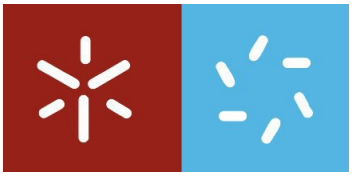

Universidade do Minho

Licenciatura em Ciências da Computação

Sistemas Operativos - Trabalho Prático
Relatório

Andre Filipe Dourado Pinheiro
(A108473)

João Francisco Sousa Oliveira
(A108647)

Pedro Dong Mo
(A108398)

17 de maio de 2025

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Sistemas Operativos, foi-nos solicitado a implementação de um serviço que permita a indexação e pesquisa sobre documentos de texto guardados localmente num computador, no qual programa servidor é responsável por registar meta-informação sobre cada documento (p.ex, identificador único, título, ano, autor, localização), permitindo também um conjunto de interrogações relativamente a esta meta-informação e ao conteúdo dos documentos.

Capítulo 2

Estrutura

A estrutura consiste num cliente (programa *dclient*) para ser usado pelo utilizador via linha de comandos e um servidor (programa *dserver*), com o qual o programa cliente irá interagir, este tem que ser inicializado com dois parâmetros, o *path* da pasta onde se encontram os documentos e o tamanho da *cache* .

2.1 Cliente (*dclient.c*)

No processo de comunicação entre o cliente e o servidor, os argumentos enviados pelo cliente são tratados e armazenados numa estrutura do tipo `MSG`, como mostra a figura 2.1.

```
typedef struct msg {
    char flag[3];
    char argv[4][200];
    int argc;
    pid_t pid;
    off_t offset;
} MSG;
```

Figura 2.1: Estrutura de dados `MSG`

Quando o cliente realiza um pedido, os seus argumentos são organizados da seguinte forma:

- **flag:** identificação do tipo de pedido
- **argv:** armazena até quatro argumentos, cada um com um tamanho máximo de 200 caracteres
- **argc:** armazena a quantidade de parâmetros introduzidos pelo cliente

- **pid:** identificador do processo do cliente, permitindo ao servidor reconhecer a origem do pedido.
- **offset:** utilizada na comunicação entre o servidor e processos-filho que este gera.

Após o preenchimento da estrutura, esta é escrita no FIFO localizado em `/fifos/C_TO_S`, criado pelo programa servidor quando este é executado, permitindo assim o envio de mensagens do cliente para o servidor.

Posteriormente, é criado um FIFO específico para a resposta do servidor para um determinado cliente, baseado no PID do mesmo. Em seguida, a resposta do servidor é lida e apresentada no terminal do cliente.

Este método permite que cada pedido seja enviado de forma organizada, contendo todas as informações necessárias para o servidor processar corretamente a operação solicitada.

2.2 Servidor (*dserver.c*)

2.2.1 Indexações em memória

Utilizamos uma árvore binária balanceada de procura para armazenar as indexações em memória, sendo a chave de cada indexação uma *string* correspondente ao **título** da indexação. As indexações são estruturas de dados definidas como na figura 2.2.

```
typedef struct index{
        char title[200];
        char authors[200];
        char path[64];
        int year;
} Index;
```

Figura 2.2: Estrutura de dados `Index`

Quando o servidor é executado, o mesmo começa por criar a árvore binária no qual serão guardadas as indexações e posteriormente a mesma é preenchida com indexações em disco.

2.2.2 Recolha das requisições

Quando é feita uma requisição a partir de um certo cliente, a informação é lida e com base na *flag* e os conteúdos dos argumentos, o servidor identifica a operação a ser realizada. Com base nisso, a função apropriada definida em `services.c` é invocada para processar o pedido. Esta abordagem permite que diferentes tipos de pedidos (como indexação, consulta ou remoção de dados) sejam tratados de forma organizada e separada.

2.2.3 *Caching*

Para a implementação do *caching*, usamos a política do FIFO (*First In First Out*) onde usamos uma *Queue* auxiliar com os endereços tal que, quando é atingido o número máximo de indexações, efetua-se *dequeue* à *Queue* e efetua-se *evict* à essa indexação da árvore em memória.

Para esta implementação, precisamos também de alterar o que cada *flag* faz e como esta influencia a política.

- Na opção `-a`, ocorre adição da indexação à árvore, faz *enqueue* da mesma e adiciona em disco, ou seja, todas as indexações estão em *cache* e estão em disco, que se situa no ficheiro `/saves`.
- Nas opções `-c` e `-l`, os processos filho comunicam com o processo pai (servidor) se:
  - a procura resultar num *miss*;
  - a indexação a ser procurada existir em disco para o servidor executar *enqueue* da mesma;
  - para indexar em memória.

  Isto é feito através do FIFO `C_TO_S` com a estrutura `MSG` e com uma *flag* personalizada *r*.
- Na opção `-d`, o servidor apenas irá diretamente para o disco remover a indexação e remover da memória se existir.
- Na opção `-s`, não há influência na questão da *cache* uma vez que visita todas as indexações e vai diretamente ao disco para ler as mesmas.
- Na opção `-f`, não há influência na questão da *cache*, pois todas as indexações já estão guardadas em disco.

2.3 Serviços (*services.c*)

No ficheiro `services.c` estão as funções que vão processar os pedidos feitos pelo cliente e traduzidos pelo servidor.

2.3.1 *flag* -a

Quando é feito um pedido com a *flag* `-a`, o servidor, em primeiro lugar, aguarda que todos os processos filho terminem, caso existam. De seguida, aloca memória para a estrutura de indexação e preenche uma estrutura `Index` com os argumentos fornecidos pelo cliente. Posteriormente, invoca a função `indexDocument`, à qual são passados como argumentos: a árvore binária, a estrutura recém-criada, o descritor de escrita utilizado para enviar

informação ao cliente, o descritor de leitura e escrita do ficheiro que contém as indexações em disco, uma *queue* que indica a próxima indexação a ser retirada da *cache*, e, por fim, o número de indexações atualmente presentes na árvore (*cache*). Esta função começa por verificar se a indexação já existe no ficheiro em disco. Caso exista, é enviada ao cliente uma mensagem a indicar que a indexação já existe. Caso contrário, a indexação é guardada no ficheiro em disco, inserida na árvore (*cache*) e adicionada ao final da *queue*. Se a *cache* atingir a sua capacidade máxima, a indexação que se encontra no topo da queue é removida da *cache* (e também da *queue*). Após a conclusão do processo, é enviada ao cliente uma mensagem a indicar que o documento foi indexado com sucesso, e a função retorna o número atualizado de indexações na árvore.

2.3.2 *flag* -c

Quando é feito um pedido com a *flag* `-c`, o servidor cria um processo filho que guarda a indexação fornecida pelo cliente e invoca a função `checkKey`. Esta função recebe como argumentos a árvore binária (*cache*), a indexação e o descritor de escrita, utilizado para enviar informações ao cliente que realizou o pedido. A função pesquisa primeiro na *cache* pela indexação. Se esta for encontrada, o conteúdo correspondente (meta-informação) é enviado ao cliente. Caso contrário, é efetuada uma pesquisa no disco. Se a indexação for encontrada no disco, procede-se do mesmo modo, enviando os dados para o cliente. Se não for encontrada, é enviada uma mensagem a informar que a indexação não existe. No final o pid do processo filho é adicionado a uma lista `pides`.

2.3.3 *flag* -d

Quando é feito um pedido com a *flag* `-d`, o servidor, em primeiro lugar, aguarda que todos os processos filho terminem, caso existam. Em seguida, guarda a indexação fornecida pelo cliente e invoca a função `checkKey`. Esta função recebe como argumentos a árvore binária (*cache*), a indexação, o descritor de escrita utilizado para enviar informação ao cliente, o descritor de leitura e escrita do ficheiro que contém as indexações em disco e, por fim, uma *queue* que indica a próxima indexação a ser retirada da *cache*. A função procura a indexação na árvore. Se a indexação existir, esta é removida da árvore binária, da *queue* e do ficheiro em disco, sendo enviada ao cliente uma mensagem de sucesso. O ficheiro é então reconstruído de forma a evitar fragmentação interna. Caso a indexação não exista, é enviada uma mensagem a indicar que a mesma não foi encontrada. No final a função retorna o número atualizado de indexações na árvore.

2.3.4 *flag* -l

Quando um pedido com a *flag* `-l` é feito para o servidor, este guarda a indexação e a palavra-chave provida do cliente e invoca uma função `searchKeywordByKey`. Esta recebe como argumentos a árvore binária, a indexação, a palavra-chave e o descritor de escrita para enviar informações para o cliente que realizou o pedido. Antes da função desempenhar sua funcionalidade, esta verifica se a chave da indexação se encontra em *cache* (árvore binária) e caso não encontre, é procurado em disco pela mesma. Quando a chave da indexação é encontrada, é executado um comando que procura as linhas no documento indexado em que a palavra-chave aparece e em seguida conta o número de linhas, sendo o comando `grep <word> <path> | wc -l` a efetuar este processo, no qual `<word>` corresponde à palavra-chave e `<path>` corresponde ao caminho para o documento, informação extraída a partir do conteúdo da indexação. Após a execução do comando, a resposta do mesmo é enviada para o cliente.

2.3.5 *flag* -s

Quando um pedido com a *flag* `-s` é feito para o servidor, este guarda a palavra-chave, o número de processos providos do cliente (caso o número de processos não seja fornecido, o valor 1 é definido por padrão) e invoca uma função `searchKeyword`. Esta recebe como argumentos a palavra-chave, o número de processos (seja este valor N) e o descritor de escrita para enviar informações para o cliente que realizou o pedido. Em seguida, são criados N processos e N pipes anónimos, um para cada processo. Cada processo executará a função `foreachIndex`, que recebe como argumentos a palavra-chave a ser procurada, o valor N, o valor de ordem do processo e os descritores do pipe desse respetivo processo.

Cada processo vai ler um segmento de texto em cada documento indexado em disco, sendo a quantidade de bytes desse segmento dividia igualmente por entre os processos, tal como é ilustrado no exemplo da Fig. 2.3. A divisão desses segmentos é feita através do valor de ordem de cada processo.

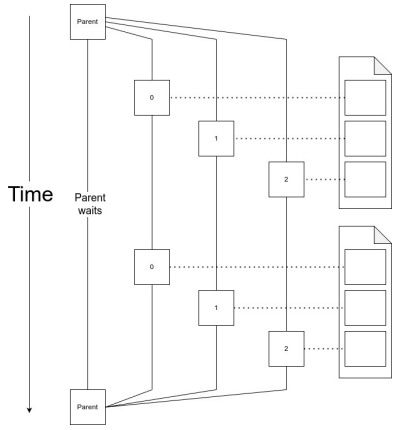

Figura 2.3: Exemplo do funcionamento da opção `-s` com $N = 3$ e dois documentos indexados

Por exemplo, se um documento possuir M bytes e N processos estão a ler o mesmo documento. Então o processo com valor de ordem i irá ler o segmento que começa na posição $i \cdot (M/N)$ até à posição $i \cdot (M/N) + M/N + \delta - 1$, em que δ corresponde à quantidade de bytes da palavra-chave, de forma evitar que os bytes da palavra entre os segmentos não sejam lidos.

Após um processo ler o segmento de bytes de um certo documento, este verifica se o segmento contém a palavra-chave e se o contiver, é escrito no seu pipe para o processo pai o tamanho da chave da indexação e em seguida é escrita a chave da indexação, permitindo assim que o processo pai possa ler as chaves das indexações que foram escritas.

Após todos os filhos terminarem de ler os documentos, o processo pai cria uma `g_hash_table` que irá funcionar como um *set* e irá inserir as chaves lidas a partir dos pipes para esse *set*, evitando assim chaves duplicadas.

Por fim, as chaves no *set* serão enviadas para o cliente que fez um pedido.

2.3.6 *flag* -f

Quando é feito um pedido com a *flag* `-f`, o servidor, em primeiro lugar, aguarda que todos os processos filho terminem, caso existam.

De seguida, envia uma mensagem ao cliente a informar que o servidor está a ser desligado. Posteriormente, invoca duas funções: `g_queue_clear` e `g_tree_destroy`, definidas na biblioteca `GLib`, para libertar os recursos associados à *cache*. Por fim, o servidor fecha o descritor `dummy_fd` do FIFO `C_TO_S`, que mantém o servidor à espera de novos pedidos provenientes dos clientes.

Capítulo 3

Avaliação experimental

Os dados para a avaliação experimental foram extraidos através de *scripts*, que podem ser encontrados em `/scripts`. Antes da obtenção dos dados, foram inicialmente indexadas 1000 documentos.

3.1 Paralelizar a pesquisa de documentos

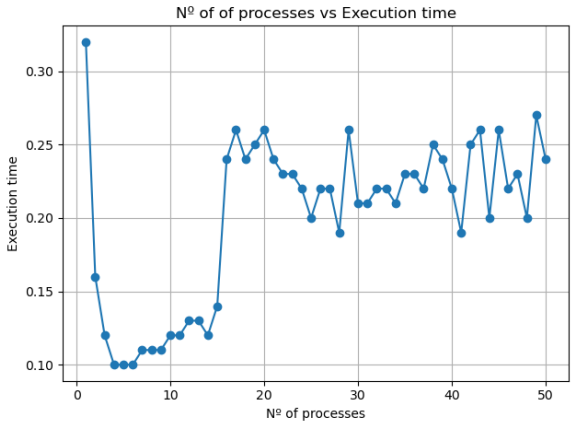

Figura 3.1: Tempo de pesquisa *vs* número de processos. Resultado obtido com 1000 documentos indexados, sendo a palavra-chave *geo* a ser procurada.

Através da Fig. 3.1, é possível observar no intervalo $[1, 6]$ uma descida acentuada no tempo de pesquisa dos documentos indexados, demonstrando assim a eficiência de paralelizar a pesquisa com processos auxiliares. Contudo, o tempo de pesquisa começa a aumentar quando são usados mais processos para paralelizar a pesquisa, fenómeno este que pode ser explicado pelo *context switch* de vários processos.

3.2 Política FIFO e desempenho da cache

Para testarmos o desempenho da cache e da política utilizada, utilizamos a mesma sequência de indexações (sendo esta sequência randómica e de tamanho 1000) em diferentes tamanhos de cache para fazer procura dos meta-dados das indexações, através da *flag* `-c`.

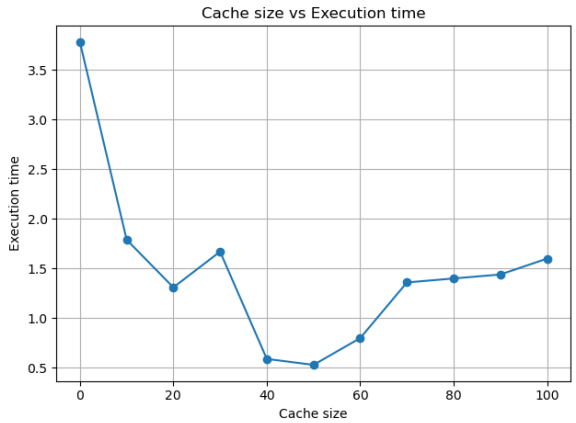

Figura 3.2: Tempo total de procura dos meta-dados *vs* tamanho da cache. Resultado obtido com 1000 documentos indexados

Como é possível observar pela Fig. 3.2, há uma redução no tempo de procura quando aumentamos o tamanho da cache no intervalo $[0, 50]$. A presença do pico quando o tamanho da cache tem valor 30 pode ser explicado pela *Anomalia de Bélády*, que diz que aumentar o tamanho da cache pode não significar maior *hit rate*.

Além disso, é possível notar que quando aumentamos a o tamanho da cache com valores acima de 50, o tempo de procura aumenta de forma sutil. Isto pode devere-se ao facto de que:

- Quanto maior a cache, maior o tempo necessário para adicionar novas chaves à árvore e autobalancear a mesma, quando ocorre um *miss*.
- Sempre que é realizado um *fork()* e quanto maior o tamanho da cache, maior será o tempo para copiar a *Queue*, a árvore e todas as indexações que estão na *heap*.
- Ocorre influência da cache do processador. Uma vez que há a cache L1 (que é a mais rápida) possui entre 2KB a 64KB e que 50 indexações ocupam aproximadamente 25KB, se a cache L1 do processador for de 32KB, os dados começam a ser lidos e escritos na cache L2, onde é mais lento e prejudica o desempenho.

Capítulo 4

Conclusão

O desenvolvimento deste trabalho permitiu-nos aplicar e consolidar diversos conceitos fundamentais da unidade curricular de Sistemas Operativos, nomeadamente a comunicação entre processos, gestão de memória, sincronização, e paralelização.

Além disso, a implementação da paralelização na pesquisa de palavras-chave demonstrou ganhos de desempenho consideráveis, especialmente em conjuntos de dados maiores, como evidenciado na avaliação experimental. Esta abordagem mostrou-se útil para explorar ao máximo os recursos computacionais disponíveis, reduzindo significativamente o tempo de resposta.

Finalmente, este projeto destacou a importância de um bom desenho de comunicação entre processos e de uma gestão criteriosa dos recursos do sistema.