Design Document: Arquitetura do Servidor HTTP Concorrente

Autores:

121420 - Kelvin Loforte

127560 - Mario Santos

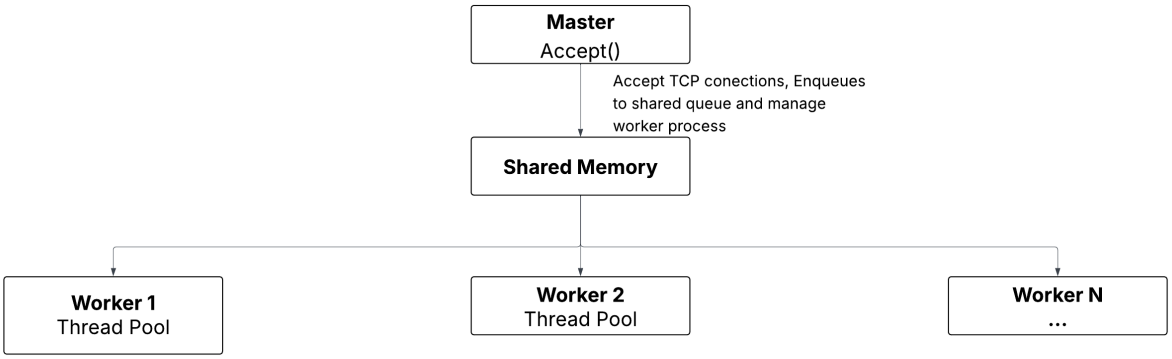
1. Introdução

Este documento detalha o design e a arquitetura do ConcurrentHTTP Server, um servidor web multi-processos e multi-threaded construído para demonstrar a gestão de concorrência e a comunicação interprocessos (IPC) em ambientes POSIX.

2. Arquitetura do Sistema: Master-Worker

2.1 Visão Geral da Arquitetura

O servidor adota um modelo **Master-Worker**, onde um processo principal (Master) gere as conexões de entrada, e um conjunto de processos filhos (Workers) executa o processamento das requisições.



A arquitetura Master-Worker, onde o Processo Master está isolado do Processamento de Requisições. O canal de comunicação é a Fila IPC (Buffer Circular) em Memória Partilhada. Cada Worker Process é um Consumidor que alimenta a sua Thread Pool interna, realizando o file serving e logging. Os Semáforos são indicados como mecanismos de proteção para o acesso à Fila IPC e ao access.log.

Ou seja, 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.

2.2 Fluxo de Requisição (Request Flow)

1. O **Master Process** executa o listen() e o accept(), recebendo um novo *socket* cliente.
2. O Master, atuando como **Produtor**, insere o *file descriptor* do *socket* cliente no Buffer Circular partilhado.
3. Um dos **Worker Processes** retira o *file descriptor* do Buffer, atuando como **Consumidor**.
4. O Worker delega o processamento HTTP a uma thread livre no seu **Thread Pool**.

5. A Thread processa o pedido (parsing, leitura de ficheiro, caching) e envia a resposta HTTP.

2.3 Detalhes Internos do Buffer Circular (expandir)

O Buffer Circular é armazenado numa região de Memória Partilhada POSIX. A estrutura contém:

- int fds[MAX_QUEUE_SIZE] - elementos;
- unsigned int head - índice de remoção;
- unsigned int tail - índice de inserção;
- unsigned int count - contador auxiliar (opcional).

A operação wrap-around utiliza aritmética modular ((index + 1) % MAX_QUEUE_SIZE).

2.4 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:

1. **Inicialização:**
   - Carregar a configuração (server.conf).
   - Configurar a Memória Partilhada para o Buffer Circular e Estatísticas.
   - Inicializar e ligar os Semáforos POSIX nomeados (mutex, empty, full, etc.).
   - Criar e ligar (bind) o listening socket (create_server_socket).

2. **Gestão de Processos:**
   - Criar o número configurado de Worker Processes (fork).
   - Implementar o signal handler para SIGINT (Ctrl+C), garantindo o graceful shutdown.
   - Recolher os Workers (waitpid) para evitar Processos Zumbis.
3. **Produtor IPC:**
   - Executar o ciclo principal (while(true)) onde bloqueia em accept() à espera de novas conexões.
   - Atuar como Produtor no Buffer Circular: Inserir o file descriptor do cliente na fila partilhada, respeitando a lógica de sincronização Produtor-Consumidor (i.e., aguardar por espaço livre).

2.5 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:

1. **Inicialização Interna:**
   - Anexar-se aos recursos IPC (Memória Partilhada e Semáforos).
   - Inicializar a sua Thread Pool interna (criando e colocando as worker threads em espera).
   - Inicializar estruturas de dados locais, como a Cache de Ficheiros LRU.

2. **Consumidor IPC:**
   - Atuar como Consumidor no Buffer Circular: Bloquear (sem_wait) até que haja um socket descriptor disponível na fila.
   - Remover o file descriptor da fila, respeitando a lógica de sincronização.

3. **Delegação de Tarefas:**
   - Após remover o file descriptor, delegar a tarefa (tratar o client_fd) a uma worker thread livre na sua pool.

4. **Processamento (via Thread Pool):**
   - A worker thread é responsável pelo processamento HTTP (ler, parsing, file serving, responder, atualizar estatísticas, logging).

3. Design do IPC: Buffer Circular e Sincronização

3.1 Recurso Partilhado (Memória Partilhada)

O canal de comunicação é um **Buffer Circular** de tamanho fixo (definido em server.conf) armazenado em **Memória Partilhada POSIX** (shm_open, mmap). Este buffer armazena os *inteiros* que representam os *file descriptors* dos sockets de clientes.

3.2 Implementação do Produtor-Consumidor com Semáforos

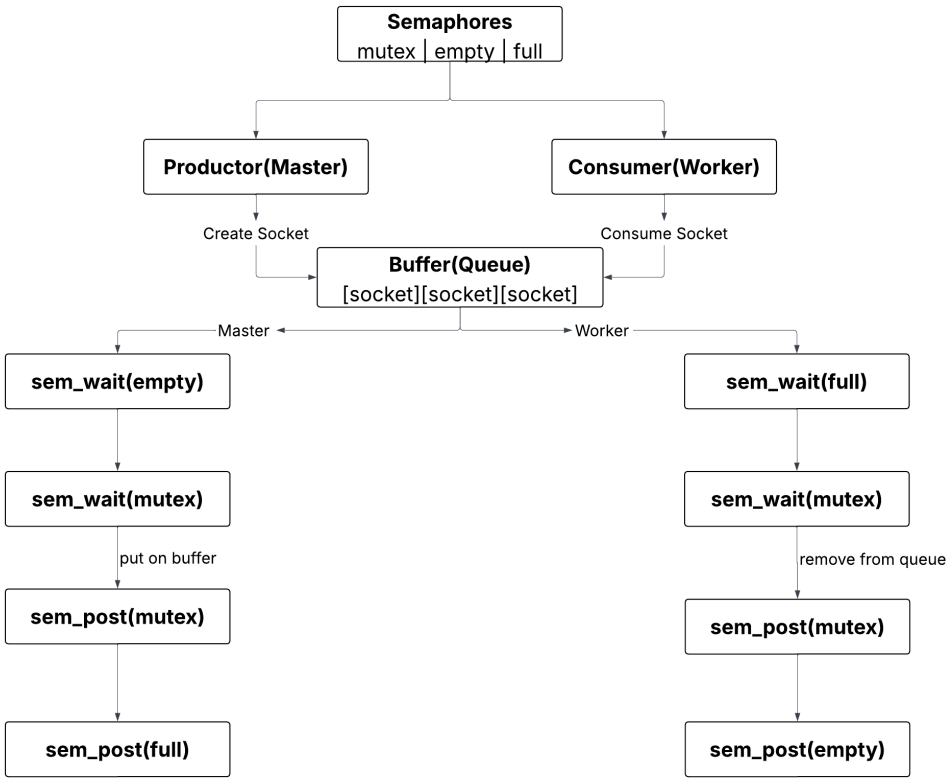
Três Semáforos POSIX (nomeados) são utilizados para resolver o problema do Produtor-Consumidor e proteger a Fila partilhada:

Semáforo	Tipo	Inicialização	Função

mutex_buffer	Binário (Mutex)	1	**Exclusão Mútua:** Garante acesso exclusivo ao Buffer.
sem_empty	Contagem	MAX_QUEUE_SIZE	**Controle do Produtor:** Bloqueia o Master se a fila estiver cheia.
sem_full	Contagem	0	**Controle do Consumidor:** Bloqueia os Workers se a fila estiver vazia.

Ordem Crucial de Operações (para prevenir Deadlock):

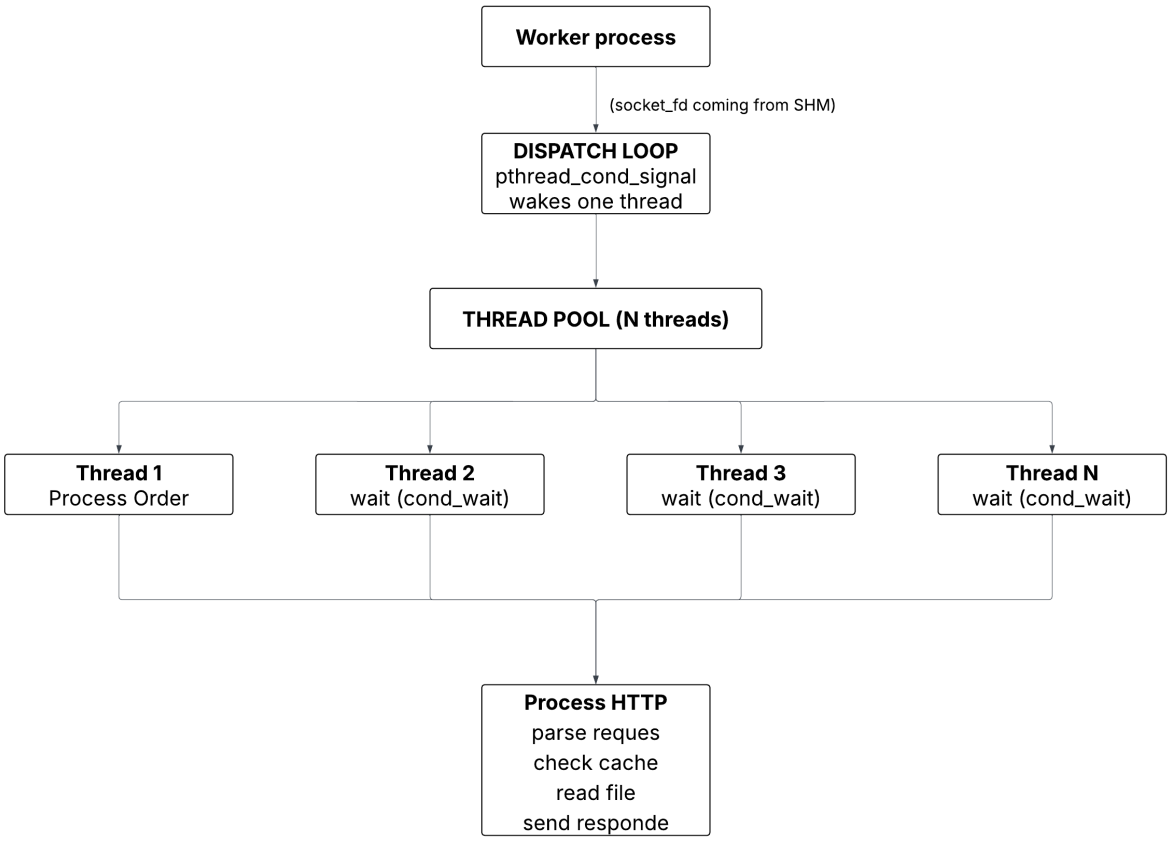
- **Master (Produtor):** sem_wait(empty) -> sem_wait(mutex) -> insert() -> sem_post(mutex) -> sem_post(full)
- **Worker (Consumidor):** sem_wait(full) -> sem_wait(mutex) -> remove() -> sem_post(mutex) -> sem_post(empty)



3.3 Tabela de Estados do Semáforo

Situação	empty	full	mutex	Descrição
Queue vazia	MAC	0	1	Workers bloqueiam em sem_wait(full)
Queue cheia	0	MAX	1	Master bloqueia em sem_wait(empty)
Uso normal	>0	>0	1	Master e Worker fluem corretamentw

4. Design da Concorrência (Thread Pools)



Cada Worker Process utiliza um Thread Pool (tamanho definido em server.conf) para maximizar a reutilização de recursos.

- **Sincronização Interna:** O Worker utiliza **Mutexes** e **Variáveis de Condição (pthread_cond_t)** para gerir o seu pool. Quando uma thread termina um pedido, espera numa Variável de Condição; quando chega um novo pedido do IPC, o Worker Process envia um pthread_cond_signal() para acordar uma thread.

5. Caching e Logging

O servidor implementa uma **Cache de Ficheiros** na memória de cada Worker para reduzir a latência de acesso ao disco. O acesso à estrutura de dados da cache é protegido por um **Mutex** local ao Worker. O sistema de logging também usa um **Semáforo** para garantir que apenas um processo/thread escreve no ficheiro access.log de cada vez (Thread-Safe Logger).