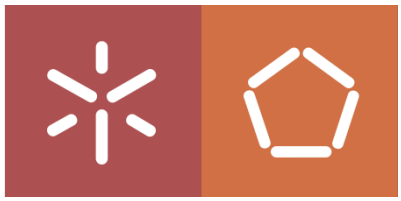
Universidade do Minho

Licenciatura em Engenharia Informática



Sistemas Operativos

2024/2025

Relatório Trabalho Prático

Serviço de Indexação e Pesquisa de Documentos

Grupo 67

a106804, Alice Soares

a106853, Ana Beatriz Freitas

a106877, José Cação

Maio 2025

Índice

1 Introdução

Este relatório aborda o desenvolvimento do trabalho prático da disciplina de Sistemas Operativos, inserido no 2ºano da licenciatura em Engenharia Informática.

O presente trabalho prático teve como objetivo o desenvolvimento de um serviço de indexação e pesquisa de documentos de texto armazenados localmente, recorrendo a uma arquitetura cliente-servidor. Este sistema permite que os utilizadores adicionem, consultem, removam e pesquisem documentos através de um programa cliente, que comunica com um programa servidor usando *pipes* com nome, FIFOs. O servidor é responsável por manter e gerir a meta-informação associada a cada documento bem como por realizar pesquisas no conteúdo dos documentos.

O projeto foi desenvolvido em linguagem C, com uso exclusivo de chamadas ao sistema permitidas no contexto da unidade curricular de Sistemas Operativos. Para facilitar a gestão de estruturas de dados, foi utilizada a biblioteca *GLib*.

Durante o desenvolvimento, foram implementadas diversas funcionalidades, incluindo a persistência da meta-informação, pesquisa concorrente por palavra-chave com controlo de número máximo de processos em paralelo, e uma cache de meta-informação para otimizar o desempenho. Estas funcionalidades foram avaliadas através de testes experimentais, que permitiram aferir o impacto das otimizações propostas.

2 Arquitetura e Estrutura do Sistema

2.1 Arquitetura de Processos

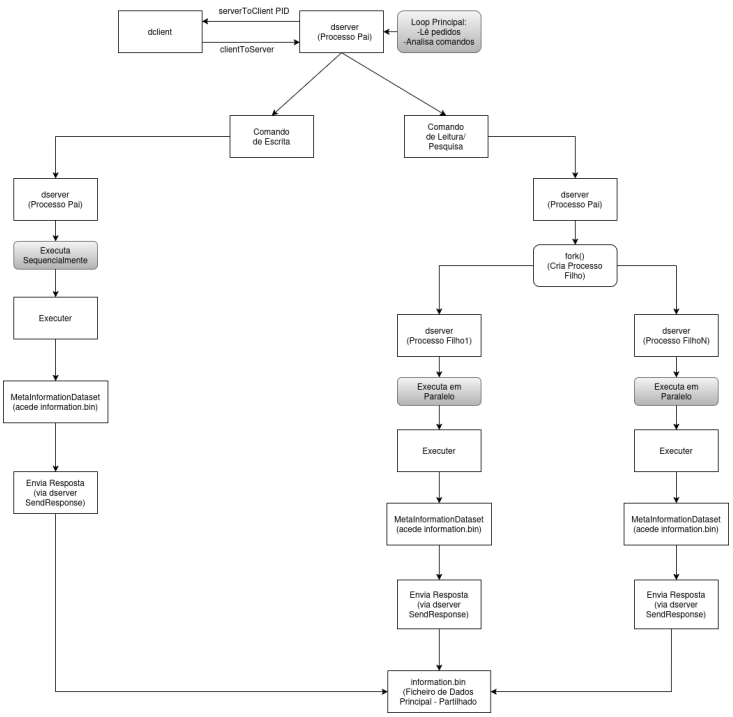


Figura 1: Arquitetura do projeto desenvolvido

2.2 Componentes do Sistema e Estruturas de Dados

O sistema foi implementado de forma modular, com o objetivo de garantir a separação de responsabilidades e facilitar a manutenção do código. Inicialmente, a abordagem adotada focou-se na separação clara entre

a receção de comandos, a representação de dados de entrada e a comunicação entre processos.

A entrada de um pedido no sistema inicia-se no cliente, que invoca o programa passando argumentos pela linha de comandos. Estes argumentos são analisados e convertidos internamente numa estrutura de abstração de comandos. Esta estrutura, definida como *Command*, permite encapsular o tipo de operação pretendida pelo cliente (adicionar, consultar, listar) e os seus argumentos. Esta conversão é essencial para uniformizar o tratamento dos pedidos, independentemente da sua origem ou do conteúdo associado.

Em situações em que o comando efetuado requer metadados documentais, por exemplo, no caso de adição de um novo documento, essa informação é representada através da estrutura *MetaInformation*. Esta estrutura armazena os campos fornecidos pelo utilizador exclusivamente no contexto do pedido em questão. Importa salientar que a estrutura *MetaInformation* não corresponde ao armazenamento persistente no servidor, mas sim à informação transitória associada a um pedido específico.

Para facilitar a comunicação entre cliente e servidor, foi criada a estrutura *Message*. Esta estrutura encapsula tanto o comando, *Command*, como os metadados, *MetaInformation*, bem como o identificador do FIFO do cliente, através do qual será enviada a resposta. Desta forma, garante-se que o servidor pode tratar o pedido de forma completa, dispondo de todos os dados relevantes numa única entidade.

A escolha da utilização de estruturas de tamanho fixo e previsível evita a complexidade da serialização dinâmica e do uso de ponteiros, uma vez que estes não podem ser transmitidos diretamente entre processos. Dado que os *pipes* nomeados funcionam com blocos binários de memória, a utilização de estruturas fixas e previsíveis evita a complexidade associada à serialização dinâmica ou ao uso de ponteiros, que não são transferíveis diretamente entre processos.

2.3 Comunicação Cliente-Servidor

A comunicação entre os processos cliente e servidor foi implementada através de FIFOs, *named pipes*, conforme imposto pelo enunciado do projeto. O sistema segue um modelo cliente-servidor síncrono: o cliente constrói um pedido, envia-o ao servidor e aguarda uma resposta.

Esta comunicação é realizada através de dois canais distintos:

- Um FIFO comum, *fifos/clientToServer*, utilizado por todos os clientes para envio de pedidos ao servidor.
- Um FIFO exclusivo por cliente, criado com base no identificador do processo, PID, utilizado para receber a resposta do servidor.

Esta separação de canais assegura que a resposta enviada pelo servidor seja corretamente direcionada ao cliente que originou o pedido, permitindo também a execução concorrente de múltiplos clientes sem colisão de mensagens.

O canal único de entrada, *fifos/clientToServer*, é criado pelo servidor no arranque e serve apenas para receção de mensagens. Para evitar que a leitura do FIFO termine com EOF quando não existem clientes ativos, o servidor mantém um descritor de escrita aberto em simultâneo com o descritor de leitura, o servidor "*dummy*". Sempre que um cliente escreve no FIFO uma estrutura *Message*, o servidor faz um *bufferedRead*() para obter o pedido completo.

Por outro lado, antes de enviar o pedido, cada cliente cria localmente um FIFO exclusivo, *fifos/serverToClient_<PID>* usando o seu PID como sufixo. Esse FIFO de resposta é enviado no próprio *Message* e permanecerá em escuta pelo cliente após o envio, bloqueado até receber a resposta final.

No lado do servidor, assim que um pedido chega, distinguimos dois casos:

- **Operações de leitura** (-c, -l, -s, -f):

Para evitar o bloqueio do ciclo principal de receção, cada pedido de consulta ou pesquisa é encaminhado para um processo filho criado com *fork*(). O filho executa a lógica correspondente, nomeadamente a consulta de metadados, contagem de ocorrências com `grep -c` ou listagem com `grep -l`, escreve o resultado no FIFO privado do cliente e então termina. Já o processo pai, imediatamente após despachar o filho, recolher eventuais *zombies* e continua a ler novos pedidos. No caso do comando *shutdown* -f, o processo pai marca uma *flag* para terminar o seu próprio *loop* assim que enviar a resposta.

- **Operações de escrita** (-a, -d): Por outro lado, a indexação e remoção envolvem atualização da estrutura global de meta-informação e do ficheiro persistente. Desta forma decidimos tratá-las sequencialmente no próprio processo pai, de modo a simplificar a sincronização de acesso ao ficheiro em disco e à cache em memória. Imediatamente após a execução, o servidor envia a confirmação através do FIFO do cliente.

No final de cada ciclo de leitura, e antes de sair, o servidor garante que todos os processos filhos terminaram e são devidamente "reaped". Deste modo, mantemos o servidor responsivo a novos pedidos, permitimos simultaneidade em operações de leitura, e garantimos que cada resposta chega ao cliente certo.

2.4 Interpretação dos Pedidos recebidos

A execução dos comandos recebidos do cliente é encapsulada no módulo *executer*, o qual abstrai o tratamento das diferentes operações disponíveis sobre os dados indexados. Este módulo expõe uma única função principal, *executer_execute*, responsável por receber um objeto *Command*, interpretá-lo com base na *flag* correspondente, e delegar a sua execução para o *handler* apropriado.

Cada comando é tratado por uma função auxiliar dedicada, responsável por executar a operação sobre a estrutura de dados partilhada *MetaInformationDataset*, e gerar uma resposta textual apropriada. Esta abordagem facilita a manutenção e a extensão do sistema para novos comandos.

Importa salientar que todas estas operações são desenhadas para serem seguras em contexto concorrente. As operações de leitura são invocadas a partir de processos filhos, conforme descrito anteriormente, permitindo múltiplas consultas em simultâneo sem bloqueios. Por outro lado, as operações de escrita são executadas no processo principal, garantindo assim consistência e evitando condições de corrida no acesso ao ficheiro persistente e à cache residente em memória.

3 Funcionalidades e Estratégias de Implementação

O sistema implementado tem como objetivo a gestão de metainformação associada a documentos de texto. Para tal, suporta diversas funcionalidades fundamentais como a adição de novos documentos ao dataset, remoção lógica de documentos existentes, consulta de metainformação por identificador, contagem do número de linhas contendo uma palavra-chave específica, e pesquisa de documentos que contenham uma determinada palavra-chave, tanto em modo sequencial como paralelo.

Adicionalmente, o sistema suporta a recuperação automática do estado anterior após reinício do servidor, através da persistência contínua da metainformação em disco, garantindo continuidade da informação armazenada.

De forma a reduzir o número de acessos ao disco e melhorar o desempenho em operações de leitura recorrente, foi implementada uma cache em memória para armazenar temporariamente até *N* entradas de meta-informação, valor este definido no arranque do servidor.

Durante o desenvolvimento do sistema de gestão de metainformação documental, foram tomadas diversas decisões de implementação com o objetivo de garantir eficiência, robustez, integridade dos

dados e suporte à concorrência. Abaixo detalham-se as principais estratégias adotadas:

3.1 Armazenamento Persistente e Identificação dos Registos

Optou-se por utilizar um ficheiro binário, *information.bin*, para armazenar os registos de metainformação documental. Esta solução permite aceder diretamente a qualquer registo através de operações de *lseek*, possibilitando consultas e remoções eficientes.

O identificador único atribuído a cada documento corresponde à posição (*offset*) no ficheiro onde o respetivo registo é armazenado. Esta abordagem simplifica o mapeamento entre ID e conteúdo e evita a necessidade de estruturas auxiliares para gerar identificadores.

3.2 Gestão de Concorrência com Bloqueios de Ficheiros

Para prevenir *race conditions* e garantir a consistência em ambientes concorrentes, cada acesso ao ficheiro binário é protegido por bloqueios *flocks* de leitura ou escrita, conforme o tipo de operação. Este controlo é especialmente importante para garantir que múltiplos processos ou *threads* não corrompem dados ao acederem simultaneamente ao ficheiro.

3.3 Estratégia de *Tombstones* para Remoção Lógica

Em vez de remover fisicamente os registos do ficheiro, foi adotada a técnica de *tombstones*. Desta forma, quando um registo é removido, é apenas marcado como apagado, preservando a posição e o conteúdo original. Esta decisão evita a reescrição dispendiosa de todo o ficheiro para reorganização e possibilita a reutilização posterior do identificador daquele registo.

Os IDs de registos removidos são armazenados numa *GQueue*, permitindo a sua reutilização eficiente para novos registos, o que evita fragmentação crescente do espaço.

3.4 Reutilização de IDs e Manutenção de Espaço

A utilização da *GQueue* para armazenar posições livres, derivadas de registos marcados como apagados, permite um controlo eficiente de espaço e a reutilização ordenada dos IDs antigos, sem necessidade de compactar o ficheiro binário, mantendo assim a integridade posicional entre ID e *offset*.

3.5 Verificação de Duplicação

Ao adicionar novos documentos, o sistema verifica se o caminho absoluto do ficheiro já se encontra indexado, assegurando que o mesmo documento não é indexado múltiplas vezes. Esta verificação é feita através da leitura sequencial dos registos não apagados. Apesar de ser uma operação linear, a sua execução apenas ocorre em inserções, sendo aceitável num contexto em que a maioria das operações são consultas. No entanto, como melhoria da eficiência, poderiamos manter uma estrutura que contivesse os caminhos para os ficheiros e o respetivo id, evitando percorrer o ficheiro inteiro todas as vezes.

3.6 Pesquisa Sequencial vs Paralela

A pesquisa de documentos que contêm uma palavra-chave foi implementada em duas variantes: uma versão sequencial, que percorre todos os ficheiros um a um utilizando *grep -l*, e uma versão paralela, que recorre a múltiplos processos *fork()* até um número máximo definido por *max_procs*.

Os resultados dos testes e a análise de desempenho destas duas versões podem ser consultados posteriormente.

3.7 Persistência da metainformação em Disco

De forma a garantir a durabilidade dos dados e permitir a recuperação do estado do sistema após reinício, a meta-informação dos documentos é persistida num ficheiro binário, *docs.dat*. Todas as operações de modificação, inserção, remoção lógica, atualização, são imediatamente refletidas neste ficheiro, utilizando operações como *open*, *write*, *lseek*, que permitem manipulação eficiente e direta da estrutura em disco.

No arranque do servidor, a função *read_metadata_file* é responsável por reconstruir o estado interno do sistema, carregando os registos válidos e preparando a fila de identificadores reutilizáveis. Esta abordagem assegura que o sistema mantém a continuidade da informação mesmo após encerramento e reinício.

A terminação controlada do servidor é feita através de um comando especifico (-f) enviado pelo cliente, que aciona a função *stop_server*. Como todas as alterações são persistidas à medida que ocorrem, não é necessário qualquer passo adicional de salvaguarda antes do encerramento.

3.8 Cache de Meta-informação

A estrutura da cache utiliza uma *GQueue* para manter a ordem de inserção dos elementos e uma *GHashTable* para permitir acesso rápido por identificador. Esta combinação permite uma política simples de substituição baseada em *First-In, First-Out*, (FIFO), onde o elemento mais antigo é removido quando a cache atinge a sua capacidade máxima.

No entanto, quando é efetuada uma consulta, ao aceder à cache com sucesso, o identificador correspondente é movido para o final da fila, simulando um comportamento de acesso recente, ainda que sem implementar uma política LRU completa.

Esta abordagem permite reduzir significativamente o custo de acessos repetidos a meta-informação, particularmente em cenários de consulta intensiva.

4 Testes e Resultados

Ao longo da realização deste trabalho prático, fomos utilizando diversos testes, tanto para verificar funcionalidades, abordagens aplicadas e para analisar o desempenho.

Para validar a implementação do sistema e garantir a sua robustez, utilizámos o *framework Bats (Bash Automated Testing System)*. Este é um sistema de testes automatizados para *scripts bash*, que permite escrever testes de forma simples, verificando o comportamento esperado de programas em linha de comandos.

4.1 Scripts de Teste Utilizados

Testes Funcionais

Para validar a implementação das funcionalidades básicas, criámos um conjunto de testes definidos no ficheiro *test_features.bats*. Estes testes cobrem as principais funcionalidades do sistema:

- Indexação da meta informação de um documento
- Consulta da meta informação de um documento

- Remoção de um índice
- Pesquisa num documento de uma certa palavra chave
- Pesquisa dos índices dos documentos com uma certa palavra chave

Também foram usados testes definidos no ficheiro *test_concurrency.bats* para verificar se a paralelização estava corretamente implementada no servidor, e se diferentes pedidos simultâneos de clientes eram atendidos de forma eficiente e sem bloqueios indevidos.

Usamos ainda, para validar a persistência da metainformação guardada, testes que fecharam e voltaram a iniciar o servidor, confirmando que a metainformação tinha sido persistida em disco e recuperada ao reiniciar o programa.

Testes de desempenho

A pesquisa de palavras-chave em documentos representa uma operação intensiva, especialmente quando aplicada a grandes volumes de dados. Para medir o impacto da abordagem de pesquisa concorrente, executámos uma série de testes, disponíveis em *test_parallel_search.bats*, que avaliam o tempo de resposta com diferentes números de processos. Os resultados foram registados em *parallel_search_results.csv*.

Para testar a cache com diferentes tamanhos foram usados os testes disponíveis em *test_cache_performance.bats* e ainda, para testar diferentes políticas de cache, corremos esses testes de novo e comparamos resultados

4.2 Resultados dos testes e Análise de desempenho

Procura Concorrente

Para avaliar de forma precisa o impacto da paralelização na pesquisa de documentos, realizámos um conjunto de testes variando o número de processos usados no servidor para realizar a procura. Para os diferente números de processos, foram guardados tempos de execução (em milissegundos).

	A	B	C	D	E	F	G
1	**mode**	**processes**	**time_ms**				
2	sequential	1	2612		parallel	25	143
3	parallel	1	1631		parallel	26	146
4	parallel	2	726		parallel	27	139
5	parallel	3	495		parallel	28	133
6	parallel	4	403		parallel	29	144
7	parallel	5	325		parallel	30	133
8	parallel	6	287		parallel	31	130
9	parallel	7	257		parallel	32	134
10	parallel	8	248		parallel	33	132
11	parallel	9	211		parallel	34	142
12	parallel	10	205		parallel	35	133
13	parallel	11	183		parallel	36	134
14	parallel	12	1100		parallel	37	139
15	parallel	13	192		parallel	38	136
16	parallel	14	176		parallel	39	130
17	parallel	15	169		parallel	40	136
18	parallel	16	171		parallel	41	138
19	parallel	17	156		parallel	42	131
20	parallel	18	147		parallel	43	135
21	parallel	19	153		parallel	44	144
22	parallel	20	136		parallel	45	157
23	parallel	21	142		parallel	46	144
24	parallel	22	139		parallel	47	125
25	parallel	23	140		parallel	48	127
26	parallel	24	140		parallel	49	139
27					parallel	50	137
28							

Figura 2: Ficheiro *parallel_search_results.csv*

A experiência foi conduzida da seguinte forma:

- O caso *sequencial* representa a execução básica do comando, sem processos, a realizar toda a pesquisa.

- Os restantes testes, *parallel*, avaliam o desempenho com diferentes números de processos, de 1 a 50.

Observa-se uma melhoria acentuada no desempenho ao aumentar o número de processos até cerca de 10 a 16, com tempos de execução a cair de 2612 ms, *sequencial*, para valores próximos dos 170 ms.

No entanto, a partir de 20 a 24 processos, os ganhos começam a estabilizar, sugerindo um ponto de saturação dos recursos disponíveis. Entre 30 a 50 processos, o tempo de execução mantém-se relativamente constante, com ligeiras oscilações, o que indica que adicionar mais processos não resulta em melhorias significativas, podendo até introduzir *overhead* por contenção de recursos e gestão de concorrência.

É particularmente interessante notar um *outlier* no teste com 12 processos, que registou um tempo de 1100 ms, significativamente mais lento do que esperado, possivelmente devido a fatores externos como carga no sistema ou interferência de processos paralelos.

Cache

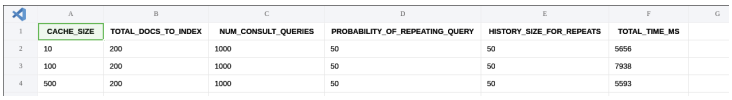
Para a cache fizemos testes com duas abordagens diferentes e diferentes tamanhos. Nomeadamente:

Write-through

Na abordagem *write-through*, todas as operações de indexação são replicadas simultaneamente tanto na memória cache quanto no ficheiro. Esta estratégia garante consistência imediata dos dados, mas tende a gerar maior latência, pois cada escrita implica duas operações.

Cache-aside

A estratégia *cache-aside* (ou *lazy-loading*) opera de maneira diferente: a aplicação acede diretamente o ficheiro quando a informação não está disponível na cache para consulta, e só então armazena a meta-informação em cache para usos futuros.



CACHE_SIZE	TOTAL_DOCS_TO_INDEX	NUM_CONSULT_QUERIES	PROBABILITY_OF_REPEATING_QUERY	HISTORY_SIZE_FOR_REPEATS	TOTAL_TIME_MS
10	200	1000	50	50	5656
100	200	1000	50	50	7938
500	200	1000	50	50	5593

Figura 3: Teste a abordagem write-through

```
cache_size,index_time,consult_time,search_time,remove_time
10,2461,16,1456,16
50,2612,19,1295,15
100,2713,21,1396,16
200,5139,18,1360,21
500,2627,19,1302,18
1000,2747,20,1325,18
2000,3039,23,3681,21
```

Figura 4: Teste a abordagem cache-aside

Durante a execução dos testes práticos, foram observados os seguintes comportamentos:

- A estratégia *cache-aside* apresentou desempenho superior em termos de latência média e tempo de resposta.

- A estratégia *write-through* mostrou-se mais lenta, devido ao custo adicional de escrita simultânea nos dois níveis de armazenamento.

A partir dos testes realizados, conclui-se que a estratégia *cache-aside* é mais adequada para o nosso sistema onde a consistência imediata dos dados não é crítica. Ela proporciona maior flexibilidade e menor sobrecarga de escrita.

5 Conclusões

5.1 Análise Critica e Resultados alcançados

O sistema desenvolvido cumpre os objetivos propostos no enunciado, permitindo a indexação, consulta, listagem, remoção e pesquisa de metainformação documental através de um modelo cliente-servidor concorrente baseado em FIFOs.

A arquitetura escolhida demonstrou ser adequada para o cenário de utilização previsto, proporcionando comunicação fiável entre processos, suporte à concorrência, com separação entre operações de leitura e escrita e gestão eficiente do armazenamento, com acesso direto a registos e controlo explícito do espaço livre.

A implementação foi validada com um conjunto de testes que cobriu cenários normais, extremos e concorrentes. Os resultados demonstraram um tempo de resposta estável nas operações de leitura, mesmo sob carga concorrente, um comportamento consistente em operações de escrita e remoção, sem perdas de integridade e a reutilização eficiente de espaço através da estratégia de *tombstones* e *GQueue*.

5.2 Otimizações e Possíveis melhorias

Embora o sistema satisfaça os requisitos funcionais do enunciado, existem diversas oportunidades de evolução e otimização que podem ser consideradas em versões futuras. Entre elas nomeamos:

- Persistência da estrutura de reutilização de *offsets*: a GQueue usada para guardar posições livres reside apenas em memória. A sua persistência para disco permitiria manter a informação entre execuções do servidor, evitando desperdício de espaço e necessidade de reindexação.
- Compactação periódica do ficheiro binário: apesar do uso de *tombstones* para simplificar a remoção lógica, o ficheiro tende a crescer indefinidamente. Poderia ser implementado um processo periódico de reorganização para eliminar espaços marcados como livres, reduzindo o tamanho total do ficheiro.
- Cache com política de substituição: atualmente, a cache funciona de forma simplificada. A implementação de políticas como *Least Recently Used* ou *Least Frequently Used* poderia melhorar o desempenho em cenários com padrões de acesso repetido.