Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Operativos

Ano Letivo de 2024/2025

Sistemas Operativos

Manuel Fernandes
A93213
May 16, 2025

Nelson Mendes
A106884

SO

Índice

1. Introdução 1

2. Desenvolvimento do Trabalho Prático 2

2.1. Arquitetura 3

2.2. Document e Message 4

2.3. DCliente 5

2.4. DServer 6

3. Scripts 8

4. Conclusão 9

1. Introdução

Nos sistemas operativos modernos, a gestão eficiente de documentos e a capacidade de realizar pesquisas rápidas e concorrentes são fundamentais para otimizar o fluxo de trabalho e a produtividade. Neste contexto, o presente trabalho prático tem como objetivo desenvolver um **serviço de indexação e pesquisa de documentos**, composto por um programa servidor (*dserver*) e um programa cliente (*dclient*), que permita aos utilizadores gerir meta-informação de documentos e realizar pesquisas sobre o seu conteúdo.

O servidor é responsável por armazenar e persistir a meta-informação dos documentos, enquanto o cliente facilita a interação do utilizador através de operações como indexação, consulta, remoção e pesquisa. Adicionalmente, o servidor incorpora funcionalidades avançadas, como **pesquisa concorrente, persistência de dados e *caching***, para melhorar o desempenho e a eficiência do sistema. A implementação recorre a mecanismos de comunicação entre processos, como *pipes* com nome, e utiliza chamadas ao sistema em C, garantindo conformidade com os requisitos do projeto.

Este relatório descreve a arquitetura do sistema, as escolhas de implementação, os desafios enfrentados e os resultados obtidos, com ênfase na avaliação experimental das otimizações realizadas. Através deste trabalho, pretende-se demonstrar a aplicação prática de conceitos fundamentais de Sistemas Operativos, como **gestão de processos, comunicação interprocessos e manipulação de ficheiros**.

2. Desenvolvimento do Trabalho Prático

O desenvolvimento deste projeto centrou-se na implementação de um serviço distribuído de indexação e pesquisa de documentos, composto por um servidor (*dserver*) e um cliente (*dclient*), que comunicam através de *pipes* com nome. Esta secção descreve a arquitetura do sistema, as estruturas de dados utilizadas, os mecanismos de comunicação entre processos e as estratégias adotadas para garantir concorrência, persistência e eficiência no acesso aos dados.

A implementação seguiu os requisitos do enunciado, privilegiando o uso de chamadas ao sistema em C (e.g., *open*, *read*, *write*, *mkfifo*) em detrimento de funções de alto nível (e.g., *fopen*, *fread*), de modo a assegurar um controlo mais granular sobre operações de I/O e gestão de processos. O servidor foi concebido para manter em memória e em disco a meta-informação dos documentos, suportando operações como **indexação, consulta, remoção e pesquisa**, enquanto o cliente atua como interface para interação do utilizador.

Além das funcionalidades básicas, foram implementadas otimizações como:

- **Pesquisa concorrente**, permitindo que múltiplos processos analisem documentos em paralelo para acelerar operações de pesquisa;
- **Persistência** em disco, garantindo a recuperação dos dados após reinício do servidor;
- ***Caching*** em memória, reduzindo o tempo de acesso a meta-informação frequentemente requisitada.

Nesta secção, detalham-se as decisões de implementação, os desafios enfrentados e as soluções adotadas, bem como a integração dos diferentes componentes do sistema. A avaliação experimental, apresentada posteriormente, valida o desempenho das otimizações introduzidas.

2.1. Arquitetura

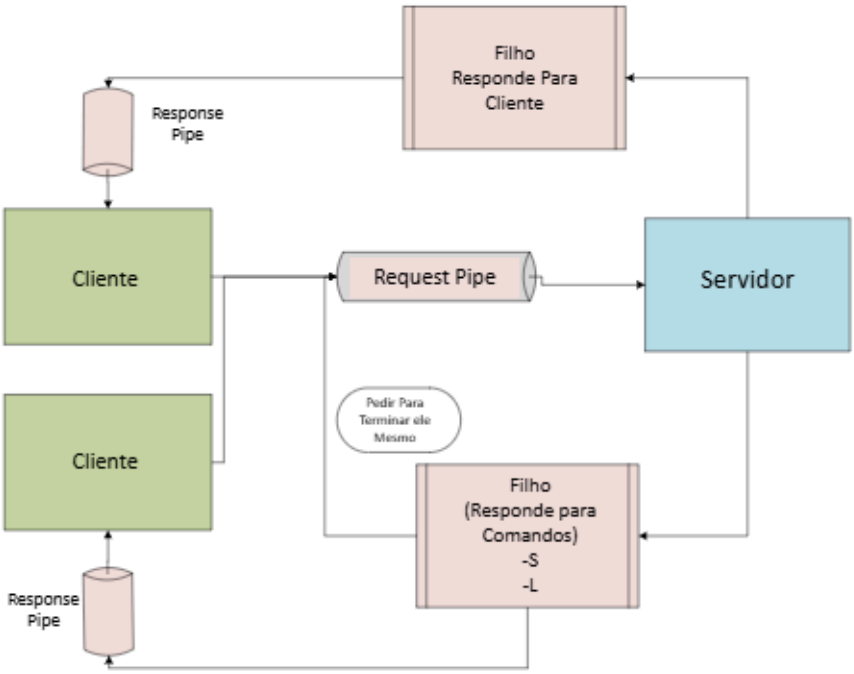

Figura 1: Visualização Prática da Arquitetura do Servidor

A arquitetura do servidor apresentada baseia-se na comunicação entre processos utilizando *pipes* nomeados (FIFOs). Múltiplos clientes comunicam-se com um servidor central através de um *pipe* de requisição compartilhado (*Request Pipe*), por onde os comandos são enviados. Cada cliente também possui o seu próprio *pipe* de resposta (*Response Pipe*), exclusivo para receber as respostas do servidor.

Quando o servidor recebe uma requisição via *Request Pipe*, ele delega o processamento a um processo filho. **Este filho pode ser responsável por executar comandos específicos** ou por enviar a resposta ao cliente através do *pipe* de resposta. Esta divisão permite que o servidor continue a receber novas requisições, enquanto os filhos tratam os pedidos paralelamente.

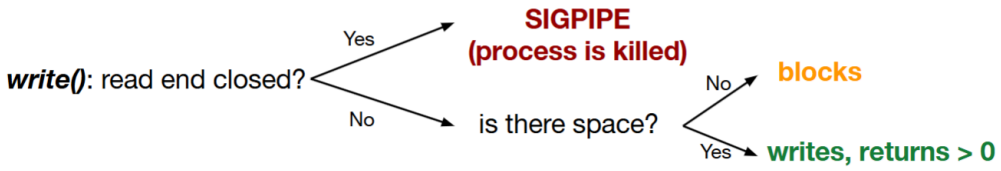

Figura 2: Visualização Prática do Processo do *Read-End*

Um ponto importante da arquitetura está relacionado com o **tratamento da função *write()* nos *pipes***. Se o cliente encerrar o seu processo e, com isso, fechar o *read-end* do *pipe*, qualquer tentativa de escrita feita por um processo que ainda mantém o *write--end* aberto resultará num erro SIGPIPE, que normalmente finaliza o processo escritor. No entanto, como a arquitetura isola este envio de resposta nos processos filhos, **o processo afetado será apenas no filho que tentou escrever no *pipe* fechado**. O servidor principal não é afetado, continuando em execução, mantendo a robustez do sistema frente à terminação inesperada de clientes.

Este comportamento garante que o servidor não é encerrado indevidamente por falhas na comunicação com os clientes, mantendo a resiliência e estabilidade do sistema.

2.2. Document e Message

Os ficheiros "*Document.c*", "*Document.h*" e "*Message.h*" representam as **funções e estrutura base** que serão utilizadas pelo servidor e pelo cliente. A necessidade de criar cada uma destas estruturas veio da forma como o servidor e o cliente operam. O cliente gera uma Mensagem, que será importada pelo servidor. Tendo em conta a *msg.action*, este irá criar, eliminar ou manipular Documentos.

A estrutura ***Document***, definida em "*Document.h*", serve como núcleo da representação de metadados no sistema, encapsulando todos os atributos essenciais de um documento (ID, título, autores, ano e caminho do ficheiro). A escolha de *arrays* de caracteres com tamanhos fixos, em vez de ponteiros para *strings* dinâmicas, foi deliberada para simplificar a serialização das mensagens e garantir limites bem definidos para transferência entre processos. As funções *document_new* e *document_free* seguem o padrão clássico de construtor/destruir de objetos em C, enquanto *document_print* oferece **funcionalidade básica de *debug***. A validação dos comprimentos das *strings* é feita através de *strncpy* para prevenir *buffer overflows*.

O sistema de comunicação está centralizado na estrutura ***Message***, que implementa um protocolo versátil para todas as interações cliente-servidor. A estrutura da mensagem utiliza um campo *action* como discriminador para operações (adicionar, consultar, apagar, pesquisar, etc.), seguindo o padrão *command*. Esta abordagem permite que um **único tipo de mensagem suporte múltiplos comandos**, onde os campos adicionais

são interpretados conforme a ação especificada. A inclusão de *fifo_name* em cada mensagem possibilita comunicação bidirecional assíncrona, onde o servidor pode enviar respostas para um FIFO específico criado pelo cliente, permitindo **múltiplos clientes a operar concorrentemente**.

2.3. DCliente

O cliente inicia criando um FIFO exclusivo para receber respostas do servidor, utilizando o PID do processo como identificador único. Esta abordagem garante que **múltiplas instâncias do cliente possam operar concorrentemente sem conflitos**, resolvendo o problema de multiplexação de respostas. O nome do FIFO é formatado como:

```
/tmp/client <PID>
```

seguindo convenções padrão para arquivos temporários em sistemas Unix. A criação do *pipe* com permissões 0666 permite acesso flexível.

A estrutura *Message* serve como envelope universal para todas as comunicações, encapsulando tanto o comando quanto os dados associados. O preenchimento dos campos é condicionado pelos argumentos de linha de comando, com validação básica do número de parâmetros para cada ação. Para operações como **adição de documentos (-a)**, os campos *title*, *authors*, *year* e *path* são copiados com *strncpy* para garantir que não excedam os tamanhos máximos definidos na estrutura. Operações como **consulta (-c) e eliminação (-d)** requerem apenas o ID do documento, enquanto **pesquisas (-s)** aceitam um parâmetro opcional *n_proc* para controlar o paralelismo no servidor.

A comunicação propriamente dita ocorre em duas fases distintas: primeiro o cliente envia a sua mensagem para o **FIFO do servidor** (*SERVER_PIPE*) em modo bloqueante (*O_WRONLY*), garantindo que a escrita seja atómica. De seguida, **abre o seu FIFO privado em modo de leitura** (*O_RDONLY*) para aguardar a resposta. Este padrão *request-reply* é clássico em sistemas de mensagens assíncronas e permite que o servidor processe pedidos no seu próprio ritmo, enquanto o cliente bloqueia apenas durante a operação de leitura da resposta.

Um aspeto importante a abordar é o **tratamento do ciclo de leitura da resposta**, que processa dados em blocos de *BUFF_SIZE* até que *read* retorne 0 (indicando fim de transmissão). Isto é fundamental para lidar com respostas potencialmente grandes, como resultados de pesquisas em documentos extensos. O uso de *write(1, buffer, n_read)* direciona a saída diretamente para o *stdout*, **facilitando a integração com *pipelines* Unix e redirecionamentos**.

Antes de terminar, o cliente remove o seu FIFO específico através de *unlink()*, seguindo boas práticas de gestão de recursos temporários. Esta limpeza **previne o acúmulo de arquivos não utilizados** no sistema de arquivos, mesmo em casos de terminação abrupta do programa.

A arquitetura geral apresenta várias vantagens:

- **escalabilidade natural** devido à comunicação assíncrona;
- **isolamento entre clientes** através de FIFOs dedicados;
- **flexibilidade** para suportar diferentes operações através de um protocolo unificado.

A principal limitação é a dependência do sistema de arquivos para comunicação, o que pode se tornar num obstáculo em ambientes de alta carga, mesmo sendo **perfeitamente adequado para muitos cenários práticos** de tamanho moderado.

2.4. DServer

O servidor inicia com a criação de uma tabela de *hash* global, utilizando a biblioteca GLib, que serve como repositório principal para todos os documentos indexados. Esta escolha permite operações eficientes de inserção, consulta e remoção, **características fundamentais para um sistema que necessita de responder rapidamente** a solicitações concorrentes.

A comunicação entre clientes e servidor é mediada por um ***pipe* que funciona como ponto de entrada** para todos os pedidos. Quando um cliente envia uma mensagem, o servidor processa-a de acordo com o tipo de operação solicitada, que pode variar desde uma simples adição ou consulta de um documento até operações mais complexas como pesquisas de conteúdo ou contagem de ocorrências de palavras-chave em ficheiros específicos. Um aspeto interessante é que **cada cliente cria o seu próprio *pipe* para receber respostas**, permitindo que o servidor envie informações de volta de forma direcionada, sem risco de interferência entre comunicações paralelas.

Para operações que envolvem processamento intensivo, como pesquisas em múltiplos documentos, **o servidor recorre à criação de processos filhos que executam em paralelo**, distribuindo assim a carga de trabalho. Esta abordagem demonstra uma preocupação com a eficiência do sistema, especialmente quando lidamos com volumes elevados de dados ou operações que beneficiem de concorrência. A gestão destes processos é feita de forma controlada, com o **servidor a coordenar a execução e a agregar os resultados** antes de enviar a resposta final ao cliente.

O ciclo de vida do servidor é contínuo, executando num *loop* infinito até receber um comando específico de terminação. Durante o seu funcionamento, **mantém abertos os canais de comunicação** necessários, assegurando que nenhum pedido fica por atender, devido a condições de corrida ou outros problemas típicos de sistemas concorrentes. Quando finalmente encerrado, o servidor liberta todos os recursos alocados, incluindo a eliminação do *pipe* e a libertação da memória ocupada pela tabela de documentos.

Algo bastante importante de referenciar é que enquanto o servidor aguarda por um pedido no *pipe* principal, o processo fica bloqueado na chamada:

```
read(fifo, &msg, sizeof(Message))
```

evitando assim esperas ativas que consumiriam recursos desnecessários. Quando um cliente envia um pedido, o servidor é desbloqueado, lê a mensagem e processa a ação correspondente, garantindo uma resposta eficiente. Esta abordagem permite um **comportamento reativo sem sobrecarga do CPU**, uma vez que o sistema operativo suspende o processo até que dados estejam disponíveis para leitura. **No final do processamento, o servidor volta a bloquear**, aguardando novos pedidos de forma otimizada.

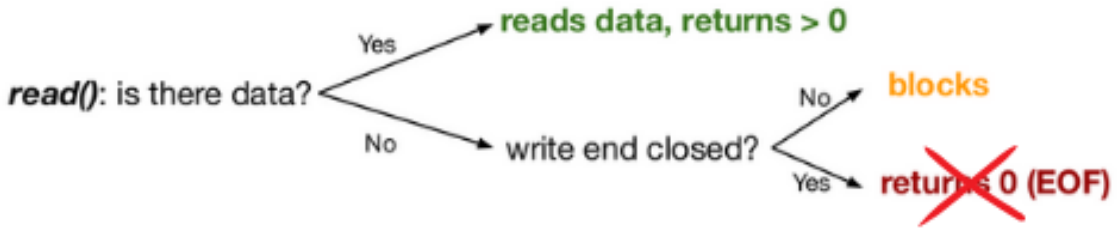

Figura 3: Visualização Prática do Processo de Leitura do *DServer*

Na implementação atual, a linha:

```
int fifo_w = open(PIPE_NAME, O_WRONLY)
```

mantém um *write end* aberto no próprio servidor, funcionando como um "*write* fictício", isto é, uma extremidade de escrita que nunca será utilizada. **Isto evita que o *read()* retorne EOF inadvertidamente**, garantindo que o servidor permaneça bloqueado até receber pedidos válidos. A **cruz na figura destaca o caminho a ser evitado** (espera ativa), enquanto a solução adotada assegura um comportamento otimizado e sem consumo desnecessário de recursos.

A nossa política de cache consiste em **guardar os N documentos processados**, assegurando um acesso mais rápido a esses dados em requisições futuras. Esta abordagem permite otimizar o desempenho do sistema, reduzindo a latência e evitando o reprocessamento de informação frequentemente solicitada. **O valor de N é definido pelo *cache_size* quando o servidor é iniciado**, garantindo um equilíbrio entre eficiência e utilização de recursos.

A robustez da solução é visível no **tratamento cuidadoso de erros e na gestão eficiente** de recursos do sistema, como descritores de ficheiros e processos filhos. Este sistema é suficientemente flexível para acomodar diferentes tipos de operações e formatos de resposta, adaptando-se conforme a natureza de cada pedido. A escolha de *pipes*, embora apresente limitações em cenários de extremamente alta carga, oferece uma **solução prática e eficaz** para a maioria dos casos de uso, com a vantagem adicional de ser facilmente depurável e monitorizável através de ferramentas padrão do sistema operativo.

3. Scripts

Os resultados apresentados nos testes de busca paralela revelam informações interessantes sobre o **desempenho das diferentes implementações de paralelismo**. Analisando os dados, observamos que em todos os casos testados, a versão que utiliza 1000 processos demonstrou ser significativamente mais rápida do que a versão com poucos processos. As diferenças de desempenho variam entre **29% e 45%**, com a busca por "Deus" mostrando a maior diferença (42% mais rápido com 1000 processos) e "Israel" a menor (29% mais rápido).

Um padrão claro emerge quando examinamos os tempos de execução: as implementações com 1000 processos **mantêm uma consistência impressionante**, com todos os tempos girando em torno de 0.005 segundos, enquanto as versões com 1 processo apresentam maior variação, oscilando entre 0.0070 e 0.0089 segundos. Esta consistência nos resultados com alta paralelização sugere que o sistema consegue **gerir eficientemente um grande número de processos** para esta tarefa específica.

A drástica redução na diferença de desempenho entre as duas execuções, quando o teste é repetido, pode ser explicada pelo **efeito de *caching* na própria máquina**. Durante a primeira execução, os dados necessários para as buscas são carregados da memória principal para a cache do processador, o que é um processo relativamente lento. Porém, nas execuções subsequentes, estes dados já estão armazenados na cache, que possui tempos de acesso muito menores, reduzindo assim o impacto da paralelização. Isto significa que **o ganho de desempenho proporcionado pelo uso massivo de processos torna-se menos evidente** quando os dados já estão pré-carregados, pois o gargalo de acesso à memória é significativamente atenuado.

Em conclusão, os resultados demonstram que, embora a abordagem com **1000 processos apresente vantagens claras na primeira execução**, esta diferença diminui quando os testes são repetidos devido à otimização proporcionada pelo *caching* da máquina. Isto reforça a importância de considerar não apenas o número de processos, mas também o comportamento do sistema de memória e a natureza repetitiva das operações ao avaliar estratégias de paralelização.

Segue-se a imagem da primeira execução do servidor:

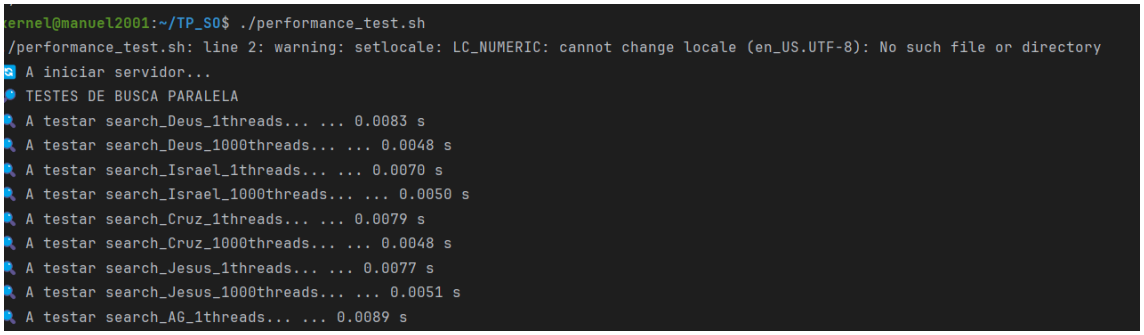

Figura 4: Resultados do *Script*

4. Conclusão

O desenvolvimento deste serviço de indexação e pesquisa de documentos permitiu consolidar conhecimentos teóricos e práticos na área de Sistemas Operativos, especificamente no que diz respeito à **gestão de processos, comunicação entre processos e manipulação de ficheiros** em ambiente Linux. A implementação do servidor e do cliente, com funcionalidades básicas e avançadas, demonstrou a importância de otimizações como **a pesquisa concorrente, a persistência de dados e o *caching*** melhoram a eficiência e a robustez do sistema.

Os testes realizados evidenciaram ganhos significativos de desempenho ao **paralelizar a pesquisa** de documentos, bem como o impacto positivo do ***caching*** na redução do tempo de acesso à meta-informação. A prioridade definida para o **preenchimento da cache** é ocupar, cronologicamente, os documentos que vão sendo guardados nesta. A escolha de políticas adequadas para a cache e a configuração do número de processos concorrentes revelaram-se fatores críticos para o equilíbrio entre desempenho e consumo de recursos.

Em suma, este projeto não apenas cumpriu os objetivos propostos, mas também proporcionou uma oportunidade valiosa para explorar desafios reais no desenvolvimento de uma das **interações mais básicas concebidas por qualquer Sistema Operativo**. As lições aprendidas e as soluções implementadas servirão como base para futuros trabalhos na área, destacando a relevância de uma abordagem sistemática e experimental no desenvolvimento de *software* complexo.