Universidade do Minho

Braga, Portugal

SERVIÇO DE INDEXAÇÃO E PESQUISA DE DOCUMENTOS

Relatório do Trabalho Prático

Sistemas Operativos

Departamento de Informática

Engenharia Informática 2024/25

Grupo 37:

A106936 - Duarte Escairo Brandão Reis Silva

A106932 - Luís António Peixoto Soares

A106856 - Tiago Silva Figueiredo

17 Maio 2025

Índice

1. Introdução

No âmbito da Unidade Curricular de Sistemas Operativos, foi proposto um trabalho prático que visava desenvolver um serviço que permita a indexação e pesquisa sobre documentos de texto, guardados localmente num computador.

Assim sendo, o nosso grupo desenvolveu então o trabalho proposto. Tal como é descrito nos requisitos apresentados pelos docentes da unidade curricular, o programa deve ser capaz de implementar um canal de comunicação servidor-cliente, onde os vários clientes podem realizar operações de indexação, remoção e consulta nos documentos guardados.

Para além disso, deve ser também possível guardar o estado do servidor quando este é terminado (ou seja, quando é usado o comando para tal) e também que seja possível recuperar o estado do servidor que esteja guardado localmente em disco.

2. Descrição da Arquitetura

Nesta secção do relatório serão abordadas as estratégias usadas na implementação do nosso programa, desde a forma como o cliente envia os seus pedidos, bem como a forma que os diferentes pedidos são verificados e executados pelo servidor e ainda como está implementada a *cache*.

2.1. MetaDado

A estrutura `MetaDado` é a estrutura que circula no programa, entre o cliente e o servidor. Esta é constituída pela informação necessária quando se pretende guardar (quando se trata de uma indexação (`-a`)) os metadados de um documento no servidor.

Os seus campos não são todos preenchidos quando uma dessas estruturas é inicializada, pois por exemplo, numa opção `-c`, não faria sentido guardar o título, autores, etc. Assim sendo, os campos vão sendo preenchidos conforme necessário.

```
struct metaDados{
    // INFORMAÇÕES DO DISCO
    char titulo[MAX_TITULO];
    char autores[MAX_AUTORES];
    int n_autores;
    int ano;
    char path[MAX_PATH];
    int pos_in_disk;
    // INFORMAÇÕES DO CLIENTE
    char buffer[512];
    int argc;
    int pid;
    // INFORMAÇÕES DO SERVIDOR
    char flag;
};
```

Figura 1: Definição da estrutura `MetaDado`

2.2. Cache

A estrutura da `Cache` é uma estrutura implementada no programa do servidor, como uma otimização à solução inicialmente proposta, onde toda a informação era guardada na memória central. Com esta estrutura é possível fazer uma gestão mais complexa dos metadados, sendo que alguns se encontram na memória central (*cache*), enquanto que outros estão já escritos em disco. Esta implementação da *cache* requer também uma política para que esta fosse manipulada.

A estrutura apresenta diversos campos, dos quais:

- `dinamica`: um valor inteiro que indica se a *cache* tem tamanho estático ou dinâmico;
- `docs`: um *array* de `MetaDados` cujo tamanho pode ser definido previamente ou infinito (caso a *cache* seja dinâmica ou não);
- `stack_to_disc`: uma `Stack` que indica as posições em disco que estão livres;
- `stack_to_cache`: uma `Stack` que indica as posições na *cache* estão livres;
- `ocupados`: um *array* de tamanho infinito (faz *realloc* se necessário) que indica onde um `MetaDado` se encontra (EM_DISCO ou EM_CACHE);

- `size`: um valor inteiro que indica o número total de `MetaDados` que foram adicionados à *cache*;
- `capacity`: um valor inteiro que indica o número máximo de `MetaDados` que podem ficar na *cache*;
- `redimensionamento`: um valor inteiro que indica o número de redimensionamentos que já foram feitos nos tamanhos da `stack_to_disc` e no `ocupados`, havendo um maior controlo sobre o tamanho da memória que se quer alocar quando estes precisam de crescer;
- `next_to_disc`: um inteiro que é incrementado de maneira a garantir que cada `MetaDado` tenha um número de identificação único.

```
struct cache {
    int dinamica;
    MetaDados **docs;
    Stack *stack_to_disc;
    Stack *stack_to_cache;
    char *ocupados;
    int size;
    int capacity;
    int redimensionamentos;
    int next_to_disc;
};
```

Figura 2: Definição da estrutura `Cache`

2.3. Stack

A estrutura `Stack` está presente dentro da *cache* implementada no programa do servidor e é utilizada para gerir as posições que se encontram livres quer seja no disco, quer seja na própria cache. Esta estrutura possui um *array* que detêm as tais posições livres, a capacidade da estrutura, e o `top` que representa a próxima posição livre a ser retirada.

```
struct stack {
    int *positions;
    int top;
    int capacity;
};
```

Figura 3: Definição da estrutura `Cache`

2.4. Cliente

O programa cliente desenvolvido neste projeto está limitado a receber a informação que vai ser processada pelo programa servidor. Essa informação é apenas junta pelo próprio, e enviada no formato de um `MetaDado`.

Para além da informação que se encontra no *buffer* como: o título, os autores, o ano, e o *path* (onde o documento está guardado), o `MetaDado` é preenchido com o *pid* (identificador do processo) do programa cliente, isto para a resposta do servidor ser direcionada unicamente para um cliente em específico, e o *argc* (número de argumentos

passados), para facilitar a verificação da sintaxe do pedido quando este é lido pelo servidor.

2.5. Servidor

2.5.1. Verificação do pedido

Após a receção de um pedido pelo servidor, e antes da sua execução, é necessário validar a sintaxe do mesmo, de acordo com o formato estabelecido. Isto é feito verificando qual o comando enviado no pedido (`-a`, `-c`, `-d`, `-l`, `-s`, `-f`, `-b`) e o *argc* escritos no `MetaDado`.

2.5.2. Execução

Caso a sintaxe de um pedido recebido pelo servidor seja validada, o programa servidor avança agora para a sua execução. A implementação feita pelo grupo tenta ao máximo paralelizar este processamento, com recurso às *system calls*.

2.5.2.1. Indexação (-a)

No caso do pedido envolver a adição de uma nova entrada, o servidor separa os campos que se encontram no *buffer* e insere-os nos campos correspondentes dos `MetaDados`, p.e.:

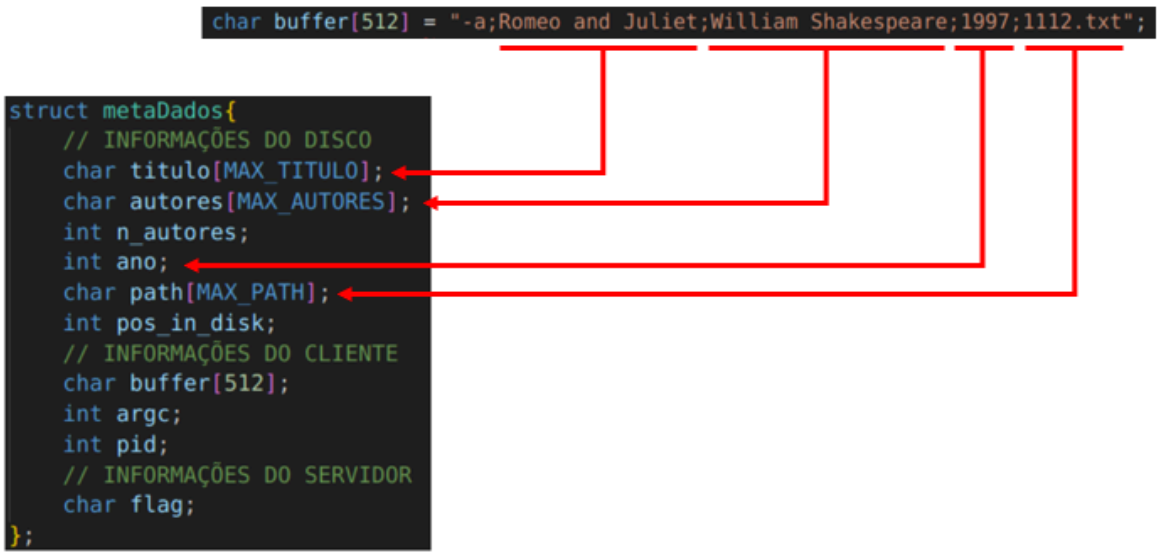

Figura 4: Exemplo da separação das informações de um *buffer* para os campos do `MetaDado`

Depois disto, insere o `MetaDado` com os campos preenchidos na *cache*.

2.5.2.2. Remoção (-d)

No caso do pedido envolver a remoção de uma entrada, o servidor verifica se a posição passada como argumento se encontra ocupada, e responde consoante. Caso esta exista, a *cache* responsabiliza-se por: caso esteja na *cache*, libertar a memória alocada (em memória) e adicionar a posição apagada à `stack_to_cache`, ou, caso esteja em disco, adicionar a posição apagada à `stack_to_disk`. Em ambos os casos, a cache atualiza

a posição, passada como argumento, do *array* `ocupados` para *LIVRE* (permitindo, posteriormente, inserir um `MetaDado` na mesma posição)

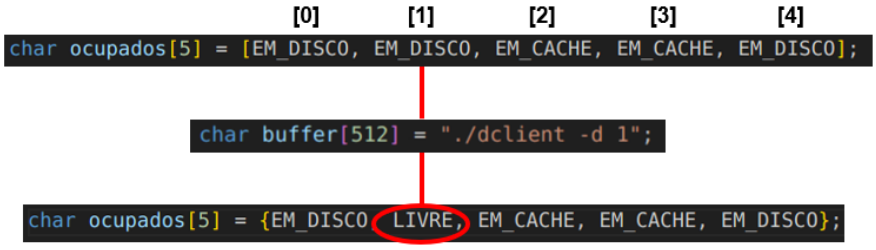

Figura 5: Exemplo da remoção na cache

2.5.2.3. Consulta (-c)

No caso do pedido envolver a procura de uma entrada, o servidor cria um processo filho que é responsável por verificar se a posição passada como argumento se encontra ocupada, respondendo consoante. Caso o `MetaDado` se encontre presente na *cache* é enviada a resposta para o cliente com a devida informação. Se o `MetaDado` se encontrar em disco, é trazido para a *cache* através de uma operação de indexação (`-a`) (contudo esta operação em concreto não enviará qualquer informação ao cliente). Depois do `MetaDado` estar em *cache* é então enviada a resposta da operação de consulta ao cliente.

2.5.2.4. Pesquisa (-s)

No caso do pedido envolver devolver uma lista de identificadores de documentos que contêm uma determinada palavra-chave, é criado um processo filho que pesquisa, no estado em que o servidor se encontra nesse momento, pela palavra-chave. Caso não seja passado o argumento opcional (o número de processos para efetuar a pesquisa de forma paralela), é usado 1 processo apenas. Caso contrário, são criados **N** processos que pesquisam sobre (assumindo, p.e., que a `size = 100`) 100/ **N** documentos (sendo **N** o argumento opcional passado pelo cliente).

2.5.2.5. Pesquisa (-l)

No caso do pedido envolver devolver o número de linha que uma dado documento que contêm uma dada palavra-chave, é criado um processo filho que pesquisa, nesse dado documento, pelas linhas em que essa palavra-chave ocorre. Depois, é enviada a resposta correspondente ao cliente.

2.5.2.6. Encerramento (-f)

No caso do pedido envolver o encerramento do servidor, o próprio chama a função `all_Cache_to_Disc()`, que é responsável por escrever para disco todos os `MetaDados` presentes na *cache*, além de também de enviar um `MetaDado` especial, com o campo `pos_in_disk` com valor -2, para um *fifo chamado DISK_WRITER_FIFO* (ver **2.6. Persistência**). Por fim, notifica o cliente que fez o pedido de que o o servidor foi encerrado.

2.5.2.7. Restauração (-b)

Como é referido na **Introdução**, um dos requisitos dos docentes da unidade curricular para este trabalho, é a possibilidade de recuperação do estado de servidor. Para isso,

o nosso grupo decidiu adicionar um novo comando ao programa servidor, que pode ser executado através de um pedido do cliente com o comando:

```
./dclient -b "path"
```

No tipo de pedido, o servidor chama a função `recupera_backup()`, que, em primeiro lugar, vai colocar a *cache* no estado em que a mesma se encontrava no momento que o programa servidor foi chamado, e, em segundo lugar, vai ler os dados contidos no ficheiro binário, onde o estado está guardado, e popular a *cache* de forma a que ela volte ao seu estado anterior.

2.6. Persistência

Após iniciar-se o programa servidor, o mesmo cria um processo filho que ficará responsável pela persistência dos metadados dos documentos em disco, até ao encerramento do mesmo. Este processo filho chama a função `write_to_disk()`, que é responsável por ler estruturas de dados do tipo `MetaDado`, que contêm os metadados dos documentos a serem armazenados em disco, de um *fifo* especial chamado *DISK_WRITER_FIFO*.

```
pid_t pid2 = fork();
if (pid2 == 0) {
    write_to_disk();
    _exit(0);
}
```

Figura 6: Parte do código referente à explicação

Existem três casos concretos em que `MetaDados` podem ser enviados para esse *fifo*, sendo eles:

- Falta de espaço na *cache*, o que pode levar à substituição de um `MetaDado`;
- O encerramento do servidor, que obriga a que os `MetaDados` presentes na *cache* sejam escritos em disco;
- Restauração de um estado anterior, onde é necessário voltar a escrever determinados `MetaDados` em disco.

Para além disso, para que o processo filho saiba quando parar de ler `MetaDados`, foi adicionada uma verificação especial que garante que que caso a função `write_to_disk()` leia um `MetaDado` com o campo `pos_in_disk` com valor igual a -2, algo que seria normalmente impossível, a execução dela termine e o processo chama a *system call exit*, permitindo que ele seja devidamente terminado durante o encerramento do servidor.

2.7. Envio da resposta

Após a execução de um pedido, é necessário enviar a resposta ao cliente correspondente. Para isto, as funções responsáveis pela execução dos pedidos criam uma *String* com a resposta, e chamam a função `envia_resposta_cliente()` que escreve, no *fifo* correspondente criado pelo cliente (usando o *pid* do `MetaDado`), a resposta.

No caso de um pedido falhar na fase de validação, é chamada a função `error_message()`, que envia uma mensagem de erro para o cliente correspondente, dependendo do tipo do pedido do mesmo.

2.8. Cache de tamanho dinâmico

Como também é referido nos requisitos do enunciado, deveríamos otimizar o nosso programa servidor de forma a incorporar a funcionalidade de ***Caching***. Para isso, implementamos uma *cache* que permite guardar apenas um limite máximo de entradas de metadados de documentoas, onde esse mesmo limite é obtido através de um dos argumentos usados ao iniciar o servidor.

De modo a expandir ainda mais o nosso projeto, decidimos adicionar uma funcionalidade extra, onde, caso o limite máximo da *cache* não seja passado como argumento, usamos uma *cache* infinita, ou seja, sem um limite máximo para o número de entradas de metadados de documentos, onde a mesma começa com tamanho igual a 10 e conforme necessário é expandida para o dobro do tamanho anterior usando *realloc*.

```
int cache_size;

if (argv[2] == NULL) {
    flag = CACHE_DINAMICA;
    cache_size = 10;
} else {
    flag = CACHE_ESTATICA;
    cache_size = atoi(argv[2]);
}
```

Figura 7: Exemplificação das opções de *Caching*

Esta *cache* infinita funciona de forma similar à *cache* com limite superior, com a diferença que, como a mesma nunca fica cheia não existe a necessidade de escrever para disco ao adicionar novas entradas ou ao fazer a restauração de um estado a partir de um ficheiro, como foi referido em **Persistência**.

Além disso, ao adicionar novas entradas, embora não seja mais necessário enviar `MetaDados` para disco, existe outra diferença em relação à *cache* com tamanho estático, sendo ela o facto das *stacks* não serem utilizadas, visto que, como nunca irão existir entradas de `MetaDados` em disco não existe a necessidade de utilizar a `stack_to_disk` e, diferente da *cache* estática, a *cache* com tamanho dinâmico possui uma lógica de adição diferente, onde a mesma começa a procurar desde a posição 0 por posições livres para ocupar, acabando com a utilidade da `stack_to_cache`.

3. Testes

3.1. Testes de Comandos

3.1.1. Testes de Indexação

Para conseguir perceber melhor o impacto que o tamanho da *cache* tem quando se indexa a informação, foram realizados 3 testes, tendo todos eles realizado a indexação de 100 000 documentos. A única diferença foram os tamanhos das *caches* usadas, que foram, respetivamente: 1 000, 10 000 e 100 000.

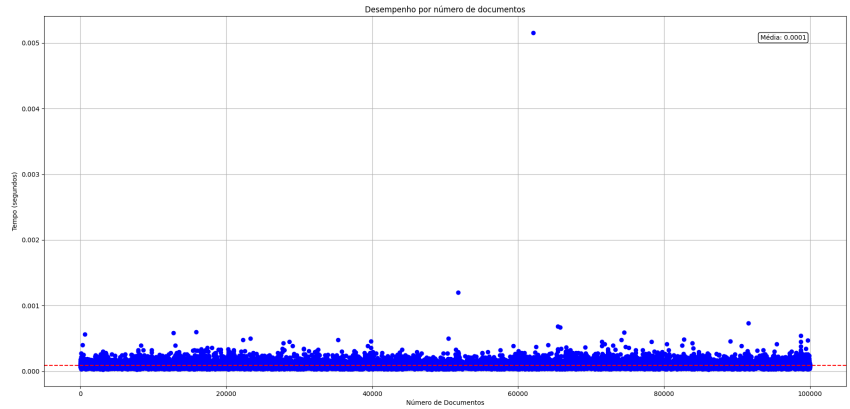

Figura 8: Teste da *Cache* com 1 000 entradas

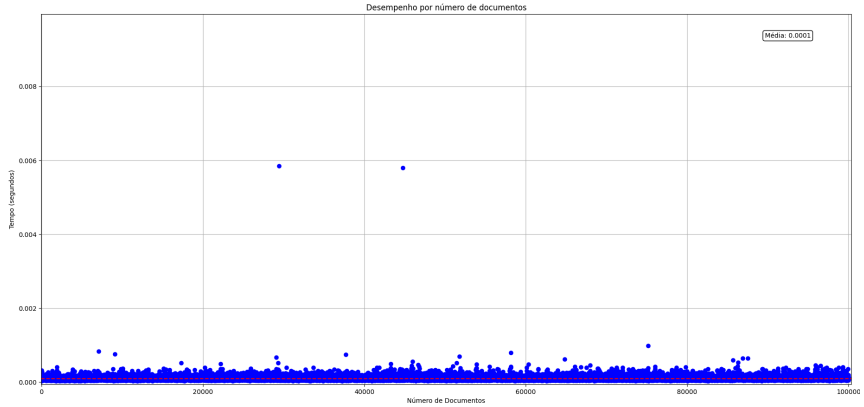

Figura 9: Teste da *Cache* com 10 000 entradas

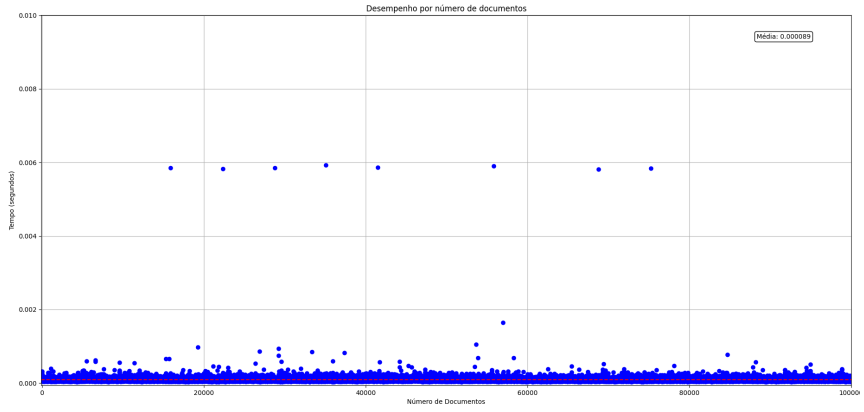

Figura 10: Teste da *Cache* com 100 000 entradas

Como é possível verificar pela observação dos gráficos, o tempo de resposta ao cliente quando se faz uma indexação, pouco varia. Isto deriva do facto de que quando o servidor lê um `MetaDado` e realiza as operações necessárias, se for necessário escrever algo em disco, existe um processo responsável por tal. Logo, como o servidor apenas precisa de inserir um `MetaDado` novo na sua *cache*. É de notar que numa fase inicial a estratégia de indexação era diferente e os resultados obtidos nos testes eram diferentes. Para melhorar os tempos de resposta aos clientes, o grupo alterou o código. Para a justificação da abordagem tomada, ver **4.2 Não Paralelização da Indexação**

3.1.2. Testes de Procura

Para perceber melhor a resposta dos processos relativamente às operações de procura, foram feitos três teste, todos com o tamanho da *cache* fixo (1 000 de tamanho), onde se avaliou o desempenho da operação com 1 processo, até 50. Nos diferentes testes foram alterados os números de indexações, 1 000, 10 000 e 100 000.

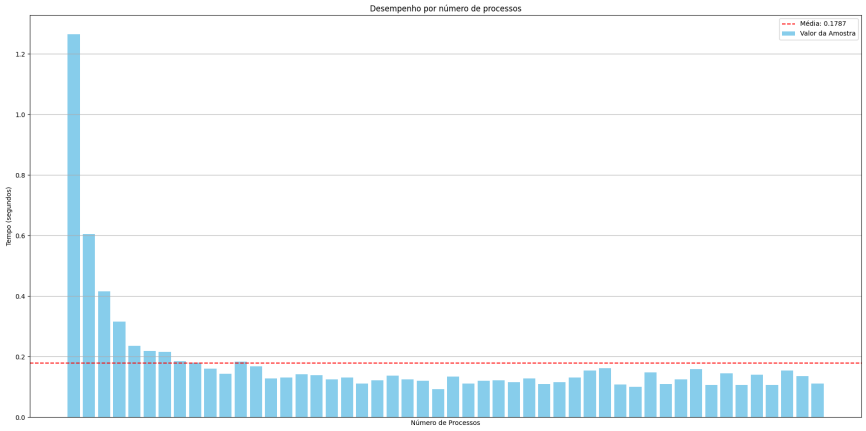

Figura 11: Teste da *Cache* com 1 000 indexações

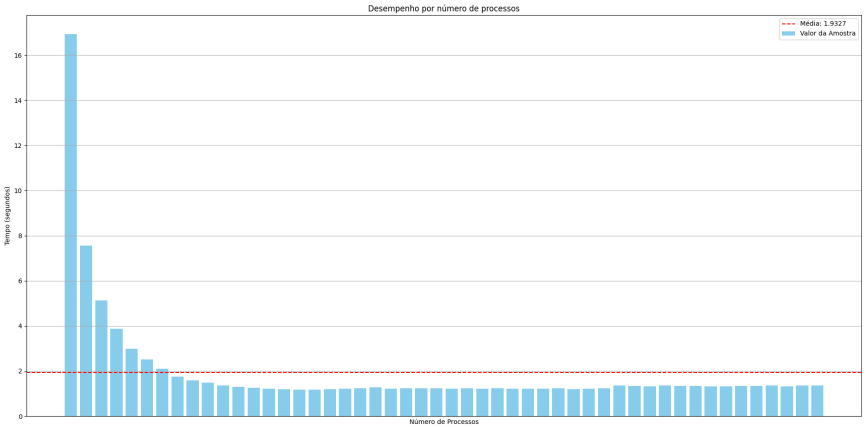

Figura 12: Teste da *Cache* com 10 000 indexações

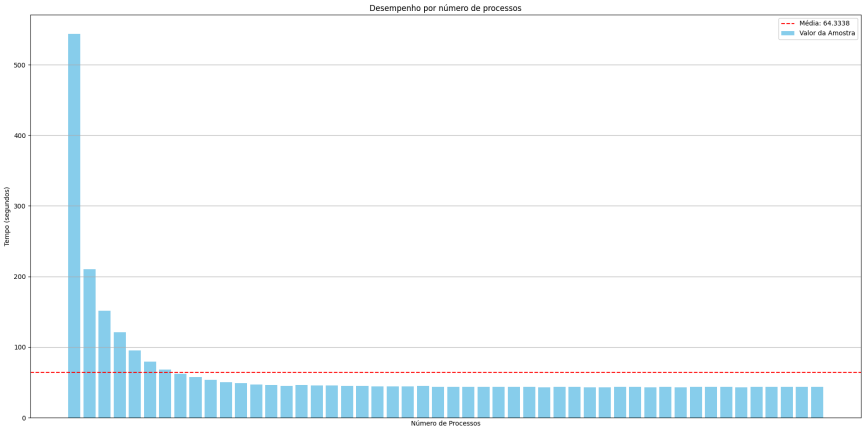

Figura 13: Teste da *Cache* com 100 000 indexações

Como é possível perceber pelos gráficos, houve um claro aumento do tempo de procura entre cada um dos testes, o que é justificado pelo número de documentos pelos quais é necessário iterar. Para além disto, o uso de processos que fazem a procura de forma paralela melhoram, obviamente, o tempo de procura. Mas isto até atingir o *bottleneck* limitado pela capacidade de processamento do computador que realizou os testes. Isto é, a partir de um certo número de processos, o tempo de procura pouco varia pois o processador não tem capacidade de realizar as operações de cada um dos processos de forma mais rápida.

3.2. Testes de Políticas de Caching

Devido ao facto de não termos implementado mais nenhuma **Política de Caching**, não foi possível fazer testes nesse sentido.

4. Decisões Importantes

4.1. Cliente envia MetaDado

A estrutura criada pelo cliente é um `MetaDado` por causa do funcionamento do servidor. Como quando o servidor lê um comando de indexação (`-a`), a estratégia implementada pelo grupo implica o reenvio dos `MetaDados` com as informações guardadas em disco pelo servidor, então, o mesmo terá de ler estruturas de dados do tipo `MetaDado`, justificando o envio destas mesmas estruturas por parte do cliente.

4.2. Não Paralelização da Indexação

Numa fase inicial, a operação de Indexação era feita usando um processo filho (usando a *system call 'fork()'*). Isto garantia que o servidor não tinha de fazer as operações indicadas em **2.5.2.1. Indexação (-a)**, e estava sempre pronto a ler novos pedidos, sendo estes processados em paralelo. Com o uso do filho para a realização das operações, era também assegurado que o cliente não bloqueava o lado de escrita do *fifo* de comunicação com o servidor. O resultado dessas operações era depois enviado novamente para o servidor.

Apesar disto, o nosso grupo decidiu alterar esta estratégia. Com isto, o servidor passou a ser o responsável pela manipulação dos `MetaDados`, que este lê do seu *fifo*, já que, após a realização de vários testes por parte do grupo, foi possível notar que, perante uma carga de trabalho grande (indexação de um grande número de MetaDados), o tempo de realização de um *fork()* aumenta linearmente, levando a tempos de resposta cada vês maiores. Concluímos ainda que, a partir de um certo ponto, o tempo de realização da chamada *fork()* excede o tempo que demoraria a processar, se fosse o servidor a alterar os `MetaDados` ele mesmo. Para além disto, quanto maior fosse o tamanho da *cache* usado, mais rápido era o aumento do tempo de resposta

4.3. Paralelização das Procuras

Quando o servidor recebe um comando de pesquisa ou de consulta, ele paraleliza a execução desse pedido de maneira a poder receber um comando novo logo de seguida. Isto poderá levar a casos em que indexações que ainda não tenham sido terminadas, não sejam contabilizadas quando se faz a pesquisa ou procura. Por outro lado, garante

que diferentes clientes que façam este tipo de operações obtenham respostas da forma mais rápida possível (por realizar o processamento de pedidos deste tipo em paralelo.)

4.4. Política de *Caching*

Usando o conhecimento obtido nas aulas teóricas da UC, o grupo desenvolveu uma **Política de Caching** semelhante à FIFO. Em geral, é atribuído ao `MetaDado` uma posição em disco e, a partir dessa posição, a posição em *cache* será sempre a mesma. Imaginando que o servidor recebe um pedido de indexação de um `MetaDado` e a *cache* desse servidor tem o tamanho 12. Supondo que já existem, no total, 56 documentos indexados, ao novo `MetaDado` seria atribuído o número 57. Com isto, a sua posição em *cache*, isto é, a posição para onde irá ser inserido em qualquer operação, será a posição $57 \bmod 12 = 9$. Para fazer isto, sempre que se adiciona um documento novo, é usado um contador que incrementa. É de notar que apenas é atribuída uma posição em disco ao `MetaDado` caso seja necessário escreve-lo em disco.

Para além disto, sempre que é necessário mapear um `MetaDado` para a *cache* apenas é necessário calcular a sua posição usando a fórmula $\text{pos} \bmod \text{tam\_cache}$, e inserir o `MetaDado` na posição calculada. Caso seja necessário inserir um `MetaDado` em disco, apenas é necessário consultar a posição para onde foi mapeado.

5. Conclusão

Em suma, o nosso grupo foi capaz de desenvolver o serviço de indexação e pesquisa sobre documentos de texto, cumprindo todos os requisitos estipulados no enunciado do trabalho prático, e conseguimos até estender alguns conceitos que não foram propostos.

Ao longo do desenvolvimento do trabalho, o projeto sofreu várias alterações, tendo tido várias versões do mesmo, e isto aconteceu porque nos fomos apercebendo que as implementações feitas não eram as melhores nem mais eficazes, e procuramos sempre melhorar.

Achamos que a versão final apresentada, responde de uma forma muito consistente ao que é pedido, e por isso estamos bastante satisfeitos com o resultado.

Contudo, com vista a melhor aplicabilidade em casos de uso reais, a **Política de *Caching*** poderia ser alterada para uma outra. Isto porque, tal como foi estudado na aulas teóricas da UC, esta política não é a mais eficiente quando comparada com outras políticas como por exemplo a **LRU**. No entanto, tendo em conta a dimensão do projeto e dado o facto da implementação da *cache* ter sido algo feito muito cedo no projeto, não sentimos necessidade de a alterar, pois os resultados que seriam de esperar foram obtidos.