

- **Curl:** Para testes funcionais (verificação de códigos 200, 404, tipos MIME).
- **Apache Bench (ab):** Para testes de carga e stress.
- **Valgrind/Helgrind:** Para verificação de gestão de memória e condições de corrida.

5.2 Resultados dos Testes

5.2.1 Testes Funcionais

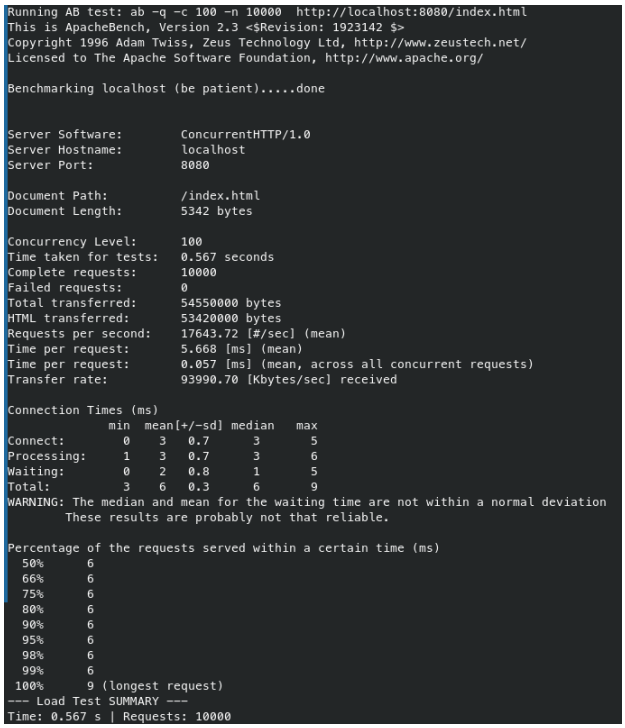
O servidor responde corretamente a pedidos GET e HEAD para ficheiros HTML, CSS, JS e imagens. A Figura 3 demonstra a execução do script de teste funcional com sucesso em todos os casos.

```
❯ ./tests/test_functional.sh
--- Starting Functional Tests ---
[PASS] GET http://localhost:8080/index.html -> Status 200
[PASS] http://localhost:8080/index.html -> Content-Type: text/html
[PASS] HEAD http://localhost:8080/index.html -> Status 200
[PASS] GET http://localhost:8080/ -> Status 200
[PASS] GET http://localhost:8080/test_css_func.css -> Status 200
[PASS] http://localhost:8080/test_css_func.css -> Content-Type: text/css
[PASS] GET http://localhost:8080/test_js_func.js -> Status 200
[PASS] http://localhost:8080/test_js_func.js -> Content-Type: application/javascript
[PASS] GET http://localhost:8080/test_pdf_func.pdf -> Status 200
[PASS] http://localhost:8080/test_pdf_func.pdf -> Content-Type: application/pdf
[PASS] GET http://localhost:8080/nonexistent_file.xyz -> Status 404
--- Test Summary ---
ALL TESTS PASSED
```

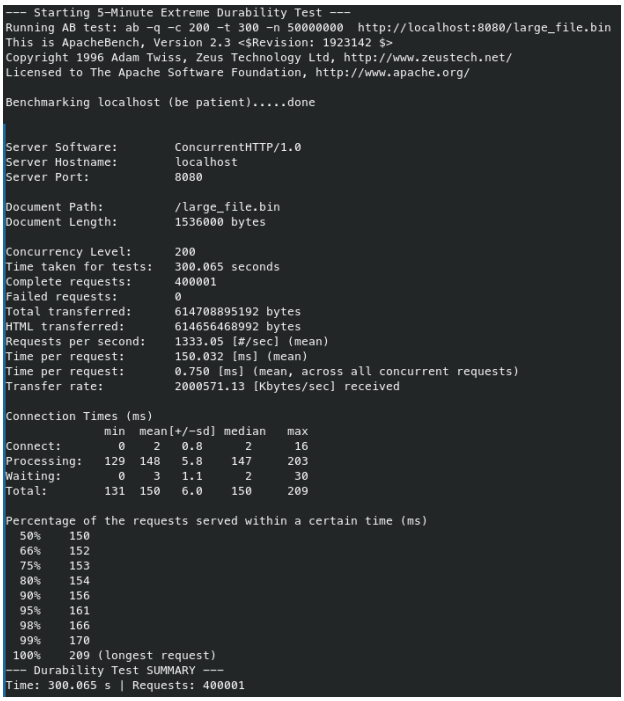
Figura 3: Resultado dos testes funcionais (verificação de status e content-type)

5.2.2 Teste de Carga e Durabilidade

Foram realizados dois testes de carga distintos utilizando o Apache Bench, cujos resultados estão apresentados lado a lado na Figura 4. O primeiro valida a capacidade de resposta sob carga elevada e o segundo testa a durabilidade num período de 5 minutos.



(a) Carga Normal (10k requests)



(b) Durabilidade (5 min)

Figura 4: Resultados dos testes de carga com Apache Bench

5.2.3 Teste de Concorrência Pura

O cliente de teste personalizado (`test_concurrent.c`) lançou 100 threads em paralelo. O output do teste, visível na Figura 5, confirma que não houve perda de pedidos, validando a robustez dos mecanismos de sincronização.

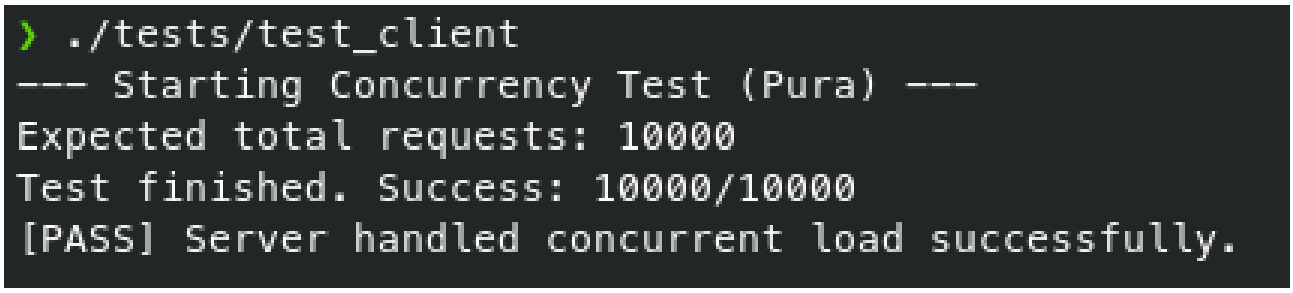


Figura 5: Execução do cliente de teste multithreaded sem perda de pacotes

6 Conclusão

O projeto permitiu consolidar conhecimentos fundamentais de Sistemas Operativos. A implementação de um servidor web funcional exigiu uma compreensão profunda de como os processos interagem e partilham recursos num sistema Linux. A arquitetura escolhida mostrou-se robusta e eficiente. A utilização de cache LRU melhorou significativamente o tempo de resposta para ficheiros estáticos frequentes, e o uso de IPC via `socketpair` provou ser uma solução elegante para a distribuição de carga entre processos.

7 Instruções de Compilação e Comandos Makefile

O projeto está configurado com um `Makefile` completo. Abaixo listam-se os comandos disponíveis:

`make` **ou** `make all`
: Compila o código fonte do servidor e o cliente de teste. Gera o executável `server` e cria a diretoria `www/` com ficheiros de exemplo, se não existirem.

`make run`
: Compila (se necessário) e inicia o servidor imediatamente com a configuração padrão. Pode ser interrompido com `Ctrl+C`.

`make debug`
: Compila o projeto com *flags* de depuração (`-g -DDEBUG`), permitindo o uso de ferramentas como GDB.

`make release`
: Compila o projeto com *flags* de otimização máxima (`-O3`), ideal para medir a performance final ou para testes de carga intensiva.

`make test`
: Executa a suite completa de testes automatizados, que inclui Testes Funcionais, Testes de Carga e Durabilidade, e Testes de Concorrência Pura. Os logs são guardados na pasta `logs/`.

`make valgrind`
: Compila em modo debug e inicia o servidor sob o **Valgrind Memcheck**. Útil para verificar fugas de memória.

`make helgrind`
: Compila em modo debug e inicia o servidor sob o **Valgrind Helgrind**. Essencial para detetar condições de corrida.

`make clean`
: Remove todos os ficheiros objeto (`.o`), os executáveis gerados e ficheiros de log temporários.