Technical Report: Implementação e Desafios do Servidor HTTP

1. Introdução

No âmbito da unidade curricular de **Sistemas Operativos**, foi proposto o desenvolvimento de um **servidor web concorrente**, com suporte para múltiplos processos e múltiplas threads, recorrendo a mecanismos avançados de **sincronização e comunicação entre processos (IPC)**.

O principal objetivo deste projeto é consolidar conhecimentos teóricos adquiridos ao longo do semestre, nomeadamente no que diz respeito à **gestão de processos**, **programação concorrente**, **sincronização**, **memória partilhada** e **comunicação inter-processos**, aplicando-os num cenário realista e próximo de sistemas utilizados em ambiente de produção.

O servidor desenvolvido segue uma arquitetura **master–worker**, onde um processo principal é responsável pela aceitação de ligações e coordenação global do sistema, enquanto vários processos trabalhadores, cada um com um conjunto de threads, tratam concorrentemente os pedidos HTTP dos clientes. Para garantir a consistência e a correta partilha de recursos, foram utilizados **semáforos POSIX**, **mutexes**, **variáveis de condição** e **memória partilhada**.

Este relatório descreve a arquitetura adotada, as principais decisões de implementação, os mecanismos de sincronização utilizados, bem como os testes realizados e os resultados obtidos, analisando ainda os desafios encontrados durante o desenvolvimento do projeto e as soluções aplicadas.

2. Enquadramento Teórico

2.1 Programação Concorrente

A programação concorrente permite a execução simultânea de várias tarefas, sendo essencial em servidores web para responder a múltiplos pedidos em paralelo. Apesar das vantagens em desempenho, exige cuidados adicionais para evitar problemas como condições de corrida e acessos inconsistentes a recursos partilhados.

2.2 Processos vs Threads

Os processos são entidades independentes, com espaços de memória separados, enquanto as threads são unidades de execução mais leves que partilham memória dentro do mesmo processo. Neste projeto, a combinação de processos e threads permite obter simultaneamente isolamento, escalabilidade e maior eficiência no tratamento de pedidos.

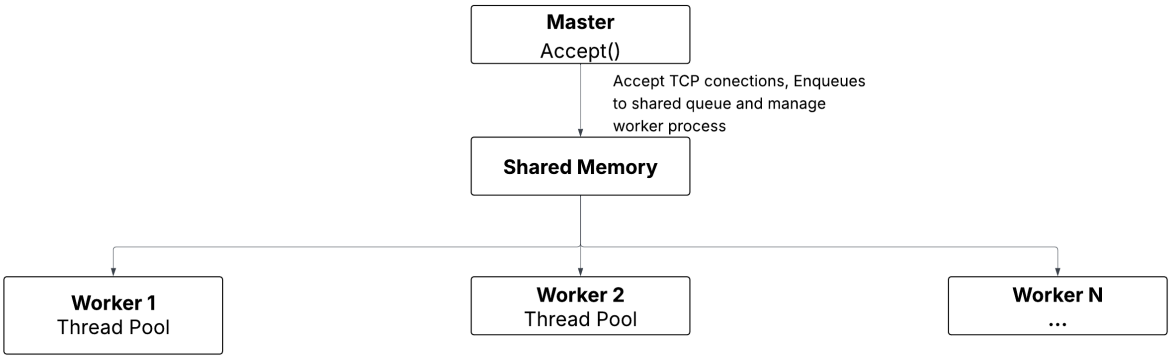
2.3 IPC e Memória Partilhada

A comunicação entre processos (IPC) é necessária para a coordenação do sistema. A memória partilhada foi utilizada por ser um dos mecanismos mais eficientes, permitindo o acesso direto aos dados por múltiplos processos, embora exija sincronização adequada.

2.4 Semáforos, Mutexes e Variáveis de Condição

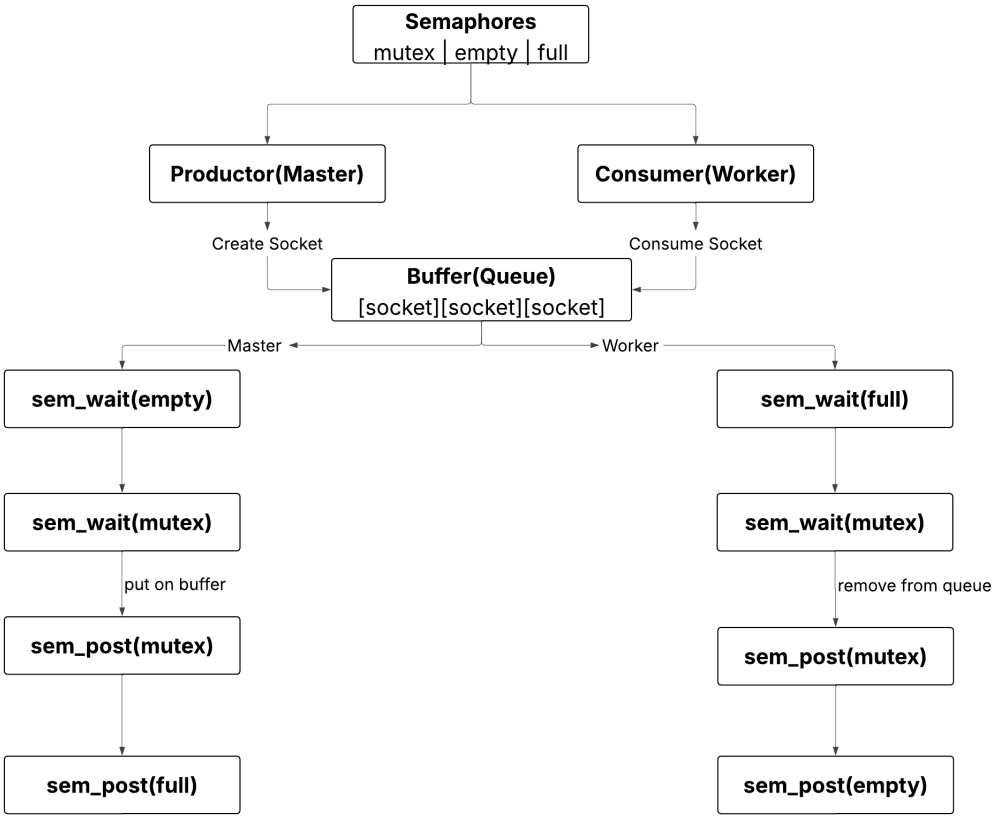
Os semáforos são usados para sincronização entre processos, enquanto os mutexes garantem exclusão mútua entre threads. As variáveis de condição permitem que as threads aguardem por eventos específicos, sendo fundamentais para a implementação de padrões de sincronização como o produtor–consumidor.

3. Arquitetura



A arquitetura é híbrida:

1. **Concorrência Interprocessos (IPC):** Master e Workers comunicam através de Memória Partilhada, protegida por Semáforos POSIX.
2. **Concorrência Intraprocessos (Threads):** Cada Processo Worker mantém um **Pool de Threads** internas (pthread) para servir múltiplos pedidos em paralelo.



3.1 Desafio 1: Corrupção de Dados no Buffer Partilhado (Race Condition)

Problema: Sem a proteção adequada, o Master e os Workers tentavam aceder e modificar os ponteiros in e out do Buffer Circular em Memória Partilhada simultaneamente, resultando na perda de pedidos e dados de buffer inconsistentes. **Solução:** Implementação imediata do semáforo binário (mutex_buffer) para garantir a **Exclusão Mútua** durante qualquer operação de leitura ou escrita no Buffer e nas variáveis de controlo (in e out).

3.2 Desafio 2: Deadlock no Modelo Produtor-Consumidor

Problema: A ordem incorreta das chamadas sem_wait() levava a um deadlock. Se um processo bloqueasse o Mutex antes de um Semáforo de Contagem, poderia prender o Mutex indefinidamente, bloqueando todos os outros processos. **Solução:** Aplicação da regra estrita de sincronização: **Bloquear o semáforo de contagem primeiro, o mutex de exclusão mútua em segundo lugar.** Isto garante que o Mutex só é detido quando um recurso está, de facto, disponível, permitindo o fluxo de trabalho.

3.3 Responsabilidades do Processo Master

O Master Process é o coordenador do sistema, focado em operações de I/O de rede e gestão do IPC. As suas responsabilidades são: inicialização, gestão de processos, produtor IPC.

3.4 Responsabilidades dos Processos Worker

Cada Worker Process é o executor de trabalho. É um processo idêntico que compete com os outros Workers pelo trabalho na fila IPC. As suas responsabilidades são: Inicialização Interna, consumidor IPC, Delegação de Tarefas, Processamento (via Thread Pool)**.**

4. Implementação

4.1 Gestão de Processos

O servidor segue uma arquitetura **master–worker**, na qual um processo principal é responsável pela inicialização do sistema, criação do socket e aceitação das ligações de entrada. Após esta fase, o servidor cria um conjunto fixo de processos worker através da chamada de sistema fork().

Cada processo worker trata concorrentemente os pedidos HTTP recebidos. O servidor implementa ainda o tratamento de sinais, nomeadamente SIGINT e SIGTERM, de forma a permitir um encerramento controlado, garantindo a terminação dos processos worker e a libertação correta dos recursos de IPC utilizados.

4.2 Gestão de Threads

Em cada processo worker, o servidor cria um **thread pool** com um número fixo de threads definido no ficheiro de configuração. Esta abordagem evita a sobrecarga associada à criação dinâmica de threads e melhora o desempenho sob carga elevada.

O servidor utiliza o modelo **produtor–consumidor**, no qual o processo master atua como produtor ao inserir pedidos na fila partilhada, enquanto as threads dos processos worker atuam como consumidoras. A **fila de pedidos** é implementada como um buffer circular limitado, permitindo controlar o número máximo de ligações pendentes e prevenindo a saturação do servidor.

4.3 Comunicação e Sincronização

A comunicação entre os diferentes processos do servidor é realizada através de **memória partilhada**, permitindo a troca eficiente de informação, como pedidos e estatísticas globais.

Para garantir a correta sincronização no acesso à memória partilhada, o servidor recorre a **semáforos POSIX**, assegurando a exclusão mútua entre processos. Ao nível das threads, são utilizados **mutexes** e **variáveis de condição** para proteger regiões críticas e coordenar o acesso à fila de pedidos. Estes mecanismos permitem prevenir **condições de corrida**, garantindo a consistência dos dados e o funcionamento correto do servidor em ambientes concorrentes.

4.4 Servidor HTTP

O servidor implementa funcionalidades essenciais do protocolo **HTTP/1.1**, suportando os métodos **GET** e **HEAD**. O tratamento dos pedidos inclui a validação do caminho solicitado, o acesso aos ficheiros no diretório configurado e a construção da resposta HTTP adequada.

O servidor devolve os códigos de estado mais comuns, como **200 OK**, **403 Forbidden**, **404 Not Found**, **405 Method Not Allowed** e **500 Internal Server Error**, bem como os respetivos cabeçalhos HTTP, incluindo **Content-Type**, **Content-Length**, **Date** e **Server**.

4.5 Funcionalidades Adicionais

Para melhorar o desempenho, o servidor implementa uma **cache de ficheiros**, permitindo servir conteúdos frequentemente acedidos sem necessidade de leituras repetidas do sistema de ficheiros. O acesso à cache é sincronizado para garantir a sua utilização segura por múltiplas threads.

O servidor inclui ainda um sistema de **logging thread-safe**, onde todos os pedidos são registados de forma consistente, evitando o interleaving de mensagens no ficheiro de log. Adicionalmente, é implementado um mecanismo de **rotação de ficheiros de log** que, ao atingir o tamanho máximo de **10 MB**, procede à **renomeação do ficheiro de log ativo**, incorporando um **timestamp correspondente ao instante da rotação**, e cria de seguida um novo ficheiro de log para continuar o registo dos eventos.

Adicionalmente, são mantidas **estatísticas partilhadas**, acessíveis por todos os processos, permitindo monitorizar o funcionamento do servidor, nomeadamente o número de pedidos tratados e os códigos de estado devolvidos. Para a implementação desta funcionalidade, foi criado um **processo filho dedicado**, cujo único objetivo é **aceder periodicamente às estatísticas partilhadas e apresentar a informação recolhida**, permitindo **desacoplar a monitorização do processamento dos pedidos**, reduzindo o impacto no desempenho e aumentando a clareza da arquitetura do servidor.

5. Testes Realizados

5.1 Testes Funcionais

Os testes funcionais tiveram como objetivo validar o correto funcionamento do servidor relativamente aos requisitos do protocolo HTTP. Foram testados os métodos **GET** e **HEAD**, o acesso a diferentes tipos de ficheiros estáticos e o comportamento do servidor perante pedidos válidos e inválidos.

Foram igualmente verificados os **códigos de estado HTTP** devolvidos, nomeadamente **200 OK**, **403 Forbidden**, **404 Not Found**, **405 Method Not Allowed** e **500 Internal Server Error**, bem como a correta geração dos cabeçalhos HTTP. Estes testes permitiram confirmar que o servidor responde corretamente em condições normais de utilização.

5.2 Testes de Concorrência

Os testes de concorrência tiveram como objetivo avaliar o comportamento do servidor perante múltiplos pedidos simultâneos. Foram realizados testes com vários clientes a aceder ao servidor em paralelo, de forma a verificar a correta sincronização entre processos e threads.

Estes testes permitiram confirmar a ausência de condições de corrida visíveis, bem como a integridade da fila de pedidos, do sistema de logging e das estatísticas partilhadas, mesmo sob acesso concorrente intensivo.

5.3 Testes de Stress

Os testes de stress foram realizados com o intuito de avaliar a estabilidade do servidor sob carga elevada e durante períodos prolongados de execução. O servidor foi submetido a um elevado número de pedidos consecutivos e simultâneos, permitindo observar o seu comportamento em situações próximas de um cenário real.

Durante estes testes, foi verificada a capacidade do servidor para continuar a responder corretamente sem falhas, bem como o correto funcionamento dos mecanismos de gestão de recursos e de encerramento controlado.

5.4 Ferramentas Utilizadas

Para a realização dos testes e validação do servidor, foram utilizadas diversas ferramentas, entre as quais:

- **curl** e **wget**, para testes manuais e verificação das respostas HTTP;
- **Apache Bench (ab)**, para testes de carga e concorrência com múltiplos pedidos simultâneos;
- **Valgrind**, para deteção de fugas de memória e validação da correta gestão de memória;

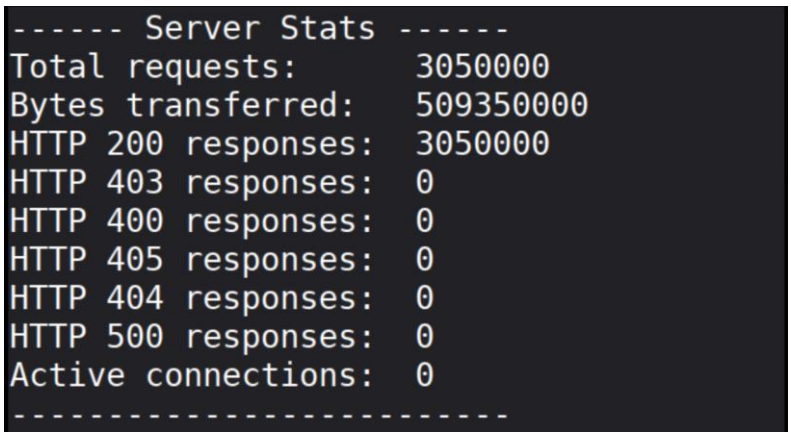
- **Helgrind/ThreadSanitizer**, para identificação de possíveis problemas de sincronização entre threads.

A utilização destas ferramentas permitiu validar de forma sistemática a correção, robustez e desempenho do servidor.

6. Resultados e Análise

6.1 Desempenho sob Carga

O servidor demonstrou um comportamento estável e consistente quando submetido a cargas elevadas, respondendo corretamente a um grande número de pedidos simultâneos sem falhas funcionais. Os testes de carga realizados evidenciaram que o modelo adotado permite manter tempos de resposta aceitáveis mesmo sob níveis elevados de concorrência.



A Figura apresenta os resultados obtidos durante os testes de desempenho, ilustrando a relação entre o número de pedidos, a concorrência e o tempo médio de resposta.

6.2 Escalabilidade

A arquitetura master–worker, aliada à utilização de múltiplos processos e threads, permite ao servidor escalar de forma eficiente com o aumento do número de pedidos concorrentes. A separação entre o processo master e os processos worker facilita a distribuição do trabalho e o aproveitamento de sistemas multi-core.

6.3 Comportamento Concorrente

Durante os testes de concorrência, o servidor apresentou um comportamento correto, não sendo detetadas condições de corrida, inconsistências nos dados partilhados ou corrupção do ficheiro de log. O uso adequado de semáforos, mutexes e variáveis de condição garantiu a sincronização correta entre processos e threads.

Adicionalmente, as estatísticas partilhadas refletiram de forma consistente o número real de pedidos processados e os respetivos códigos de estado devolvidos, confirmando a robustez dos mecanismos de sincronização implementados.

6.4 Limitações Observadas

Apesar dos resultados positivos obtidos, foi identificada uma limitação no que diz respeito ao encerramento controlado do servidor (graceful shutdown). Em determinadas situações, o servidor não consegue terminar todas as threads e processos de forma completamente sincronizada, podendo deixar pedidos pendentes ou exigir uma terminação forçada.

Esta limitação não compromete o funcionamento normal do servidor durante a sua execução, mas representa um aspeto a melhorar em trabalhos futuros, nomeadamente ao nível da coordenação entre processos e da sinalização do encerramento às threads ativas.

7. Conclusão

O desenvolvimento deste projeto permitiu atingir os principais objetivos propostos, nomeadamente a implementação de um servidor web concorrente baseado numa arquitetura master–worker, com suporte para múltiplos processos e múltiplas threads. Ao longo do trabalho, foram aplicados de forma prática conceitos fundamentais de Sistemas Operativos, como programação concorrente, sincronização, comunicação entre processos e gestão de recursos, resultando num servidor funcional e robusto.

De uma forma geral, o servidor apresentou um comportamento correto, tendo passado com sucesso a maioria dos testes funcionais, de concorrência e de stress realizados. A utilização de mecanismos de sincronização adequados garantiu a consistência dos dados partilhados, a integridade do sistema de logging e o correto funcionamento das estatísticas, mesmo sob cargas elevadas. Apesar de existirem limitações no que diz respeito ao graceful shutdown, o servidor demonstrou estabilidade durante a sua execução normal.

Como trabalho futuro, poderão ser introduzidas melhorias ao nível do encerramento controlado do servidor, assegurando a terminação coordenada de todas as threads e processos ativos. Adicionalmente, seria possível expandir o servidor com funcionalidades extra, como suporte para ligações persistentes (HTTP Keep-Alive), otimizações adicionais na cache de ficheiros ou mecanismos de monitorização mais avançados, contribuindo para um aumento do desempenho e da robustez do sistema.