TRABALHO PRÁTICO

Serviço de Indexação e Pesquisa de Documentos

SISTEMAS OPERATIVOS

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Grupo 1

REALIZADO POR:
Marco Rocha Ferreira, A106857
Marco Rafael Vieira Sèvegrand, A106807
Nuno Henrique Macedo Rebelo, A107372

DATA:
5 de Maio, 2025

ÍNDICE

1. INTRODUÇÃO

Este relatório apresenta uma análise detalhada da implementação do Serviço de Indexação e Busca de Documentos. O sistema foi desenvolvido como parte do trabalho prático da unidade curricular de Sistemas Operativos, seguindo os requisitos especificados no enunciado.

O serviço implementado consiste numa arquitetura cliente-servidor que permite aos utilizadores indexar, consultar e pesquisar documentos de texto. O sistema utiliza mecanismos de comunicação entre processos através de named pipes (FIFOs) e técnicas de gestão de processos para alcançar uma gestão eficiente e confiável dos documentos.

O projeto foi implementado com sucesso, incluindo todas as funcionalidades requeridas:

- Indexação, consulta e remoção de metadados de documentos
- Procura de palavras-chaves em documentos
- Procura concorrente utilizando múltiplos processos
- Persistência dos metadados
- Sistema de cache configurável

Para além disso, foram implementadas otimizações para melhorar o desempenho do sistema, especialmente para operações de procura em grandes coleções de documentos.

2. ARQUITETURA DO SISTEMA

2.1. Visão Geral

O sistema segue uma arquitetura cliente-servidor composta pelos seguintes componentes principais:

- Cliente (`dclient`): Fornece uma interface de linha de comando para os utilizadores interagirem com o servidor. O cliente envia pedidos ao servidor e apresenta os resultados ao utilizador.
- Servidor (`dserver`): Gere a indexação de documentos, armazenamento de metadados e operações de procura. O servidor processa os pedidos dos clientes e pode criar processos filhos para operações de procura.
- Protocolo de Comunicação (`protocol`): Facilita a troca de mensagens entre cliente e servidor usando named pipes (FIFOs). O protocolo define a estrutura das mensagens e os mecanismos de comunicação.
- Sistema de Cache (`cache`): Otimiza o desempenho armazenando metadados de documentos frequentemente acessados na memória, utilizando uma política de substituição LRU (Least Recently Used).
- Gestão de Índice (`index`): Lida com o armazenamento persistente de metadados de documentos em disco, garantindo que a informação sejam preservadas mesmo após o encerramento do servidor.

- Motor de Procura (`search`): Processa procuras de conteúdo em documentos, com suporte para operações concorrentes utilizando múltiplos processos.

A comunicação entre cliente e servidor é realizada através de pipes nomeados, com um pipe de pedidos partilhado por todos os clientes e pipes de resposta individuais para cada cliente. O servidor utiliza o identificador de processo (PID) do cliente para criar um pipe de resposta dedicado.

3. DETALHES DA IMPLEMENTAÇÃO

3.1. Protocol

Objetivo do Módulo

O módulo `protocol` estabelece um protocolo de comunicação interprocessos (IPC) via named pipes (FIFOs), permitindo a troca estruturada de dados entre clientes e servidor. Garante padronização, controle de operações e tratamento assíncrono dos pedidos.

Estrutura e Funcionamento

Este módulo apresenta dois elementos fundamentais que estruturam a comunicação entre processos: a configuração dos pipes, e a definição da estrutura de dados para as mensagens trocadas.

A configuração dos pipes estabelece os canais de comunicação entre clientes e servidor. Existe um pipe principal, denominado `request_pipe`, utilizado para o envio de requisições dos clientes ao servidor. Para as respostas, cada cliente possui um pipe exclusivo, cujo nome é gerado dinamicamente com base no identificador do processo (PID), seguindo o formato `response_pipe_<PID>`. Esta abordagem garante que as respostas do servidor sejam enviadas diretamente ao cliente correto, mesmo quando vários clientes enviam pedidos em simultâneo.

O módulo disponibiliza um conjunto de funções que facilitam a gestão destes pipes:

Criação e remoção de pipes: As funções `create_pipe()` e `delete_pipe()` permitem, respetivamente, criar e eliminar pipes. Estas funções garantem que os canais de comunicação são corretamente inicializados antes do início das trocas de mensagens e removidos quando deixam de ser necessários,.

Abertura e fecho de pipes: Através das funções `open_pipe()` e `close_pipe()`, é possível abrir um pipe para leitura ou escrita, conforme o papel do processo (cliente ou servidor), e fechá-lo de forma segura após a utilização.

A definição da estrutura de dados é feita através da struct Packet, que encapsula toda a informação relevante para cada operação. Cada pacote contém:

- O código da operação (`code`), que identifica o tipo de pedido ou resposta;
- O PID do processo emissor (`src_pid`);
- O identificador único de um documento (`key`);
- O número de linhas obtidas (`lines`), quando aplicável;

- Campos de texto para palavra-chave, título, autores, ano de publicação e caminho do ficheiro (`keyword`, `title`, `authors`, `year`, `path`);
- O número de processos concorrentes nas pesquisas (`n_procs`), quando aplicável.

Para manipular estes pacotes, o componente inclui funções específicas:

A função `create_packet()` permite criar e inicializar uma nova estrutura Packet com todos os campos necessários para a operação pretendida. Por sua vez, a função `delete_packet()` liberta a memória ocupada por um pacote após este já não ser necessário, evitando leaks de memória.

O envio de mensagens é realizado através da função `send_packet()`, que escreve o conteúdo de um pacote num pipe aberto. A receção é feita pela função `receive_packet()`, que lê os dados de um pipe e reconstrói a estrutura Packet correspondente. Ambas as funções garantem que a informação é transmitida de forma estruturada e robusta, com tratamento de erros caso a operação não seja bem sucedida.

3.2. Client

Objetivo do Módulo

O módulo `client` é responsável por servir de interface entre o utilizador e o serviço de indexação e pesquisa de documentos, permitindo a execução de várias operações através de argumentos de linha de comando. Este módulo comunica com o servidor através do protocolo definido em `protocol`, utilizando named pipes para garantir uma comunicação estruturada, segura e eficiente.

Estrutura e Funcionamento

O funcionamento do `client` pode ser descrito pelas seguintes etapas principais:

Validação dos Argumentos: Logo no início da execução, o cliente valida os argumentos recebidos. A função `validate_args()` assegura que cada operação recebe o número correto de argumentos e que, quando necessário, estes são do tipo adequado (por exemplo, números inteiros para identificadores de documentos). Para isso, recorre à função auxiliar `validate_number()`. Caso a validação falhe, o programa apresenta ao utilizador uma mensagem de erro e termina, prevenindo a execução de operações inválidas.

Criação do Pedido: Após validar os argumentos, o cliente constrói um pacote de pedido (Packet) utilizando a função `create_request()`. Este pacote inclui toda a informação relevante para a operação pretendida, como o tipo de pedido (por exemplo, `ADD_DOCUMENT`), o PID do processo cliente, identificadores de documentos, palavra-chave, título, autores, ano, caminho do ficheiro e o número de processos concorrentes quando aplicável.

Por exemplo, ao adicionar um documento, são preenchidos os campos de título, autores, ano e caminho; ao consultar ou eliminar, apenas o identificador do documento é necessário.

Preparação do Pipe de Resposta: Para receber a resposta do servidor, o cliente cria um pipe de resposta exclusivo, cujo nome é baseado no PID do processo (por

exemplo, `response_pipe_12345`). Este pipe garante que a resposta do servidor é entregue apenas ao cliente que fez o pedido.

Envio do Pedido ao Servidor: O cliente abre o pipe global de pedidos (`request_pipe`) para escrita e envia o pacote de pedido através da função `send_packet()`. Após o envio, liberta a memória do pacote e fecha o pipe de pedidos.

Processamento da Resposta: A função `process_response()` é responsável por abrir o pipe de resposta, receber o pacote enviado pelo servidor e apresentar os resultados ao utilizador. O processamento da resposta depende da operação realizada:

Para operações simples como adicionar, consultar, eliminar ou contar linhas, o cliente verifica o código de resposta (`SUCCESS` ou `FAILURE`) e apresenta a mensagem apropriada.

Para a pesquisa de documentos (-s), o cliente pode receber múltiplos pacotes, cada um com um identificador de documento correspondente aos resultados encontrados. Estes identificadores são armazenados numa estrutura dinâmica (GArray) e apresentados em formato de lista.

Limpeza e Encerramento: Por fim, o cliente elimina o pipe de resposta criado, liberta toda a memória utilizada e encerra o programa de forma limpa.

3.3. Server

O módulo `server` é o núcleo do sistema, responsável por receber, processar e responder a todos os pedidos dos clientes, garantindo a gestão dos documentos e a coordenação das operações de pesquisa e manipulação de dados. A comunicação é realizada através de named pipes (FIFOs), utilizando uma estrutura de dados padronizada para garantir a integridade e clareza das mensagens trocadas.

Estrutura e Funcionamento

O servidor segue um ciclo de funcionamento contínuo, composto pelas seguintes etapas principais:

Inicialização: No arranque, o servidor valida os argumentos da linha de comandos, que incluem o caminho para a pasta dos documentos e, opcionalmente, o tamanho da cache. Em seguida, inicializa a cache do sistema e cria o pipe global de pedidos (`REQUEST_PIPE`), que será utilizado para receber todas as requisições dos clientes.

Ciclo Principal: O servidor entra num ciclo onde permanece à escuta de novos pedidos. Cada pedido é recebido como um pacote (`Packet`) através do pipe de pedidos. Para cada pacote recebido, o servidor identifica o tipo de operação solicitado através do campo code e encaminha o pedido para a função de tratamento correspondente.

Processamento dos Pedidos

Adicionar os metadados de documento: O servidor chama a função `AddDocument()` com os dados recebidos. Se a operação for bem-sucedida,

responde ao cliente com o identificador do novo documento; caso contrário, envia uma resposta de erro.

Consultar os metadados de um documento: Utiliza a função `consultDocument()` para obter os metadados do documento solicitado. Se encontrado, devolve os dados ao cliente; caso contrário, indica que não foi possível encontrar o documento.

Eliminar os metadados de um documento: Invoca `deleteDocument()` para remover o documento identificado. O resultado da operação é comunicado ao cliente.

Contar as linhas num documento com uma keyword: Chama `search_keyword_in_file()` para contar quantas linhas do documento contêm a palavra-chave indicada, devolvendo o resultado ao cliente; caso não encontre o documento retorna uma resposta de erro.

Pesquisar os documentos com uma keyword: Dependendo do número de processos concorrentes indicado, utiliza `docs_with_keyword()` ou `docs_with_keyword_concurrent()` para procurar todos os documentos que contêm a palavra-chave. Os identificadores dos documentos encontrados são enviados um a um ao cliente. Se não houver resultados, é enviada uma mensagem de falha.

Desligar o servidor: Ao receber este pedido, o servidor termina o ciclo principal e envia uma resposta de sucesso ao cliente.

Resposta ao Cliente: Para cada pedido, o servidor constrói um pacote de resposta e envia-o através de um pipe exclusivo para o cliente, cujo nome é baseado no PID do processo cliente (`response_pipe_<PID>`). Este mecanismo garante que cada cliente recebe apenas as respostas destinadas ao seu pedido, mesmo em cenários com múltiplos clientes ativos em simultâneo.

Concorrência e Processos Filho: Para operações potencialmente demoradas (contagens e pesquisas), o servidor utiliza a função `fork()` para criar processos filho que executam essas tarefas de forma independente, permitindo que o servidor principal continue a atender outros pedidos sem bloqueios. Após a conclusão da tarefa, o processo filho envia um sinal ao servidor principal para indicar que terminou.

Encerramento e Limpeza: Quando o servidor recebe o pedido de encerramento, termina o ciclo principal, espera pelo término de todos os processos filho, destrói a cache e elimina o pipe de pedidos, garantindo que todos os recursos são libertados corretamente.

3.4. Command

O script Command, tal como o nome sugere, implementa os comandos básicos de gestão de documentos para o sistema de indexação. A função `add_document` recebe os atributos de um documento (título, autores, ano e caminho) e empacota-os numa estrutura que é enviada para a cache. Já a função `consult_document` permite recuperar os dados de um documento guardado, a partir da chave.

Além disso, o script inclui a função `delete_document`, que remove um documento do sistema com base na chave fornecida, e `all_valid_keys`, que lista todos os documentos atualmente indexados.

Todos estes sistemas apoiam-se numa cache que funciona como intermediário entre a memória estática e temporária semelhante ao comportamento observado da RAM para com o disco.

3.5. Index

O Index trata diretamente da persistência dos documentos no ficheiro de índice (`index`). A função `IndexGetKey` calcula qual será a próxima chave disponível, com base no tamanho atual do ficheiro, assumindo que cada entrada ocupa um espaço fixo correspondente ao tamanho da estrutura que contém todos os metadados necessários para `indexPackage`. Ao dividir o tamanho total do ficheiro pelo tamanho de uma entrada, obtém-se o *offset* onde o próximo documento deverá ser guardado.

As funções `IndexAddManager`, `IndexConsultManager` e `IndexDeleteManager` realizam, respetivamente, a escrita, leitura e eliminação lógica de documentos no ficheiro. Todas utilizam a chave como base para calcular o *offset* correto dentro do ficheiro. `IndexAddManager` escreve uma estrutura completa na posição indicada, enquanto `IndexConsultManager` lê os dados de uma posição específica e devolve a estrutura já alocada em memória. Já o `IndexDeleteManager` sobrescreve o espaço de um documento com um pacote "em branco", funcionando como uma remoção lógica. A escolha da eliminação lógica foi pensada com o objetivo de preservação das chaves para todos os documentos indexados.

3.6. Cache

O módulo de gestão da cache tem como objetivo intermediar o acesso aos dados armazenados em disco, minimizando a latência e otimizando a eficiência das operações através da utilização de uma cache em memória. A estrutura baseia-se na utilização de uma `GHashTable` da biblioteca GLib para armazenamento temporário dos elementos, e de um array auxiliar (`OnCache`) que mantém o controlo direto sobre as páginas atualmente em cache e auxilia na implementação da política de *Approximating LRU*.

Cada entrada em cache é representada por uma estrutura `CachePage`, que armazena a chave do elemento, um contador de referências (`ref`) e um indicador de modificação (`dirty`). Sempre que um elemento é consultado ou inserido, este é carregado na cache, respeitando a sua capacidade máxima, capacidade esta que pode ser definida pelo utilizador. Quando esta capacidade é atingida, é então aplicada a política de *Approximating LRU* .

Durante a escrita, se um elemento tiver sido modificado (`dirty = 1`), é garantida a sua persistência no armazenamento permanente através da função `IndexAddManager` referida acima. O módulo também permite operações de leitura (`cacheGet`), escrita (`cacheAdd`), remoção (`cacheDelete`) e destruição da

cache (`cacheDestroy`), sempre assegurando a consistência dos dados entre a memória e o disco.

Importa referir que, apesar de funcional, este módulo possui potenciais pontos de otimização, bem como algumas limitações e problemas detetados durante o desenvolvimento e testes. Estes serão devidamente analisados e discutidos em secções posteriores do relatório.

3.7. Search

A componente das pesquisas consiste, essencialmente, em 3 funções, cada uma implementando um tipo de pesquisa diferente.

Em primeiro lugar, a função `search_keyword_in_file`, que implementa a opção -l, funciona em torno do programa `grep` e tem como parâmetros a palavra a pesquisar (`keyword`), o nome do ficheiro alvo (`file_name`), o caminho da pasta (`folder_path`) e um booleano que indica se deve ser procurada apenas a primeira ocorrência da palavra (`one_ocurrence`). Para realizar a procura, é criado um processo filho, que irá comunicar o resultado com o pai através de um *pipe* anónimo. Após o `fork`, o processo filho fecha o descritor de leitura do *pipe* e redireciona o seu *standard output* para o descritor de escrita do *pipe*, usando a chamada ao sistema `dup2`, para que o pai possa ler o resultado. Por fim, o filho faz `exec` do programa `grep` com diversas *flags*, das quais é importante referir a -c (imprimir o nº de linhas em vez das linhas em si) e a -m (permite ajustar o nº de ocorrências a detetar; útil caso `one_ocurrence` seja verdadeiro). Já do lado do processo pai, ele fecha o descritor de escrita do *pipe* e depois espera pelo filho, usando a chamada `wait`, para recolher o *exit status* deste último e verificar que não ocorreram erros. De seguida, o pai lê do *pipe* anónimo o resultado da procura e, se tudo decorreu sem erros, retorna este resultado; caso contrário, retorna -1.

Em segundo lugar, a função `docs_with_keyword` executa a procura -s de forma sequencial. Para isso, começa por obter da cache um *array* com todas chaves dos documentos que estão indexados pelo servidor no momento. Depois, para cada chave no *array*, interage com a cache para obter a metainformação associada a essa chave, chamando de seguida a função `search_keyword_in_file`, com o `one_ocurrence` a verdadeiro. Em cada iteração destas, se o resultado de chamar esta última função for maior do que 0, acrescenta a um *array* de resultado a chave em questão. O valor de retorno da função é precisamente este *array*.

Por fim, a `docs_with_keyword_concurrent` implementa a procura -s utilizando um dado número de processos. Efetivamente, a função começa por preparar dois *pipes* anónimos (um para o pai escrever todas as chaves (1) e outro para os filhos escreverem os resultados das procuras que efetuarem (2)), lançando depois todos os processos. Cada processo filho vai ler do *pipe* 1 uma chave e fazer algo semelhante ao que é feito na `docs_with_keyword`, i.e., chamar a `search_keyword_in_file` e escrever a chave para o *pipe* 2 se o resultado da procura for maior que 0. No que toca ao processo pai, ele escreve para o *pipe* 1

todas as chaves, depois lê do *pipe* 2 os resultados e, antes de devolver o resultado total, espera por cada filho.

Tal como foi descrito, optamos por implementar a procura concorrente fazendo o pai escrever todas chaves num só *pipe* e os filhos lerem todos deste mesmo *pipe*. Acreditamos que esta solução seja melhor do que fazer uma distribuição estática das chaves pelos filhos (por exemplo, com 3 filhos, atribuir um terço das chaves a cada filho) pois garante uma melhor distribuição do trabalho. Sempre que cada processo termina uma procura, ele vai logo ao *pipe* ler uma nova chave, até estas se esgotarem. Desta forma, não acontecem situações em que um filho fica, por coincidência, com a maioria das procuras "pesadas" e faz os restantes processos esperarem só por ele, mesmo que estes últimos já tenham terminado.

4. DESAFIOS E SOLUÇÕES

4.1. Processos Filho

Durante o desenvolvimento do servidor, a gestão dos processos filhos revelou-se um dos principais desafios, especialmente devido à necessidade de suportar operações concorrentes sem bloquear o processamento de outros pedidos.

Desafio 1: Processos Zombie após Fork

Para permitir que operações potencialmente demoradas, como consultas e pesquisas, não bloqueassem o processamento de novos pedidos, optou-se por criar processos filhos através de `fork()` para tratar estes pedidos de forma independente. No entanto, verificou-se que, ao terminarem, estes processos filhos permaneciam no sistema como processos zombie, uma vez que o processo pai (servidor principal) não fazia wait() imediatamente sobre eles. A acumulação de processos zombie pode degradar o desempenho do sistema e consumir recursos desnecessariamente.

Solução 1

Para resolver este problema, implementamos um mecanismo em que cada processo filho, ao terminar o seu trabalho, atua como um cliente e envia um pedido especial ao servidor (processo pai) a solicitar que este execute o `wait()` correspondente. Desta forma, o processo pai é notificado do término dos filhos e pode libertar corretamente os recursos associados, evitando a acumulação de processos zombie.

Desafio 2: Inconsistência da Cache após Fork

Como a cache de meta-informação foi implementada como uma variável global no servidor, ao fazer `fork()`, cada processo filho ficava com uma cópia independente da cache. Isto significa que qualquer operação (mesmo apenas de leitura) feita pelo filho não se refletia na cache do pai, o que causava inconsistências e desperdício de memória.

Solução 2

Para garantir a consistência e eficácia da cache, decidimos que todas as operações que envolvem a cache, incluindo leituras, fossem feitas pelo processo pai, antes do fork. Assim, o processo pai trata de toda a lógica relacionada com a

cache e apenas depois cria o processo filho para executar o restante trabalho (por exemplo, a pesquisa no conteúdo dos ficheiros).

4.2. Cache

Durante o desenvolvimento do módulo de cache, surgiram diversos desafios relacionados à eficiência do mecanismo de substituição e à performance geral da cache, especialmente em cenários com padrões de acesso aleatórios.

A implementação inicial utilizava uma `GList` da GLib como suporte à política de *Approximating LRU*. Esta abordagem permitia manter um controlo direto sobre a ordem de utilização dos elementos, mas revelou-se pouco eficiente em testes práticos. Em particular, os acessos aleatórios evidenciaram um desempenho pouco satisfatório, com poucos ou nenhuns ganhos face à ausência de cache num index de tamanho médio.

Com o intuito de melhorar a performance, foi realizada uma reformulação da estrutura de controlo, substituindo a `GList` por um array, mantendo a lógica *Approximating LRU*. Esta alteração permitiu reduzir o custo das operações de remoção e atualização das entradas em cache uma vez que os acessos e , resultando numa ligeira melhoria de desempenho, principalmente em cache de tamanho menor. No entanto, os resultados ainda ficaram aquém do esperado.

Como alternativa, foi testada uma política de substituição baseada em eliminação aleatória, onde, sempre que a cache atinge a sua capacidade máxima, é escolhida aleatoriamente uma entrada para remoção. Esta abordagem demonstrou uma performance comparável à versão baseada em LRU com array, sendo consideravelmente mais simples de implementar e manter.

Por fim, identificou-se uma outra possível melhoria que não foi executada devido à sua complexidade e falta de tempo, sendo esta a implementação de um sistema de coleta em bloco, onde nesta técnica, ao invés de se armazenar apenas o bloco diretamente consultado, a cache passa a carregar também blocos adjacentes, antecipando acessos futuros e aumentando a localidade espacial. Esta abordagem ainda se encontra em fase de planeamento e experimentação.

5. TESTES E OTIMIZAÇÃO

5.1. Cache

Foram realizados testes de desempenho da cache com tamanhos de 50 e 5000 entradas, avaliando três tipos de acesso: aleatório, sequencial e repetido, com volumes variando de 30 a 5000 acessos.

Os resultados (Anexo 2) mostraram que caches maiores tiveram tempos médios significativamente menores, especialmente em acessos aleatórios e sequenciais, devido à redução de cache misses. Em acessos repetidos, a cache maior também apresentou excelente desempenho, aproveitando a localidade temporal.

Posto isto, as principais otimizações observadas incluem:

- Redução de cache misses com maior capacidade.

- Aproveitamento da localidade temporal, mantendo dados recentes.
- Exploração da localidade espacial em acessos sequenciais.

5.2. Pesquisa Concorrente

De forma a avaliar o ganho de desempenho ao utilizar vários processos para fazer a pesquisa concorrentemente, foram feitos testes nas seguintes condições:

- *dataset* fornecido pela equipa docente (1645 entradas);
- i7-1355U (10 núcleos, 2 com *Hyper-Threading*) e PC no modo desempenho;
- palavra “marco”, que não ocorre em nenhum ficheiro;
- o servidor executou apenas estes pedidos e um de cada vez, disponibilizando o maior número possível de núcleos para a pesquisa.

Os resultados obtidos (Anexo 1) evidenciam o grande impacto da paralelização da pesquisa, podendo esta chegar a ser cerca de 10 vezes mais rápida do que a versão sequencial. Note-se que os tempos estabilizam a partir dos 12 processos, o que seria de esperar, pois a máquina utilizada possui 12 *threads*, tal como indicado acima. Portanto, ao serem utilizados mais do que 12 processos na pesquisa, estes deixam de ser atribuídos cada um a uma thread, pelo que o tempo deixa de diminuir.

6. CONCLUSÃO

O desenvolvimento do Serviço de Indexação e Pesquisa de Documentos constituiu uma valiosa oportunidade de aprendizagem que nos permitiu consolidar conhecimentos teóricos e práticos da unidade curricular de Sistemas Operativos.

Através da implementação da arquitetura cliente-servidor com named pipes, aprofundamos a nossa compreensão sobre comunicação entre processos, aprendendo a gerir os desafios de sincronização e troca de dados entre componentes independentes do sistema.

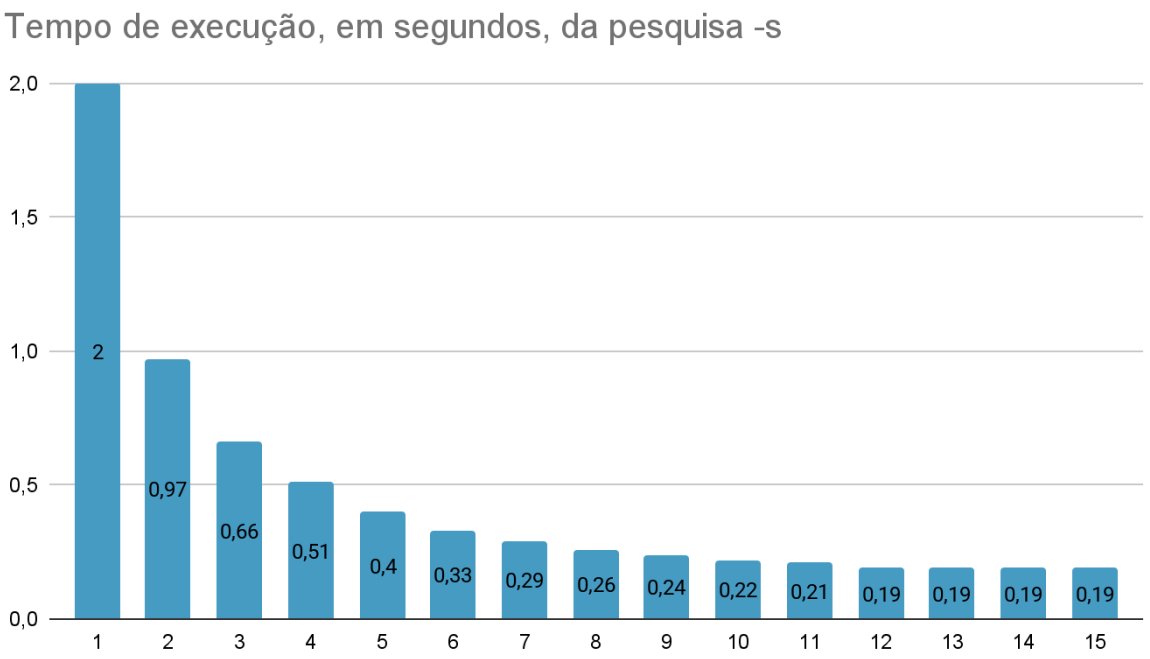
O trabalho com a cache e armazenamento persistente proporcionou-nos conhecimentos importantes sobre gestão de memória e políticas de substituição, obrigando-nos a refletir criticamente sobre as diferentes abordagens e a otimizar a nossa solução inicial.

A implementação da pesquisa concorrente foi particularmente enriquecedora, permitindo-nos aplicar conceitos fundamentais de paralelismo e observar na prática os seus benefícios de desempenho. Aprendemos a importância de considerar as capacidades do hardware subjacente e a distribuir eficientemente as tarefas entre vários processos.

Os desafios encontrados, como a gestão de processos zombie e a manutenção da consistência da cache após fork, proporcionaram-nos valiosas lições sobre a complexidade da programação de sistemas e a importância de uma abordagem metódica na resolução de problemas.

Este projeto reforçou as nossas competências de trabalho em equipa e de gestão de projetos de software complexos, preparando-nos melhor para futuros desafios na área de Engenharia Informática.

ANEXOS



Anexo 1: Tempo de execução da pesquisa -s com diferente número de processos

Relatório de Testes de Cache - 50
Data: 15/05/2025 18:50:59

Acessos Aleatórios

Acessos	Tempo Médio (s)
30	.019
60	.056
100	.078
300	.211
500	.406
1000	1.499
2000	2.415
3000	2.172
4000	3.054
5000	3.878

Acessos Sequenciais

Acessos	Tempo Médio (s)
30	.021
60	.040
100	.075
300	.182
500	.334
1000	.627
2000	1.389
3000	1.968
4000	2.853
5000	3.603

Acessos Repetidos

Acessos	Tempo Médio (s)
30	.018
60	.038
100	.064
300	.197
500	.345
1000	.603
2000	1.242
3000	1.790
4000	2.343
5000	3.263

Relatório de Testes de Cache - 5000
Data: 15/05/2025 18:54:38

Acessos Aleatórios

Acessos	Tempo Médio (s)
30	.017
60	.034
100	.056
300	.165
500	.272
1000	.551
2000	1.160
3000	1.656
4000	2.237
5000	3.121

Acessos Sequenciais

Acessos	Tempo Médio (s)
30	.019
60	.038
100	.060
300	.176
500	.318
1000	.565
2000	1.107
3000	1.670
4000	2.214
5000	2.768

Acessos Repetidos

Acessos	Tempo Médio (s)
30	.018
60	.034
100	.058
300	.166
500	.278
1000	.553
2000	1.111
3000	1.663
4000	2.225
5000	2.768

Anexo 2: Comparação da performance cache tamanho 50 contra tamanho 5000