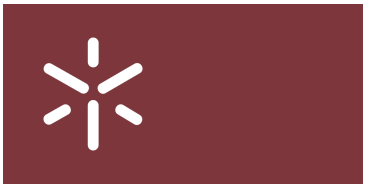


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Operativos

Ano Letivo de 2024/2025

Projeto - Grupo 50

João Teixeira A106836
Simão Mendes A106928
Tiago Brito A106794

Maio, 2025

Índice

Lista de Figuras

Lista de Tabelas

Lista de Anexos

1. Introdução

O presente documento tem como finalidade relatar o trabalho realizado no projeto da unidade curricular de ***Sistemas Operativos* (SO)**, que têm como objetivo a criação de um serviço que permita a indexação e pesquisa sobre documentos de textos guardados localmente num computador, o serviço deverá utilizar dois programas, o programa servidor *(dserver)*, cuja função é a registração da meta-informação sobre cada documento e permitindo a execução de um conjunto de interrogações sobre a meta-informação e ao conteúdo dos documentos e o programa *(dclient)* que servirá como a forma de os clientes interagirem com o serviço.
Entre as funcionalidades posteriormente faladas, salientam-se a utilização de pesquisa concorrente para a pesquisa global de uma palavra chave permitindo assim melhorar a performance na execução dessa ***query*** e a utilização do algoritmo ***Clock*** para a manipulação das entradas mantidas na cache.

2. Arquitetura do Sistema

2.1. Diagrama de Módulos

O diagrama abaixo apresenta, de forma simplificada, os principais módulos funcionais do programa. Este diagrama ilustra as interações entre os módulos e descreve o funcionamento do serviço desde o pedido efetuado pelo cliente até a resposta dada pelo servidor.

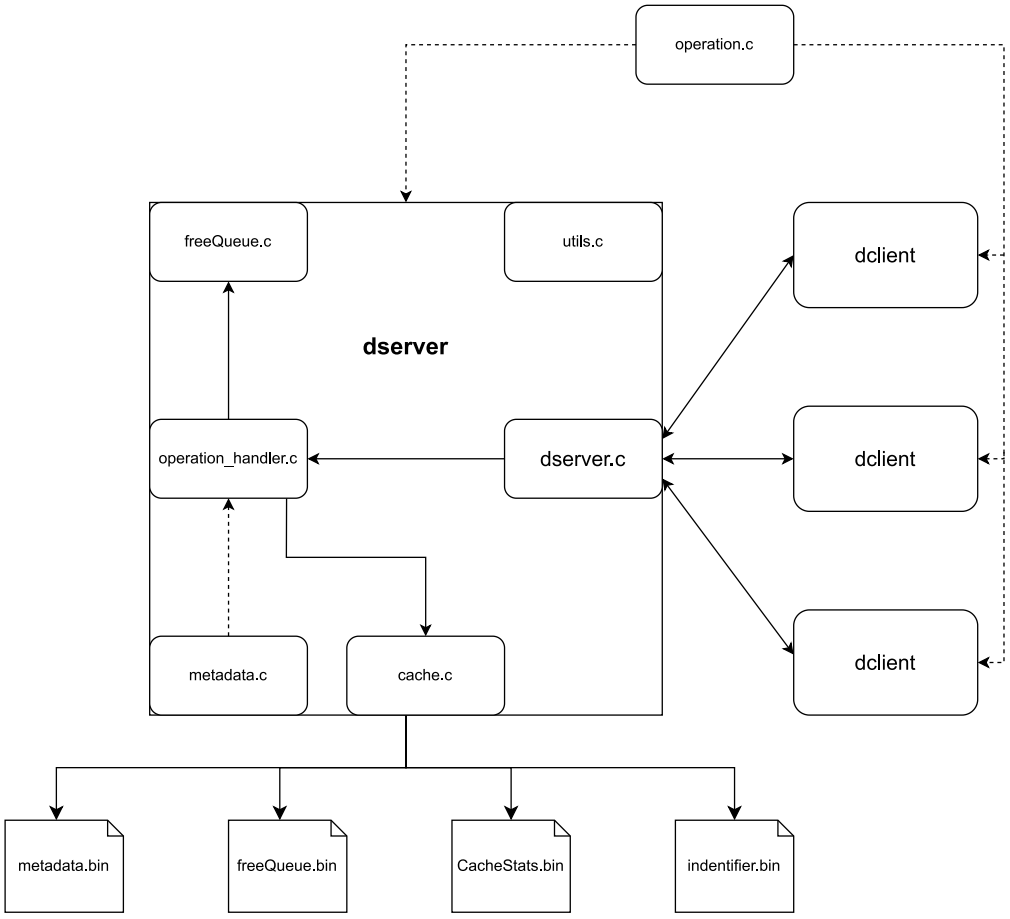


Figura 1: Arquitetura do Sistema.

2.2. Comunicação Cliente-Servidor

Para a comunicação entre cliente e servidor, optou-se pela utilização de *named pipes* (FIFOs), uma vez que estes fornecem um mecanismo simples, eficiente e adequado para comunicação entre processos em sistemas UNIX, especialmente quando os processos não têm relação hierárquica (como pai-filho), como é o caso de um sistema cliente-servidor.

A escolha dos pipes nomeados permitiu uma implementação assíncrona e modular, facilitando o desenvolvimento independente do cliente e do servidor. Além disso, são facilmente criados e geridos no sistema de ficheiros, o que simplifica o *debug* e análise do comportamento do sistema.

Estrutura da Comunicação:

- **Pipe principal de pedidos:** Foi criado um FIFO nomeado com caminho constante `tmp/serverChannel`, usado exclusivamente para enviar pedidos dos clientes para o servidor. Este pipe é aberto para leitura e escrita no lado do servidor (modo não bloqueante) e para escrita no lado dos clientes. A decisão de manter um único canal de entrada para o servidor facilita a monitorização e centralização dos pedidos.
- **Pipes individuais de resposta:** Para responder a cada cliente de forma dedicada, o servidor cria um FIFO específico com base na macro `CLIENT_BASE_PATH` que corresponde a `tmp/clientChannel` e concatena o PID do cliente ao nome, garantindo um canal exclusivo por cliente (`tmp/clientChannel<PID>`). Desta forma, cada cliente pode ler a sua resposta sem interferência de outros clientes. O servidor escreve nesse pipe e o cliente lê.

Vantagens da abordagem

- **Isolamento entre clientes:** cada resposta é enviada num canal exclusivo, evitando concorrência na leitura.
- **Escalabilidade controlada:** o servidor pode gerir múltiplos clientes simultaneamente, mantendo o pipe de pedidos como ponto único de entrada e utilizando pipes individuais para saída.

3. Funcionalidades Implementadas

Nesta secção será apresentado as interações possíveis entre o cliente e o servidor, dando uma breve introdução às funções implementadas, mostrando a abordagem usada para a resolução das interrogações, para além de mostrar exemplos do seu funcionamento. Para a criação do pedido do cliente, foi criada uma estrutura denominada ***operation***, que armazena as informações fornecidas pelo cliente, juntamente com o número de caracteres ocupados pela linha, a ***flag*** associada à operação e o ***pid*** do processo do cliente. Para a resposta dada pelo servidor foi utilizada a mesma estrutura que o cliente enviou ao servidor, onde será modificada, terá guardada a resposta ao cliente, o número de caracteres que a mesma ocupa e a ***flag*** (r), não se alterando o ***pid*** que está guardado na operação enviada ao cliente.

3.1. Indexação (-a)

A funcionalidade de indexação foi implementada com o objetivo de permitir que o cliente envie informações ao servidor sobre determinados documentos, possibilitando que esses dados sejam utilizados em outras funcionalidades oferecidas pelo serviço.
Para a execução desta funcionalidade o utilizador deverá enviar o titulo, os autores, o ano e o nome do documento para o programa ***dclient*** com a utilização da flag -a, o servidor irá depois adicionar a um

ficheiro a informação sobre esse documento garantido que a informação seja mantida mesmo depois de o server ser desligado.
A metainformação é inserida na primeira posição livre da fila de espaços disponíveis (*freeQueue*), ou, caso esta esteja vazia, no final do ficheiro via *append*, tornando esta operação eficiente (complexidade $O(1)$).

```shell
./bin/dclient -a "title" "authors" "year" "path"
```

3.2. Consulta (-c)

A funcionalidade de consulta foi implementada com o objetivo de poder ser consultada a meta-informação de um determinado documento que tenha sido previamente anexado.
Para a sua execução, o utilizador deverá fornecer o identificador único da meta-informação desejada. A operação recorre a uma *cache* para otimizar o tempo de acesso, especialmente em consultas repetidas.

```shell
./bin/dclient -c "key"
```

3.3. Remoção (-d)

A funcionalidade de remoção foi implementada com o objetivo de poder permitir aos utilizadores remover a meta-informação de um determinado documento. No entanto, a remoção não implica a eliminação física do registo. A metainformação é apenas marcada como inválida, ficando disponível para reutilização posterior. Para mais detalhes, consultar a Seção 4.5.
Para a execução desta funcionalidade o utilizador deverá enviar o identificador único da meta-informação que se pretenda remover.

```shell
./bin/dclient -d "key"
```

3.4. Contagem de Linhas (-l)

A funcionalidade de contagem de linhas permite ao utilizador pesquisar em um documento o número de linhas que possuem uma determinada palavra. O sistema consulta inicialmente a *cache* para obter rapidamente a localização do ficheiro, otimizando o desempenho da operação.
Para a execução desta funcionalidade, o utilizador deverá enviar o identificador único da meta-informação do ficheiro onde pretende realizar a contagem, bem como a palavra que será usada como critério para a contagem de linhas.

```shell
./bin/dclient -l "key" "keyword"
```

3.5. Pesquisa Global (-s)

A funcionalidade de pesquisa global permite ao utilizador pesquisar por todos os documentos anexados por o servidor procurando por aqueles que possuem uma determinada palavra-chave.
A pesquisa é paralelizada com a criação de múltiplos processos. São lançados processos indefinidamente até serem analisados todos os metadados, mas o sistema apenas mantém ativos *nr_processes* em simultâneo, como definido pelo utilizador, permitindo controlar o uso de recursos.
Para a execução desta funcionalidade, o utilizador deverá enviar a palavra-chave e o número de processos que gostaria que fossem criados.

```shell
./bin/dclient -s "keyword" "nr_processes"
```

3.6. Parar o Servidor (-f)

Permite encerrar o servidor de forma segura. Para terminar a execução do servidor, um utilizador deverá enviar um pedido ao servidor com a *flag -f*. Ao receber este pedido, o servidor fecha todas as ligações, guarda os dados necessários em disco e liberta os recursos utilizados, garantindo a integridade dos dados e do sistema no encerramento.

```shell
./bin/dclient -f
```

4. Otimizações Realizadas

Nesta secção, descrevem-se as otimizações implementadas no serviço de indexação e pesquisa de documentos. O objetivo destas otimizações é melhorar a **eficiência**, **escalabilidade** e **robustez** do sistema, garantindo tempos de resposta **mais rápidos** e uma **gestão mais eficaz** dos recursos, mesmo com um número elevado de documentos e operações concorrentes. Para tal, foram introduzidas várias funcionalidades adicionais ao servidor, complementando a implementação base.

As otimizações incluem a possibilidade de realizar pesquisas de forma concorrente, a persistência da meta-informação dos documentos, a utilização de uma cache configurável para acelerar o acesso a dados frequentemente utilizados, e a gestão de memória e processos através de mecanismos adicionais, nomeadamente uma *free queue* e um *garbage collector*. Cada uma destas otimizações será detalhada nas secções seguintes.

4.1. Pesquisa Concorrente

Criação de processos filhos

A pesquisa por palavra-chave nos documentos é realizada de forma concorrente através da criação de processos filhos com a chamada ao sistema *fork()*. Quando o servidor recebe um pedido de pesquisa, ele lança imediatamente um processo filho responsável por gerir toda a operação de pesquisa, libertando-se de seguida para continuar a tratar outros pedidos. O processo filho encarrega-se, por sua vez, de criar subprocessos responsáveis por executar o comando *grep* para verificar a existência da palavra num ficheiro específico.

Esta estratégia evita que o servidor principal fique bloqueado devido a operações de pesquisa mais demoradas, garantindo que permanece sempre disponível para outros clientes. Cada pesquisa é independente, e os resultados são recolhidos pelo processo filho através de pipes anónimos.

Execução em paralelo controlada

O processo filho responsável pela pesquisa implementa um controlo do número máximo de processos subprocessos ativos. Antes de criar um novo subprocesso, é verificado se o número atual de processos em execução já atingiu o limite definido (*nr_processes*), valor este passado como argumento pelo cliente. Se o limite tiver sido atingido, o processo aguarda pela conclusão de alguns subprocessos antes de continuar a lançar novos. Este mecanismo assegura uma execução paralela eficiente, evitando a sobrecarga do sistema e garantindo o equilíbrio entre desempenho e estabilidade.

Comando de execução:

```shell
./bin/dclient -s "keyword" "nr_processes"
```

4.2. Persistência

Estrutura e armazenamento dos dados em disco

A meta-informação dos documentos é armazenada em ficheiros binários: `metadata.bin` para as informações de cada documento, `freeQueue.bin` para a fila de posições livres e `identifier.bin` para o próximo identificador livre a ser usado no sistema. Cada vez que um documento é indexado ou removido, o sistema atualiza os ficheiros em memória de forma eficiente, refletindo o estado atual dos dados.

Leitura dos dados no arranque e finalização

Quando o servidor é iniciado, ele carrega os ficheiros `metadata.bin`, `freeQueue.bin` e `identifier.bin`, restaurando o estado interno do sistema e permitindo que o servidor retome à sua operação imediatamente. Ao ser encerrado (por meio do comando -f), o servidor guarda todas as novas modificações aos ficheiros.

Explicação do mecanismo de persistência

A persistência dos dados é fundamental para garantir que as informações não se percam após reinicializações ou falhas do servidor. O uso de ficheiros binários facilita uma leitura e escrita rápidas e eficientes, minimizando o impacto no desempenho do sistema.

4.3. Caching

Cache em memória

Para melhorar o desempenho, o servidor utiliza uma cache em memória que armazena os metadados mais acedidos. O tamanho da cache é definido pelo utilizador no momento da abertura do servidor e permanece fixo durante toda a execução. A cache é preenchida a partir do ficheiro `metadata.bin` no arranque. As operações de busca utilizam a cache, acelerando o acesso aos metadados e reduzindo leituras diretas do disco.

Política de substituição

Quando a cache atinge a capacidade máxima, aplica-se o ***Clock Algorithm***, que aproxima o comportamento do *Least Recently Used* (LRU). Cada entrada possui um bit de "uso recente", que é verificado ciclicamente. Se o *bit* for 0, a entrada é removida; se for 1, o *bit* é reconfigurado para 0 e o ponteiro avança para a próxima entrada. O processo continua até que uma percentagem do tamanho da cache seja removida, garantindo que as entradas menos usadas sejam eliminadas. Após a remoção, a cache é repopulada a partir do ficheiro `metadata.bin`, reiniciando a leitura se necessário. A imagem ilustrativa do funcionamento do *Clock Algorithm* pode ser encontrada em Anexo 1.

Fluxo de Execução

O ciclo de funcionamento da cache segue as seguintes etapas:

- **Inicialização:** No arranque do servidor, a cache é preenchida com as primeiras entradas lidas do ficheiro `metadata.bin` até atingir a capacidade definida.
- **Busca:** As consultas são feitas diretamente à cache. Se o metadado estiver presente, é marcado como recentemente usado. Se não for encontrado, prossegue-se para a substituição.

- **Substituição:** Se o metadado não for encontrado, aplica-se a política *Clock*, analisando ciclicamente as entradas da cache e removendo as menos recentemente usadas para libertar espaço.
- **Repopulação:** A cache é reabastecida com novos metadados a partir do ficheiro `metadata.bin`, continuando a partir da última posição lida.

O processo de substituição e repopulação é repetido até todas as entradas terem sido analisadas uma vez. Se o metadado não existir no ficheiro, o servidor retorna `NULL`. A imagem ilustrativa do fluxo de execução da cache pode ser consultada em Anexo 2.

4.4. Free Queue

Objetivo

A *freeQueue* serve para otimizar a remoção de metadados no ficheiro `metadata.bin`. Quando um metadado é removido, ele é marcado como inválido e a sua posição é armazenada na *freeQueue*. Isto evita a necessidade de mover todas as entradas subsequentes, o que seria um processo custoso de consolidação, onde o sistema teria de copiar todas as entradas seguintes ao metadado removido para a sua posição, a fim de preencher o espaço livre. Em vez disso, a posição do metadado removido é reaproveitada quando necessário, reduzindo a fragmentação e a sobrecarga de I/O. A *freeQueue* é persistente, guardando o seu estado no ficheiro `bin/freeQueue.bin`, e funciona em conjunto com o *garbage collector* para reorganizar e compactar o ficheiro sempre que necessário.

Funcionamento

Ao iniciar o servidor, a *freeQueue* é carregada do ficheiro `bin/freeQueue.bin` se este existir. Caso contrário, é criada uma nova fila vazia. Quando um metadado é removido, a sua posição no ficheiro é marcada como inválida e inserida no final da fila. Quando o servidor precisa de espaço para novos metadados, tenta primeiro reutilizar uma posição livre removendo a posição do início da fila. Se a fila estiver vazia, o sistema escreve os dados diretamente no final do ficheiro. A *freeQueue* é salva no fim da execução do servidor em disco para garantir persistência, e pode ser limpa sempre que necessário. Este processo minimiza a consolidação do ficheiro e trabalha em conjunto com o *garbage collector* para otimizar a reutilização do espaço. Todo este processo pode ser encontrado em Anexo 3.

Notas Complementares

- A *freeQueue* guarda apenas a posição no ficheiro, sob a forma de índice, onde está a entrada de metadado inválida, não o conteúdo do próprio metadado.
- A *freeQueue* é usada pelos processos de inserção e eliminação de metadados, bem como pelo *garbage collector*.

4.5. Garbage Collector

Objetivo

O *garbage collector* limpa metadados inválidos no ficheiro `metadata.bin`, removendo entradas desnecessárias e reescrevendo o ficheiro com apenas dados válidos, contribuindo para a eficiência do sistema.

Funcionamento

O *garbage collector* é executado após a eliminação de um número definido de metadados ou sempre que o servidor é iniciado. O processo ocorre da seguinte forma:

- **Limpeza da *freeQueue*:** A *freeQueue* é limpa para garantir que as posições livres estão atualizadas.
- **Leitura do Ficheiro:** O ficheiro de metadados é lido em blocos, cujo tamanho é determinado por um valor fixo.

- **Filtragem de Entradas Válidas:** As entradas válidas dentro de cada bloco são copiadas para um ficheiro temporário.
- **Limpeza do Ficheiro Original e Escrita do Temporário:** O ficheiro original é limpo, e os dados do ficheiro temporário são escritos de volta no ficheiro original em blocos.
- Limpeza: O ficheiro temporário é eliminado e os ficheiros são fechados.

Este processo encontra-se representado em Anexo 4.

Resultado

O processo de *garbage collection* elimina os metadados inválidos, garantindo que apenas dados válidos permaneçam no ficheiro de metadados. Além disso permite que a *freeQueue* não cresça descontroladamente. Tudo isto melhora a eficiência do sistema, reduz a fragmentação e otimiza o espaço e coesão do armazenamento.

5. Avaliação Experimental

5.1. Metodologia

A avaliação experimental teve como objetivo analisar o desempenho do sistema em dois eixos principais: **pesquisa paralelizada** (global), **escalabilidade com múltiplos processos concorrentes** e **impacto da cache** no tempo de resposta. Para garantir reprodutibilidade, os testes foram realizados num ambiente controlado com as seguintes configurações:

- ***Hardware:***
  - **Processador:** Intel Core i7-1255U, 10 *cores* (2 de desempenho + 8 de eficiência), 12 *threads*.
  - **Memória RAM:** 16 GB DDR4 3200 MT/s.
  - **Armazenamento:** SSD NVMe, 1500 MB/s leitura.
- ***Software:***
  - **Sistema Operativo:** ManjaroLinux 25.0.1.
  - **Versão do Compilador:** GCC 14.2.1.

Teste 1: Pesquisa Paralelizada Global

- **Objetivo:** Medir o ganho de desempenho ao paralelizar a pesquisa de uma *keyword* nos documentos indexados.
- **Parâmetros:**
  - ***Keyword:*** "romeo".
  - **Número de processos:** 1 ou 2.
- **Procedimento:**
  - **Execução do script `./paralelism.sh` contendo os seguintes comandos:**

```bash
time ./bin/dclient -s "romeo" 1   # 1 processo
time ./bin/dclient -s "romeo" 2   # 2 processos
```

Obs: Realizou-se a repetição do teste 5 vezes para garantir consistência dos resultados.

Teste 2: Escalabilidade com Processos Concorrentes

- **Objetivo:** Avaliar o impacto do aumento de processos no tempo de resposta para operações concorrentes.
- **Parâmetros:**
  - **Número de repetições da operação concorrente a realizar:** 10 (Pesquisa global da *string* "abcdefghijklmnopqrstuvwxyz").
  - **Número de processos:** 16 (utiliza 1 processo inicialmente e este número vai sendo incrementado de 1 em 1 até 16).
- **Procedimento:**
  - **Execução do script:**

```bash
./processTest.sh 10 16  #10 repetições por configuração, até 16 processos
```

Durante a execução, obtém-se o tempo médio após a execução de 10 operações para cada configuração.

Pré-condição para a execução dos Testes 1 e 2:

Para os dois testes realizados acima, realizaram-se os seguintes passos:

- **Indexação de 16.450 documentos via:**

```bash
./addGdatasetMetadata.sh dataset/Gcatalog.tsv
```

- **Reinicialização do servidor entre testes** para evitar interferência de cache residual, bem como **limpeza dos ficheiros binários de *metadata* residuais**.

Teste 3: Impacto da Cache e Comparação com Operação Direta

- **Objetivo:** Comparar o tempo de resposta entre duas configurações:
  - **Cache ativa** (função *searchCache* original, usando *Clock Algorithm*).
  - **Cache desativada** (função *searchCache decoy* que ignora a cache).
  - Além disso, medir a taxa de acertos (***hit rate***) da cache em cenários realistas.
- **Configurações do Teste:**
  - **Versões do Programa:**
    - **Configuração A:** Programa com *searchCache* funcional (cache ativa).
    - **Configuração B:** Programa com função *decoy* (ignora cache).
  - **Parâmetros Fixos:**
    - **Número de operações:** 100 consultas aleatórias (*flag* `-l`).
    - **IDs aleatórios:** Valores entre 1 e 50000 (distribuição uniforme).
  - **Parâmetros Variados:**
    - **Tamanho da cache** (só relevante na Configuração A): De 1000 a 51000 entradas (incrementos de 5000).
- **Procedimento:**
  - **Pré-Indexação Automática:** O *script* indexa 49350 documentos (30 execuções de `./addGdatasetMetadata.sh dataset/Gcatalog.tsv`) para garantir dados suficientes.

- **Execução para Cada Configuração:**

```bash
# Configuração A (Cache Ativa)
./cacheTest.sh 100  # Versão do programa com a função normal de procura
```

```bash
# Configuração B (Cache Desativada)
./cacheTest.sh 100  # Versão do programa modificada com função decoy ativa
```

- **Fluxo Detalhado (Configuração A):**
  - Para cada **tamanho de cache (i):**
    - **Inicia o servidor com tamanho i:**

      ```bash
      ./bin/dserver dataset/ $i
      ```

    - **Executa 100 consultas aleatórias:**

      ```bash
      ./bin/dclient -l $random_number "romeo"  # random_number ∈ [1, 50000]
      ```

  - **Métricas Coletadas (no ficheiro cache.txt e no terminal):**
    - **Tempo médio por operação.**
    - ***Hit rate.***
    - **Encerra o servidor.**
- **Fluxo para Configuração B:**
  - Mesmo procedimento, mas usando a **função *decoy*** (sem cache).

Resumo Metodológico

Os Testes 1 e 2 focaram-se na **concorrência:** o primeiro comparou **ganhos de paralelização** (1 vs. 2 processos), enquanto o segundo analisou **escalabilidade incremental** (1 a 16 processos). O Teste 3 isolou o **impacto da cache**, contrastando uma versão otimizada (*Clock Algorithm*) com uma função *decoy* que acessa os dados diretamente. Esta abordagem permitiu quantificar **ganhos de desempenho, taxas de acertos** (*hit rate*) e *overhead* de I/O, validando empiricamente as otimizações descritas no sistema. A metodologia adotada assegurou dados robustos para análise crítica, conforme detalhado na secção seguinte.

5.2. Resultados

Teste 1: Pesquisa Paralelizada Global *(Para mais detalhes, consultar Tabela 1, Tabela 2 e Anexo 5)*
- **1 processo:** Tempo médio de 27.9 segundos (variação: ±0.5s entre execuções).
- **2 processos:** Tempo médio de 14.8 segundos (redução de 53%).
- **Consistência:** Resultados replicados em 10 execuções (desvio padrão < 0.3s).

Teste 2: Escalabilidade com Processos Concorrentes *(Consultar Tabela 3 e Anexo 6 para detalhes)*
- **Melhor desempenho:** 3.42 segundos com 16 processos (redução de ± 85% vs. 1 processo - 23.46s).
- **Ponto de saturação:** Ganhos marginais < 5% além de 10 processos (ex: 3.57s com 10 vs. 3.42s com 16).

Teste 3: Impacto da Cache *(Para mais detalhes, consultar Tabela 4, Anexo 7, Tabela 5 e Anexo 8)*
- **Cache ativa (36000 entradas - melhor tempo registado):**
  - **Tempo médio:** 0.00608 segundos para realizar 100 operações de teste.
  - ***Hit rate:*** 100% (máximo observado).

- **Cache desativada:**
  - Tempo médio de 0.01231 segundos (2.02× mais lento).

(Resultados completos para todos os testes acima podem ser observados em Tabela 1, Tabela 2, Anexo 5, Tabela 3, Anexo 6, Tabela 4, Anexo 7, Tabela 5 e Anexo 8).

5.3. Análise

Teste 1: Paralelização

A redução de $\sim 53\%$ no tempo com 2 processos confirma a eficiência da estratégia *fork-exec* para operações de I/O. O não atingimento de um valor superior a esse deve-se ao *overhead* de criação de processos e sincronização (ver Anexo 5).

Teste 2: Escalabilidade

A escalabilidade quase linear até 4 processos sugere bom aproveitamento dos cores físicos. A saturação além disso reflete limitações de *hyper-threading* e contenção de recursos (CPU/RAM). A curva de desempenho (Anexo 6) ilustra claramente o *diminishing returns*.

Teste 3: Cache

O *hit rate* de 81% com cache grande (antes de chegar aos 100%) valida o *Clock Algorithm* como política eficiente. A diferença de 0.00622s entre cache ativa/desativada comprova que a cache reduz drasticamente (cerca de metade) o I/O em operações repetidas (gráficos em Anexo 7 e Anexo 8).

6. Conclusão

O projeto desenvolvido no âmbito da unidade curricular de Sistemas Operativos implementou um serviço **cliente-servidor** para indexação e pesquisa de documentos, integrando funcionalidades como **indexação (-a)**, **consulta (-c)**, **remoção lógica (-d)**, **pesquisa global paralelizada (-s)** e **gestão segura de recursos (-f)**. As otimizações — como **cache** com política *Clock Algorithm*, *free queue* e *garbage collector* — garantiram **eficiência**, reduzindo I/O e **melhorando tempos de resposta.**

A avaliação experimental evidenciou ganhos significativos: **53% de redução** no tempo de pesquisa com 2 processos, ***hit rate* de 81%** (antes de atingir os 100%) na cache (0.00608s vs. 0.01231s sem cache) e escalabilidade linear até 4 processos. O sistema mostrou-se robusto, equilibrando concorrência, persistência e gestão de recursos, validando as escolhas de design em ambientes de alta demanda.

7. Bibliografia

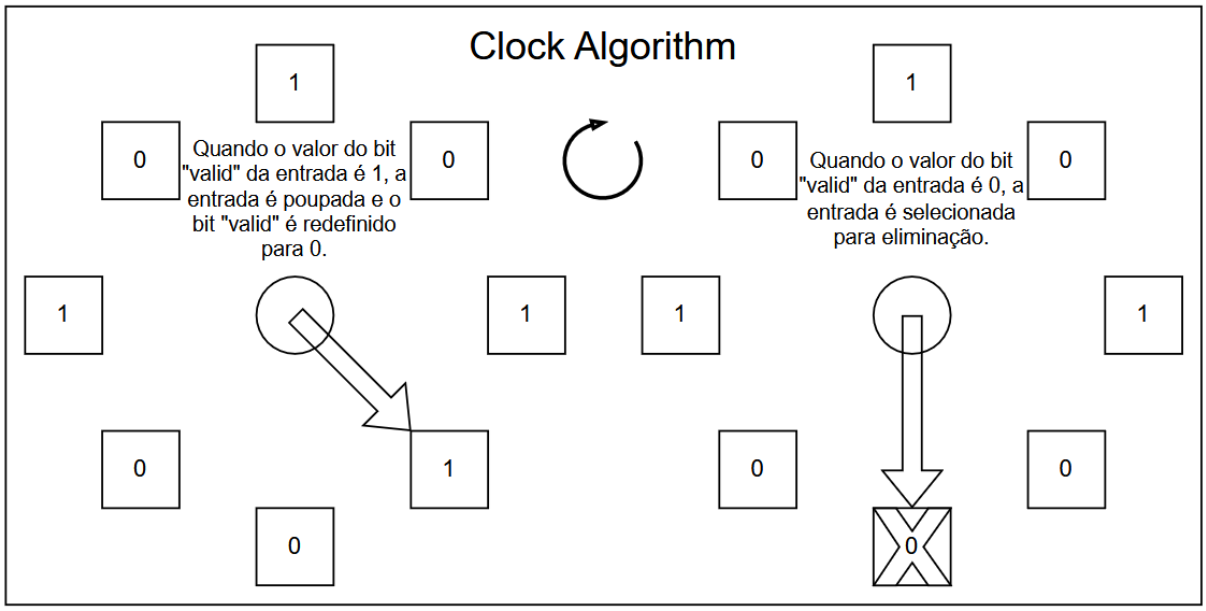
[1] J. T. M. Paulo, «Slides da Unidade Curricular de Sistemas Operativos».

[2] Comunidade Linux, «Man Pages - Documentação de Comandos e Chamadas ao Sistema». [Online]. Disponível em: https://man7.org/linux/man-pages/

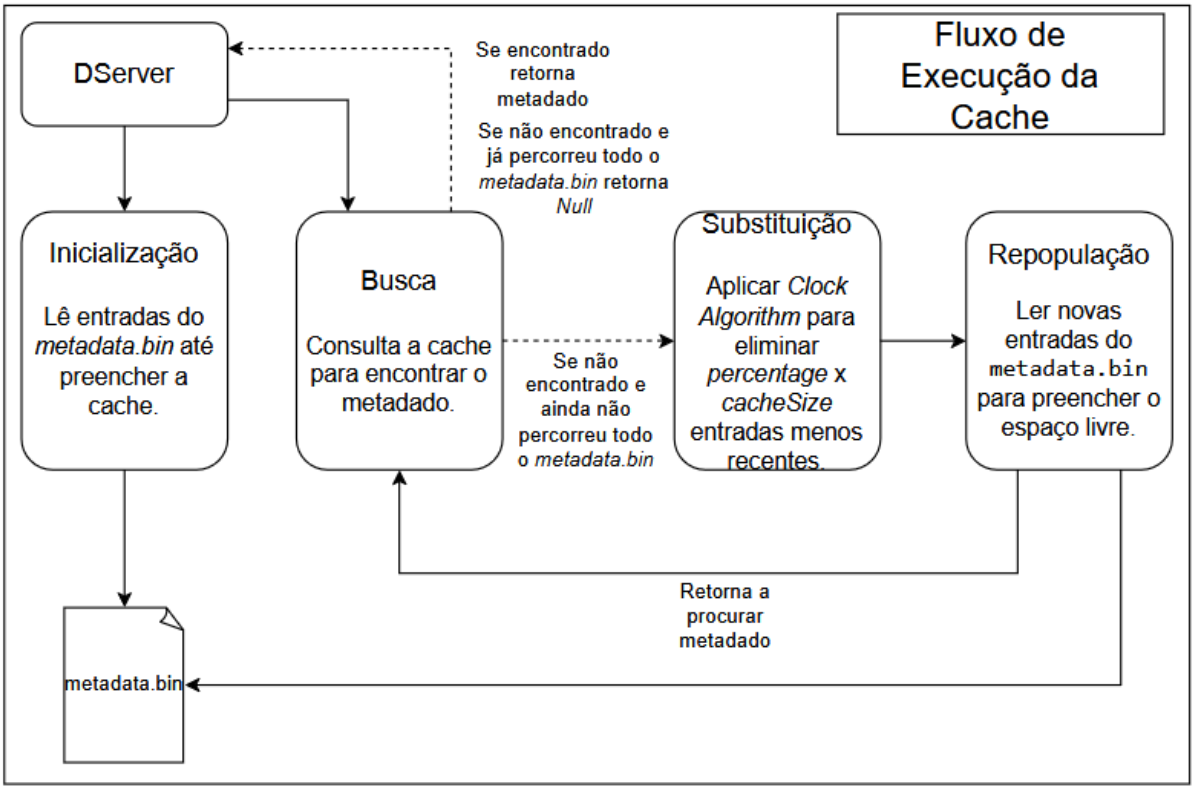
[3] A. C. Arpaci-Dusseau Remzi H. & Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci--Dusseau Books.

[4] The GNOME Project, «GLib: Low-level Core Library». [Online]. Disponível em: https://developer.gnome.org/glib/stable/

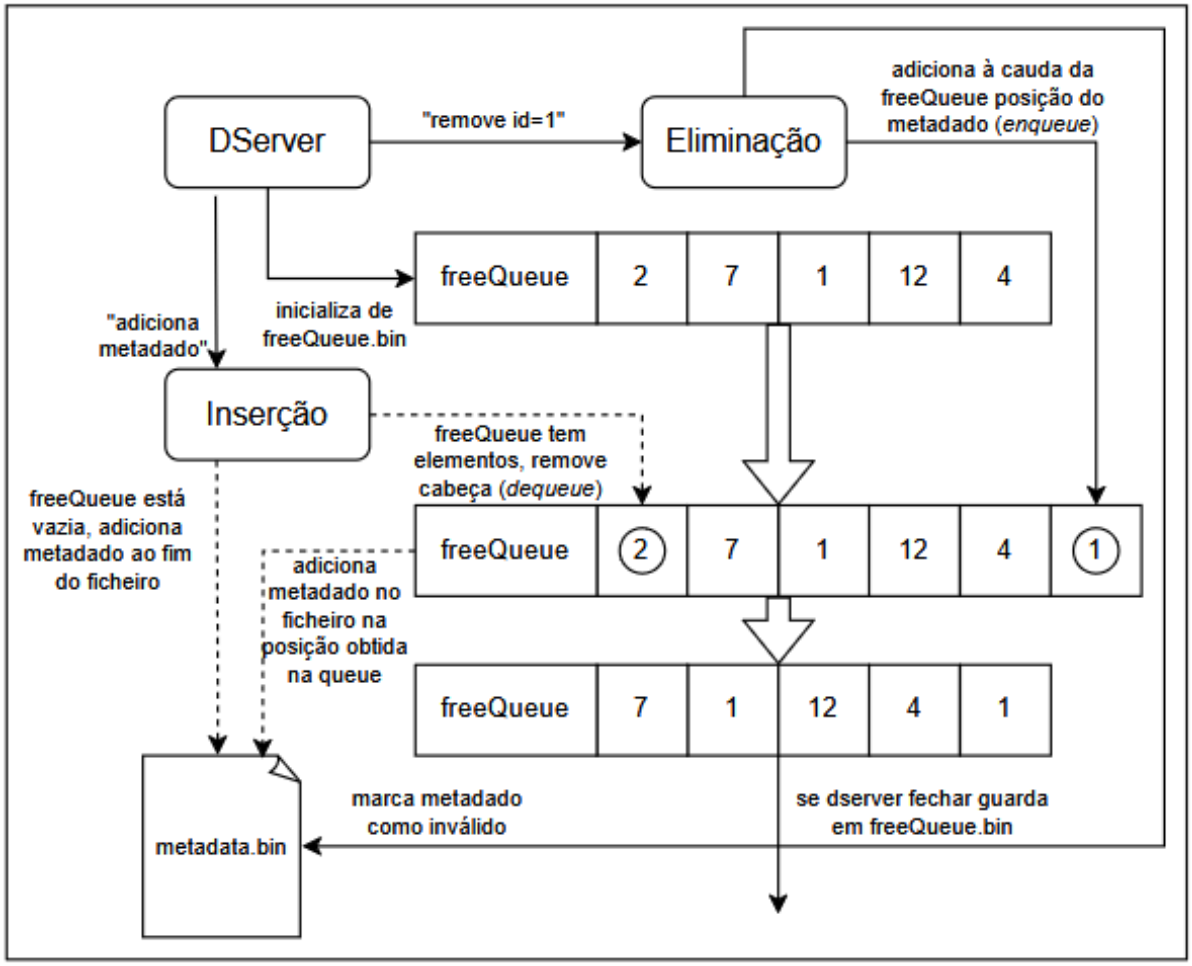
8. Anexos



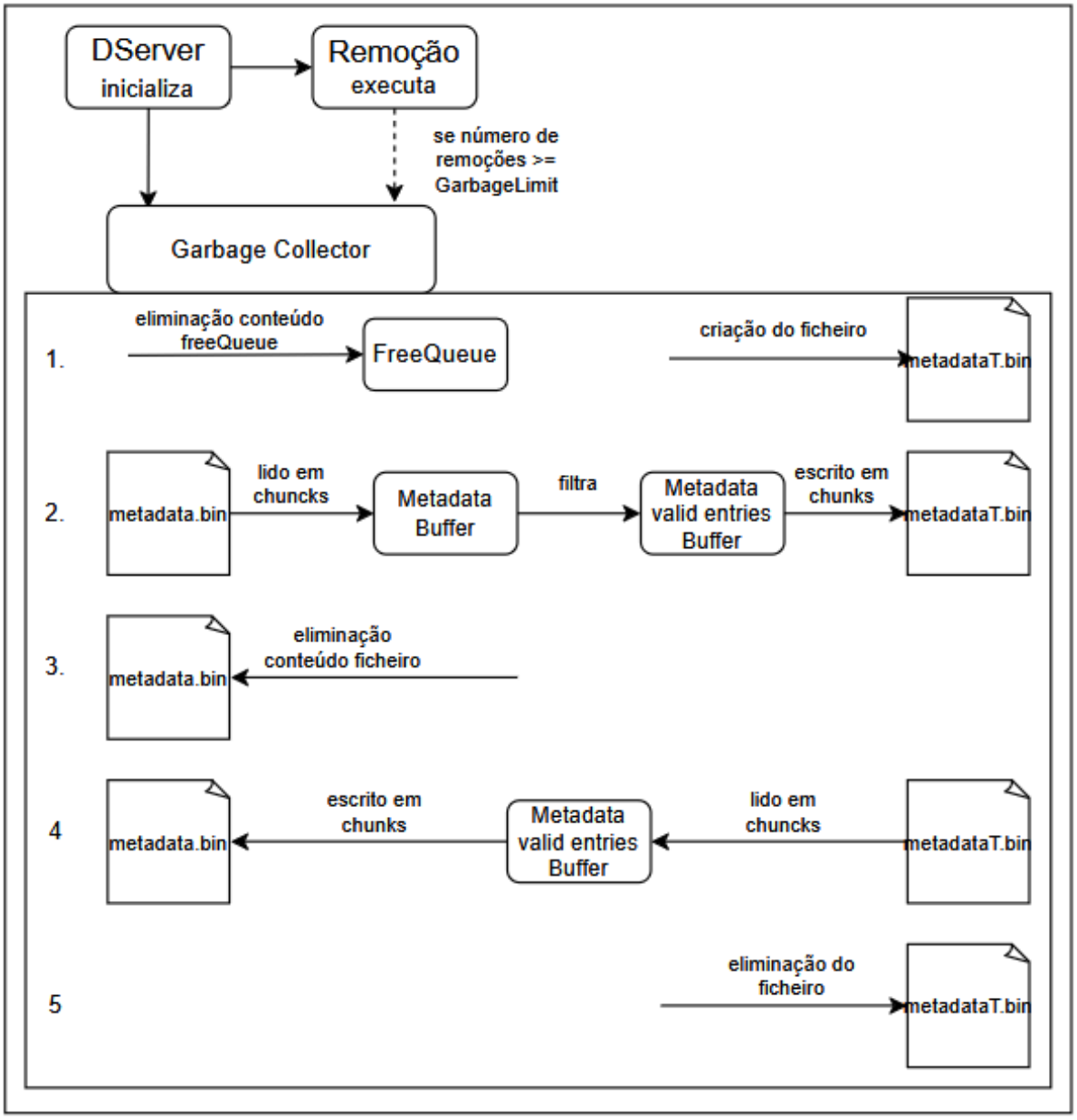
Anexo 1: Funcionamento do *Clock Algorithm*.



Anexo 2: Fluxo de execução da cache.



Anexo 3: Funcionamento da *FreeQueue*.



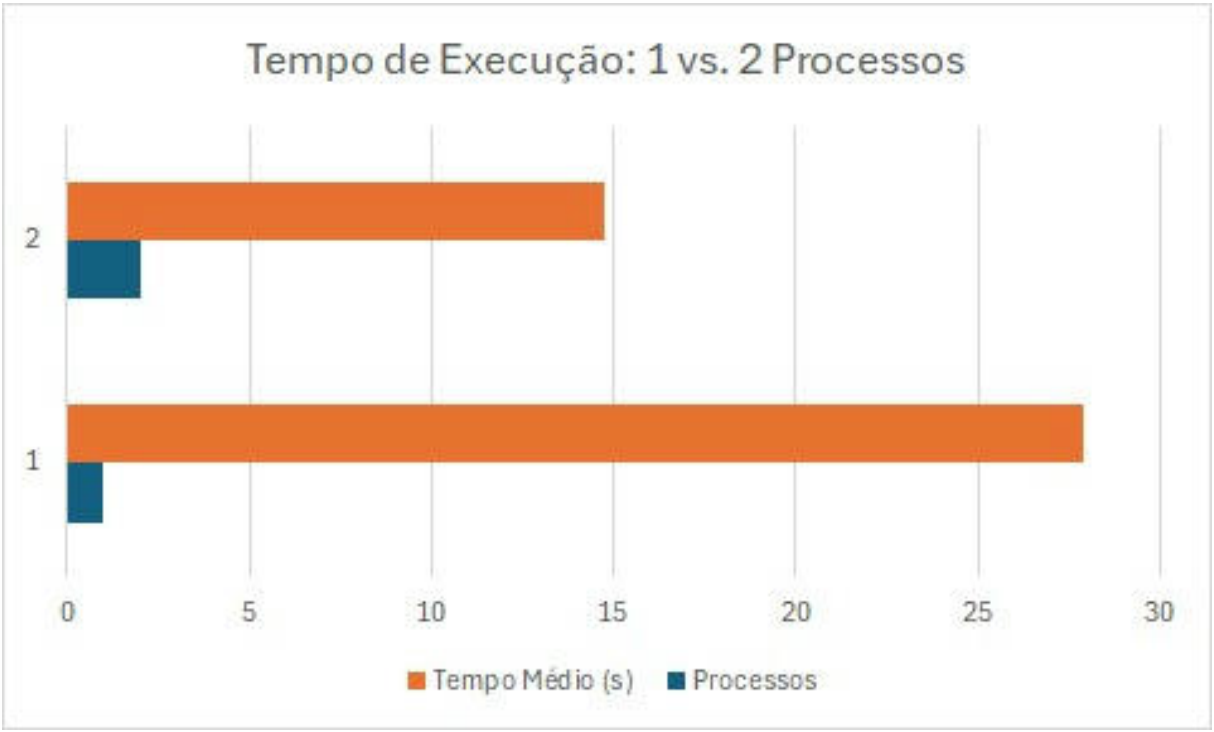
Anexo 4: Processo de execução do *Garbage Collector*.

1 Processo - Tempo (s)	2 Processos - Tempo (s)	Diferença - Tempo (s)
28.442	14.557	13.885
28.628	14.819	13.809
28.138	15.083	13.055
27.532	14.414	13.118
26.865	15.048	11.817

Tabela 1: Resultados dos Teste 1 (Paralelismo 1 vs. 2 Processos)

1 Processo - Tempo (s)	2 Processos - Tempo (s)	Diferença - Tempo (s)
27.921	14.7842	13.1368

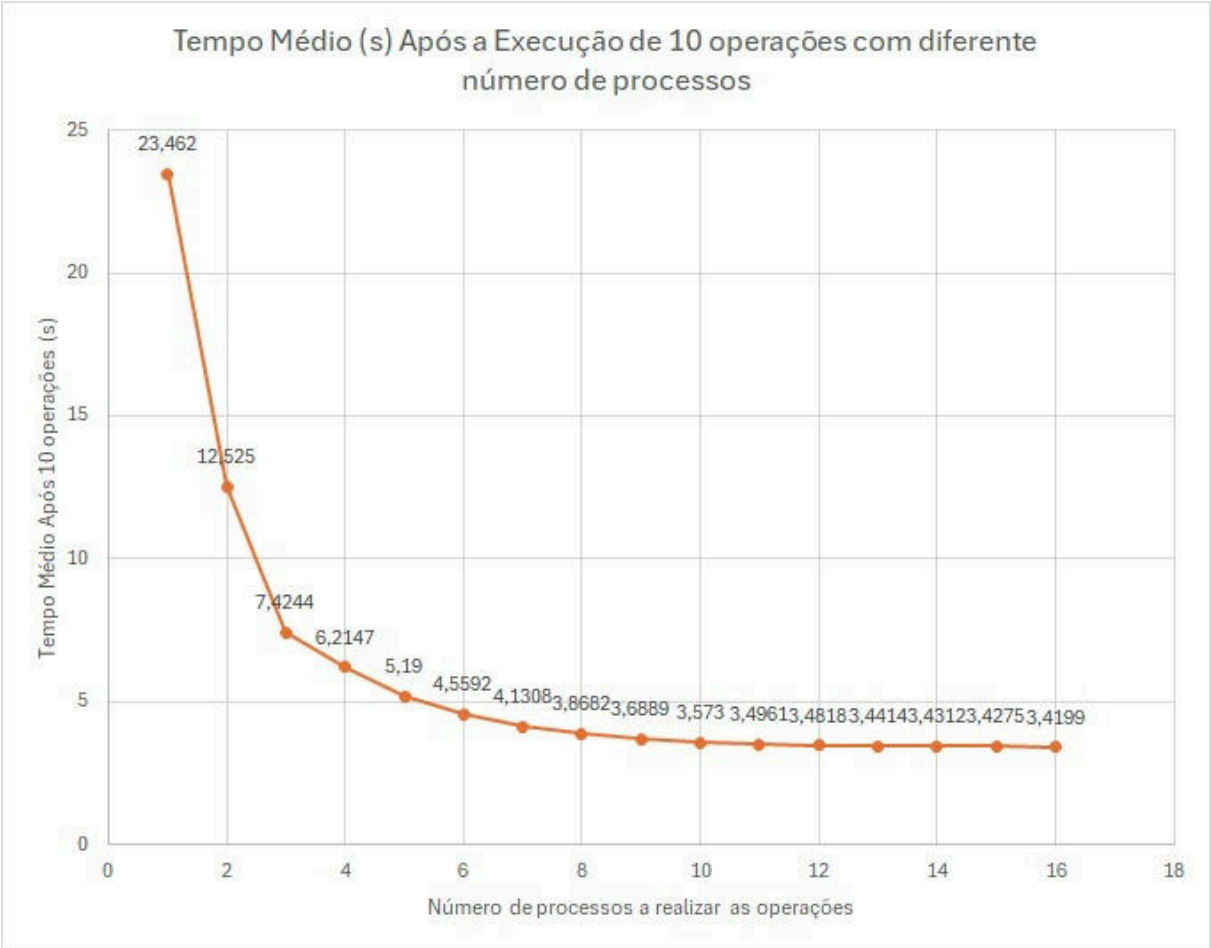
Tabela 2: Média dos Resultados dos Teste 1 (Paralelismo 1 vs. 2 Processos)



Anexo 5: Tempo de Execução: 1 vs. 2 Processos.

Processos	Tempo Médio (s)
1	23.462
2	12.525
3	7.4244
4	6.2147
5	5.19
6	4.5592
7	4.1308
8	3.8682
9	3.6889
10	3.573
11	3.4961
12	3.4818
13	3.4414
14	3.4312
15	3.4275
16	3.4199

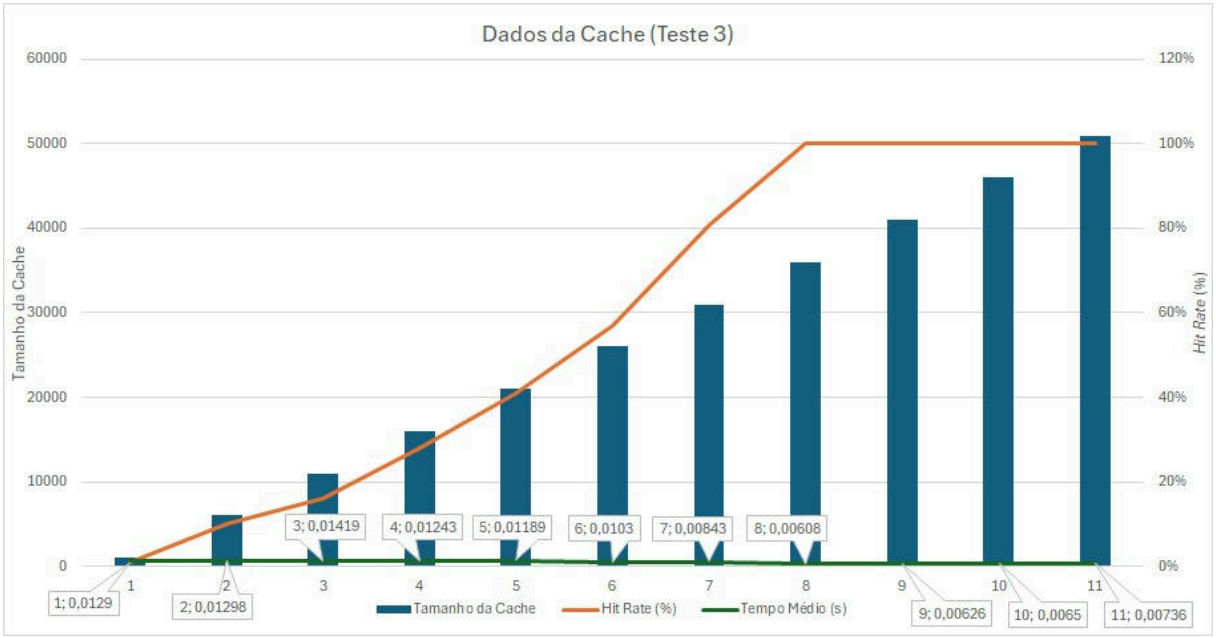
Tabela 3: Resultados dos Teste 2 (Escalabilidade com Processos Concorrentes)



Anexo 6: Escalabilidade com Processos Concorrentes.

Tamanho da Cache	*Hit Rate* (%)	Tempo Médio (s)
1000	1%	0.0129
6000	10%	0.01298
11000	16%	0.01419
16000	28%	0.01243
21000	41%	0.01189
26000	57%	0.0103
31000	81%	0.00843
36000	100%	0.00608
41000	100%	0.00626
46000	100%	0.0065
51000	100%	0.00736

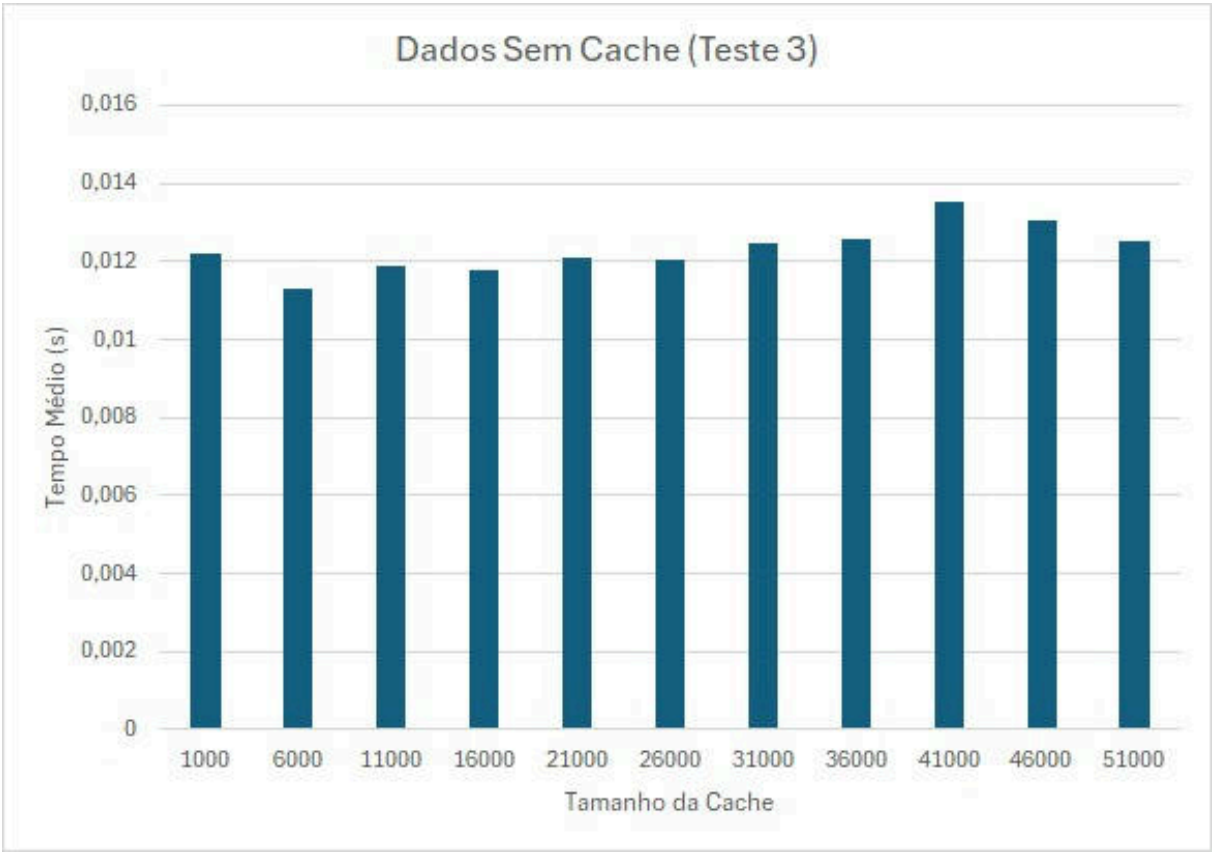
Tabela 4: Resultados dos Teste 3 (Dados Utilizando a Cache)



Anexo 7: Resultados do Teste 3 utilizando uma configuração com cache.

Tamanho da Cache	Tempo Médio (s)
1000	0.01218
6000	0.01129
11000	0.01189
16000	0.0118
21000	0.01207
26000	0.01203
31000	0.01245
36000	0.01259
41000	0.01355
46000	0.01307
51000	0.0125

Tabela 5: Resultados dos Teste 3 (Dados Sem o Uso da Cache)



Anexo 8: Resultados do Teste 3 utilizando uma configuração sem o uso cache.