Universidade do Minho
Escola de Engenharia

Sistemas Operativos
Trabalho Prático

Grupo 17

André Campos
a104618

Beatriz Peixoto
a104170

Diogo Neto
a98197

Índice

1. Introdução....2
2. Servidor....2
2.1. Arquitetura....2
2.2. Tratamento de pedidos....3
2.3. Persistência dos documentos....3
2.4. Resposta aos clientes....4
3. Queries....5
3.1. Indexação de documentos (-a)....5
3.2. Consulta de documento (-c)....5
3.3. Remoção de documento (-d)....5
3.4. Pesquisa sobre o conteúdo de um documento (-l)....5
3.5. Lista de documentos sobre um conteúdo (-s)....6
3.5.1. Concorrência....6
3.6. Encerramento do servidor (-f)....6
4. Cliente....7
4.1. Envio de pedidos....7
4.2. Leitura de respostas....7
5. Caches....7
5.1. Cache FIFO....8
5.2. Cache LRU....8
5.3. Cache LFU....9
6. Testes de desempenho....9
6.1. Paralelizar a pesquisa de documentos....9
6.2. Avaliação de Desempenho....9
7. Conclusão....10

1. Introdução

Neste trabalho prático de Sistemas Operativos pretende-se implementar um serviço que permita a indexação e pesquisa sobre documentos de textos guardados localmente num computador. O programa servidor é responsável por registar meta-informação sobre cada documento, permitindo também um conjunto de queries relativamente a esta meta-informação e ao conteúdo dos documentos.

Os utilizadores devem utilizar um programa cliente para interagir com o serviço. Esta interação permitirá que os utilizadores adicionem ou removam a indexação de documentos e que efetuem pesquisas sobre os documentos indexados. O programa cliente executa uma operação por invocação, não sendo um programa interativo.

2. Servidor

O servidor está dividido em três processos principais:

- **Processo Principal :** processo que trata dos pedidos dos clientes e faz a gestão do servidor
- **Processo Ficheiro :** processo que manipula a base de dados e que realiza as queries necessárias para responder aos clientes
- **Processo Respostas :** processo unicamente responsável por enviar respostas aos clientes correspondentes

Estes processos comunicam entre si através de pipes anónimos pois são processos relacionados (pai e filho).

2.1. Arquitetura

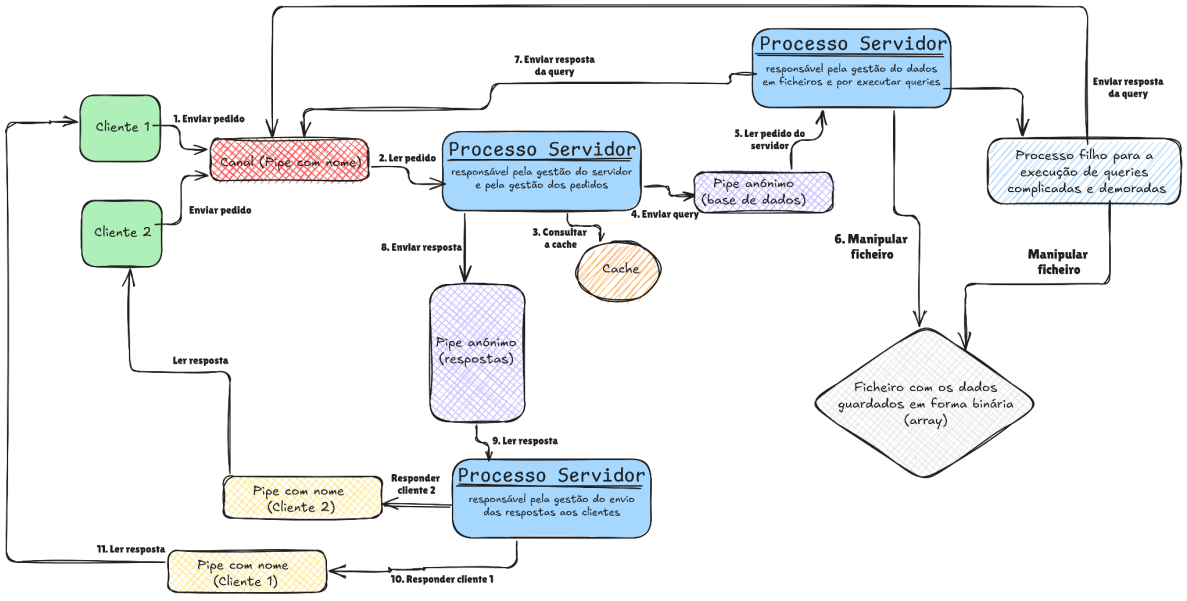

Figura 1. Arquitetura do trabalho-prático

Imagem está presente nos anexos para uma melhor visualização

2.2. Tratamento de pedidos

Os clientes enviam pedidos através do **Canal**, ou seja, do pipe com nome do servidor. O servidor lê os pedidos e pode tomar as seguintes decisões com base de quem mandou o pedido:

Se foi um cliente que enviou:
O servidor procura se existe uma resposta na cache:

- Se existir, responde ao cliente.
- Se não existir, coloca o pedido do cliente em espera e envia um pedido para o processo da base de dados para que ele consiga executar a query pedida.

Se foi o servidor que enviou:
Guarda a resposta na cache e responde a todos os clientes que estiveram à espera daquela resposta

Notas relevantes:

- Como o processo da base de dados tem constantemente o pipe com nome do servidor aberto em modo de escrita, isto faz com que o processo principal nunca bloqueie ou termine caso não haja qualquer cliente a escrever no pipe com nome
- Como a lógica do servidor está dividida em mais que um processo, isto faz com que um cliente não fique bloqueado por causa de outros pedidos de outros clientes *(A forma de como o processo da base de dados realiza as queries faz com que os clientes não fiquem bloqueados por causa de pedidos de outros clientes)*

2.3. Persistência dos documentos

Os dados do servidor ficam guardados num ficheiro chamado **array**. Como o tamanho da meta-informação de cada documento é fixo e, como os índices de cada documento são números inteiros, podemos usar um ficheiro como um array.

Cada documento, para além das meta-informações pedidas, têm campos adicionais como:

- **Valido :** indica se o documento foi eliminado ou não
- **Tempo Lógico :** indica em que tempo aquele documento foi manipulado

No arranque do servidor, o ficheiro é lido, altera-se o tempo lógico de cada documento para zero e são guardados os índices eliminados que podem ser reutilizados para indexar novos documentos.

O processo responsável por este ficheiro lê constantemente os pedidos do pipe anónimo e:

- **Se o pedido for um pedido simples :** o processo executa a query imediatamente pois não vale a pena a criação de um processo filho para resolver a query
- **Se o pedido for um pedido demorado :** o processo cria um ou vários processos filhos, atribui-lhes a query e continua a responder os próximos pedidos enquanto que os processos filhos ficam ocupados com as queries demoradas

Com base nisto, temos algumas observações interessantes:

- Os pedidos dos clientes não são bloqueados por causa de outros pedidos de outros clientes pois há processos diferentes a responder a queries de forma concorrente
- Por causa de termos pesquisa concorrente, temos problemas em relação a documentos apagados e documentos novos enquanto um processo está a fazer uma procura linear no ficheiro, pois esse processo pode responder à query de forma incorreta.
- O uso de tempo lógico ajuda o processo a entender quais documentos tem acesso ou não, ou seja, se um documento foi apagado depois da criação do processo, o processo lê o conteúdo do documento pois, apesar de dizer que o documento foi apagado, esse documento foi apagado depois daquele pedido.
- Como os índices reutilizados são apenas calculados no arranque do servidor e não quando qualquer documento é apagado, isto garante que não há nenhum índice que pode ser apagado e indexado durante toda a execução do programa, garantindo que os processos-filhos respondam a queries corretamente.

2.4. Resposta aos clientes

Este módulo é responsável por tratar das respostas dos clientes e enviá-las para os mesmos através de pipes com nome.

Conversão da Cache para Respostas

A função **convertToResposta** é responsável por converter uma entrada da cache (CacheEntry) para uma estrutura Resposta. Dependendo do tipo de pedido feito pelo cliente(PEDIDO_INSERT, PEDIDO_GET, PEDIDO_DELETE, PEDIDO_LINES, PEDIDO_LIST, PEDIDO_SHUTDOWN), a função preenche os campos da estrutura Resposta com os dados correspondentes.

No caso do PEDIDO_LIST em que é necessário enviar uma lista de índices dos documentos, uma vez que há limite de tamanho das mensagens, os dados são divididos em vários fragmentos de modo a encaixarem-se no tamanho disponível para a estrutura Resposta.

Envio da Resposta ao cliente

O processo principal do servidor cria um pipe anónimo de respostas. Este pipe serve como meio de comunicação entre o processo principal e o processo encarregado de enviar as respostas para os clientes. Quando uma resposta é convertida através da função **convertToResposta**, o processo principal escreve a estrutura Resposta no pipe anónimo de respostas.

Deste modo, o processo responsável por entregar a resposta ao cliente, lê os dados do pipe anónimo de respostas e, para estabelecer a conexão com o cliente pretendido, abre o pipe com nome do cliente usando o PID que se encontra na resposta. Assim, o processo escreve a resposta no pipe com nome do cliente.

Este modelo de comunicação utilizando pipes permite que cada cliente receba apenas a resposta que lhe está destinada e também que múltiplos clientes possam interagir com o servidor de forma independente e eficiente.

3. Queries

Nesta secção vamos explicar como o processo responsável pela base de dados realiza cada query.

3.1. Indexação de documentos (-a)

A estrutura desta tarefa é: *dclient -a "title" "authors" "year" "path"*

Seleciona-se o índice onde vai ser colocado o novo documento, que pode ser um índice reutilizado ou um índice que adiciona um documento novo no fim do array do ficheiro. Ao adicionar o documento, indicamos que o documento é válido e indicamos qual o tempo lógico onde o documento foi criado.

3.2. Consulta de documento (-c)

A estrutura desta tarefa é: *dclient -c "key"*

Calcula-se o offset de onde iremos ler o documento *(index * tamanho de cada entrada)* e lemos o que está no ficheiro. Se foram lidos dados e o documento está válido, lemos as meta-informações do mesmo. Caso contrário, dizemos que o documento não existe

3.3. Remoção de documento (-d)

A estrutura desta tarefa é: *dclient -d "key"*

Calcula-se o offset de onde iremos ler o documento *(index * tamanho de cada entrada)* e lemos o que está no ficheiro. Se foram lidos dados, lemos o documento e altera-se o campo de “válido” para indicar que este documento foi eliminado, indicando também quando este documento foi eliminado.

3.4. Pesquisa sobre o conteúdo de um documento (-l)

A estrutura desta tarefa é: *dclient -l "key" "keyword".*

Nesta tarefa foi desenvolvida a função **tarefa_numero_linhas,** que é responsável por contar o número de linhas de um dado ficheiro associado a uma *key*, que contenham uma dada *keyword*. Para desenvolver a tarefa foram utilizados os comandos **grep** e **wc**. O

comando **grep** procura uma dada *keyword* dentro de um ficheiro e o **wc** conta o número de linhas no caso desta tarefa.

Foi criado um pipe anónimo p1 para fazer a comunicação entre o comando **grep** e o comando **wc**. É feito **fork()** para criar um processo filho que irá executar o comando **grep**. A função **dup2** é utilizada para redirecionar a escrita do comando **grep** para o pipe p1 de modo a permitir que o resultado do **grep** seja enviado para o pipe p1 e não para o terminal. É utilizado o comando **exec** para executar o comando grep.

O processo pai faz novamente um **fork()** e cria um processo filho. Este processo cria um pipe anónimo p2 para permitir que o resultado da contagem de linhas pelo comando **wc** seja recebido pelo processo pai. Deste modo, este processo filho lê do pipe p1 o output do comando **grep**. Através da chamada **dup2**, a escrita do comando **wc** é redirecionada para o pipe p2. Tal como já referido, o comando **exec** permite executar o comando **wc**.

Assim, o processo pai recebe o resultado do comando **wc** através da extremidade de leitura do pipe p2 e é devolvido o resultado desta tarefa.

3.5. Lista de documentos sobre um conteúdo (-s)

A estrutura desta tarefa é: *dclient -s "keyword".*

Nesta tarefa foi desenvolvida a função **tarefa_lista_documentos,** que é responsável por devolver uma lista de índices de documentos que contêm uma dada *keyword.* Para realizar esta tarefa, foram percorridos todos os documentos e, para cada um deles, através da função **tarefa_numero_linhas** determinou-se o número de linhas que continham a *keyword***.** Caso o número de linhas devolvido fosse positivo, então o índice desse documento é adicionado à lista.

No final é devolvida a lista de índices dos documentos com a *keyword* indicada.

3.5.1. Concorrência

A estrutura desta tarefa é: *dclient -s "keyword" "nr_processes"*

Esta é a versão da lista de documentos sobre um conteúdo (-s) que contém concorrência. Nesta versão, o trabalho de pesquisa pelos documentos é dividido por um número de processos. Esse número de processos é definido no parâmetro *nr_processes,* permitindo assim distribuir a carga de trabalho de forma paralela. Deste modo, cada processo filho pesquisa sobre o documento entre um intervalo *start* e *end* definido.

3.6. Encerramento do servidor (-f)

A estrutura desta tarefa é: *dclient -f*

Após o servidor receber este pedido, executa os seguintes passos:

1. Envia um pedido para o processo da base de dados para que ele pare de responder aos pedidos,
2. Após enviar o pedido à base de dados, qualquer pedido dos clientes são respondidos indicando que o servidor está a encerrar,

3. Ao receber a confirmação de encerramento das atividades do processo da base de dados, a receção de pedidos através do pipe com nome termina e fecha os descritores dos pipes anónimos,
4. O processo de respostas, após ver que não há escritores do pipe anónimo, termina as suas atividades.
5. O processo principal fica à espera que os processos filhos terminem e finalmente termina.

4. Cliente

Na execução do cliente, indicamos, nos argumentos da sua invocação, qual a query que gostaríamos que fosse realizada pelo servidor e ficamos à espera que o mesmo nos indique a resolução da qual query.

4.1. Envio de pedidos

O programa cliente analisa se a query está válida, ou seja, se a query pedida é válida, se tem os argumentos certos e se os argumentos estão válidos, como por exemplo, verificando se um índice é um número válido ou se o tamanho de um campo não excede o tamanho máximo permitido.

Após essa verificação, é criado um pedido, que será enviado através do pipe com nome do servidor, contendo toda a informação relevante para a execução da query.

No pedido que o cliente envia ao servidor, indica também qual o seu PID, para que o servidor consiga saber em qual pipe com nome deve escrever a resposta.

4.2. Leitura de respostas

Após enviar o pedido ao servidor, o cliente é responsável por ler e processar a resposta ao seu pedido através de um pipe com nome.

Desta forma, o cliente cria o seu pipe com nome identificado pelo seu PID, o que garante unicidade. É através deste pipe que lê a resposta ao seu pedido. Uma vez que existem pedidos, nomeadamente o PEDIDO_LIST , que podem vir fragmentados em pacotes, a leitura ocorre de forma iterativa enquanto houver pacotes para ler. Desta forma, cada estrutura Resposta lida, é processada e formatada de acordo com o tipo de pedido (PEDIDO_INSERT, PEDIDO_GET, PEDIDO_DELETE, PEDIDO_LINES, PEDIDO_LIST, PEDIDO_SHUTDOWN).

Assim, a resposta correspondente é apresentada no terminal ao cliente.

5. Caches

Para a componente de caching deste projeto, foram desenvolvidas 3 caches, cada uma com uma política diferente: a cache com a política *First In First Out* (FIFO), a cache *Least Recently Used* (LRU) e a cache *Least Frequently Used* (LFU). Para além destas caches, o servidor também tem a opção de não utilizar cache.

As entradas que entram na cache são entradas de **consulta**, **deleção** e **número de linhas de uma keyword**. Manter as entradas com a lista de índices daria muito

trabalho de manter coerente na cache com todos os pedidos de adição e remoção de documentos.

Em relação a pedidos de indexação de documentos, as entradas apenas são usadas uma vez, pois não é possível 2 documentos serem indexados no mesmo índice. Ao manter em cache, estaríamos a perder tempo a procurar e colocar uma entrada que só será usada uma única vez durante toda a execução do servidor.

5.1. Cache FIFO

A cache FIFO implementa a política de substituição simples First In First Out, ou seja, os elementos são removidos da cache pela ordem de inserção. Esta política assume que a primeira entrada a ser adicionada será a primeira entrada a ser removida, quando a cache atinge um limite de capacidade.

A implementação desta cache baseia-se na biblioteca *GLib* e é utilizada uma *GQueue* para armazenar as entradas (*CacheEntry*) por ordem de inserção.

Quando uma nova entrada deve ser inserida na cache e esta já atingiu o seu limite, a entrada mais antiga, presente na cabeça da queue, é removida.

A política FIFO pode não ser a mais eficiente em termos de desempenho ou reutilização de dados, uma vez que remove a entrada mais antiga independentemente do número de acessos ou frequência de uso, podendo descartar elementos ainda úteis. No entanto, a sua simplicidade e previsibilidade tornam-na uma boa base de comparação para outras políticas de cache mais sofisticadas.

Assim, apesar de ser uma política simples, a inclusão da cache FIFO é pertinente para avaliar o impacto de diferentes políticas de cache neste projeto.

5.2. Cache LRU

A cache temporal implementa uma política de substituição do tipo *Least Recently Used (LRU)*, cuja lógica consiste em manter os elementos recentemente utilizados na memória, descartando os mais antigos quando o espaço limite é atingido, esta política é apropriada em contextos onde o padrão de acesso a dados segue o princípio da localidade temporal.

A implementação baseia-se na biblioteca *GLib* e utiliza uma estrutura de fila (*GQueue*) para armazenar as entradas de cache de forma sequencial, embora reconheçamos que existem outras técnicas que permitem uma pesquisa mais eficiente.

Cada entrada armazenada na cache é encapsulada numa estrutura auxiliar designada por *CacheTimeEntry*, que contém um ponteiro para a entrada de cache original (*CacheEntry*), bem como um campo de *timestamp* que indica o momento em que a entrada foi inserida ou acedida pela última vez, este *timestamp* permite manter um registo temporal simples e sempre atualizado.

Quando uma nova entrada é inserida e a cache já atingiu o seu limite, a entrada com o *timestamp* mais antigo é removida, garantindo que permanecem as mais recentemente utilizadas e, a atualização deste *timestamp* ocorre sempre que uma entrada é acedida, permitindo manter a ordenação implícita baseada no recente uso.

Apesar de recorrer a uma pesquisa linear para identificar a entrada mais antiga, esta solução mostrou-se suficiente para os requisitos do projeto.

5.3. Cache LFU

A cache de frequência implementa uma política de substituição do tipo *Least Frequently Used (LFU)*, cuja lógica consiste em manter os elementos mais utilizados na memória, descartando os menos utilizados quando o espaço limite é atingido, esta política é apropriada em contextos onde o padrão de acesso a dados segue o princípio da localidade espacial.

Cada entrada armazenada na cache é encapsulada numa estrutura auxiliar, que contém um pointer para a entrada de cache original (*CacheEntry*), bem como um campo de frequência que indica quantas vezes a entrada foi acedida.

Quando uma nova entrada é inserida e a cache já atingiu o seu limite, a entrada com a menor frequência é removida, garantindo que permanecem as mais utilizadas e, a atualização desta frequência ocorre sempre que uma entrada é acedida.

6. Testes de desempenho

Nesta secção vamos realizar testes ao trabalho prático para analisar e avaliar o desempenho do mesmo.

6.1. Paralelizar a pesquisa de documentos

Query: ./dclient -s “SO”

Número de processos	1	2	5	10	100
Tempo (segundos)	3,115	2,695	0,971	0,591	0,647

Apesar de aumentarmos a quantidade de processos, chega a um ponto onde o tempo para a criação de processos prejudica o tempo da resposta.

6.2. Avaliação de Desempenho

Nesta secção vamos analisar o desempenho de uma script com 300 comandos *(test.sh)* com diferentes caches e com diferentes quantidades de entradas

Sem cache

Número de entradas	*Não tem relevância*
Tempo (segundos)	82,672

Cache FIFO

Número de entradas	5	10	50	100
Tempo (segundos)	71,327	65,949	53,858	53,227

Cache LRU (tempo)

Número de entradas	5	10	50	100
Tempo (segundos)	53,739	53,323	53,118	53,527

Cache LFU (frequência)

Número de entradas	5	10	50	100
Tempo (segundos)	53,214	53,494	53,954	53,271

Percebemos que o uso de caches resulta num menor tempo de execução das 300 tarefas.
As caches LRU e LFU tiveram os melhores tempos e com tempos muito parecidos.
A cache FIFO tem melhor tempo que sem cache, e percebemos que quantas mais entradas, melhor é o tempo.
O servidor sem cache é o que tem pior tempo.

Porém, é importante ressaltar que esta script acessa os mesmos documentos *(documentos cujos índices estão entre 1 e 20)*. Como há bastantes cache hits, o uso de caches compensa.
Se os documentos fossem todos diferentes, a melhor estratégia seria não usar cache, pois a maioria dos pedidos seriam cache misses e ao usar caches, o tempo iria piorar por causa da cache penalty.

7. Conclusão

Através deste trabalho prático conseguimos utilizar todas as primitivas do Sistema Operativos, perceber como as system calls funcionam, concluir que aceder ao disco é mais demorado que aceder à memória principal, utilizar processos e verificar como o uso de cache é benéfico.

Acreditamos que concluímos todos os requisitos do projeto e ficamos satisfeitos com o resultado do mesmo.