Trabalho Prático de Sistemas Operativos

Grupo:

- Diogo Ribeiro A106906
- Carolina Martins A107285
- Filipa Gonçalves A107329

Resumo:

Foi proposto ao nosso grupo o desenvolvimento de um projeto que visa implementar um serviço para indexação e pesquisa de documentos de texto armazenados localmente. O servidor regista meta-informações dos documentos (como título, autor, ano e caminho). O cliente interage com o servidor para adicionar, remover, pesquisar documentos ou encontrar informações detalhadas sobre os ficheiros indexados executando uma operação por vez.

O sistema enfrenta desafios como a necessidade de uma comunicação eficiente entre cliente e servidor, o gerenciamento rápido de grandes volumes de dados, o cuidado com o impedimento do servidor ao receber vários pedidos de múltiplos clientes assim como a persistência dos documentos ao ser reiniciado e a gestão de uma cache para um mais rápido acesso aos documentos.

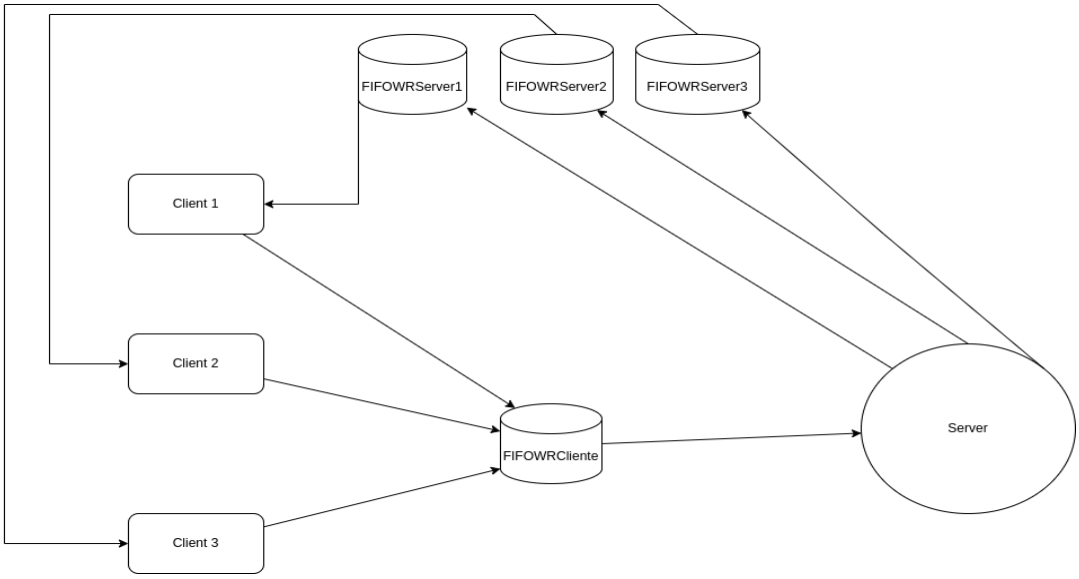
Tudo isso apenas usando system calls (fork, exec, dups, write, read, open, close).

Universidade do Minho
Instituto de Educação

Arquitetura

Este programa segue um modelo Client/Server, onde vários clientes podem enviar requisições para um servidor, ou seja, vários programas "clientes" estão a ser executados, enviando diferentes tipos de requisições (indexação, deleção, consulta, etc.) de ficheiros, enquanto um programa "servidor" lê essas requisições, processa-as e envia as respostas de volta para o cliente específico.

A comunicação entre o cliente e o servidor é realizada através de FIFO's, também conhecidas como named pipes.



Cliente:

O programa do cliente é considerado o mais simples entre os dois (Server, Cliente) pois apenas implica receber os argumentos, abrir o FIFO (*writeClientFIFO*) e escrever os mesmos numa estrutura conhecida pelo server.

Após a escrita, entra num estado de espera pela resposta do server que ao ser recebida depois será escrita no stdout.

Server:

No início do projeto o programa Server começava por tentar ler as informações enviadas pelo cliente.

À medida que o grupo implementou a cache, a persistência e um método para remover os “processos zombies” tornou-se importante fazer algumas preparações antes de começar a tentar ler as informações dos clientes.

Server Inicialização:

Começamos por alocar memória para a cache, de seguida enchemos a mesma com o máximo de índices presentes na persistência (melhor explicado mais à frente) tendo em atenção o máximo de entradas a que a cache tem acesso.

Depois do servidor estar inicializado abrimos um loop que vai começar a ler as requisições dos clientes.
Começa por ler os dados que possam estar no writeClientFIFO e de seguida tenta ler se os dados vêem de um cliente ou de um processo filho.

Após ter lido as informações de algum cliente o servidor verifica o tipo da operação podendo a mesma ser síncrona ou assíncrona.
Operações síncronas são as de adição (-a) que permite indexar um novo ficheiro com os seus metadados e a de deleção (-d) permite eliminar um índice com os seus metadados e ficheiros.
As operações assíncronas são as que permitem algum tipo de consulta.
As de consulta do índice (-c) devolvem ao cliente as informações do índice com a key enviada pelo cliente. A de procura de uma palavra num certo índice (-l) e a de consulta de uma palavra em todos os ficheiros indexados (-s)

Após o server saber se a operação é síncrona ou assíncrona caso venha do cliente faz uma de duas coisas:

Caso seja síncrona, ele limita-se a atualizar a cache e o disco (explicado melhor mais à frente) e manda as informações necessárias para o cliente por um FIFO com o pid do cliente que mandou o pedido (ex: writeServerFIFO7850, writeServerFIFO6700).

Caso seja assíncrona o servidor cria um filho (usando fork()), encontra os dados na cache ou no disco e depois consoante a operação processa (-s , -l), ou não (-c) os dados necessários e escreve no FIFO do cliente (writeServerFIFO**PID_DO_CLIENTE**) as informações necessarias. De seguida manda os dados ao cliente e ao contrário das operações síncronas, têm ainda de fazer duas tarefas antes de finalizar, sendo essas, escrever o seu pid no FIFO (writeClientFIFO) e caso tenha encontrado o índice que andava à procura escrever o mesmo também no mesmo FIFO.

Explicação da comunicação entre pai e filho:

Uma operação assincrona consiste na execução da mesma enquanto outras estão a ser executadas “Ao mesmo tempo”.
E para o fazer é necessário que o programa server (Pai) crie um filho (com fork()) que execute a operação, enquanto o (Pai) continua a percorrer o loop principal a desempenhar outras tarefas. Permitindo assim o programa server ser Multi-Client, ou seja, ser capaz de aceitar pedidos de vários clientes ao mesmo tempo.
Contudo há dois problemas associados em fazer fork():

- Quando o processo pai faz um fork(), é criada uma cópia exata do seu espaço de memória, incluindo a cache. Isto significa que o processo filho recebe uma réplica completa do estado do pai no momento do fork(). Consequentemente, qualquer alteração feita na cache pelo filho não afeta a cache do pai, e vice-versa, pois agora são espaços de memória independentes.

- Segundo, um processo filho só é completamente removido do sistema quando ele termina (através de um exit()) e o pai realiza a operação de espera (wait() ou waitpid()) para recolher o seu estado de término. Caso contrário, o processo filho permanece como um zombie no sistema, ocupando recursos até ser devidamente tratado.

Para resolver qualquer um dos dois problemas, era necessário implementar uma forma de comunicação entre pai e filho. O meu grupo decidiu que a melhor solução seria reutilizar o FIFO (writeClientFIFO).

Ao ter uma mensagem no fifo que tivesse o campo zombiePid a algo que não -1 o server saberia que aquela mensagem era de um processo filho.
E apartir daí ele dá wait desse mesmo pid e caso a mensagem também contenha informação sobre um indice o server saberá que têm que atualizar a cache com o mesmo.

Server Encerramento:

O cliente têm ainda a opção de encerrar o server com a operação (-f) aonde o server irá fechar todos os descriptores ainda não fechados e libertar a memória alocada para a cache.

Cache:

O meu grupo decidiu implementar uma cache usando a política LRU (Least Recently Used) devido à sua simplicidade e leveza na gestão. A principal vantagem da LRU é a sua eficiência em manter os itens mais recentemente acessados na cache, removendo os menos utilizados. A implementação pode ser otimizada utilizando uma combinação de uma tabela hash e uma lista duplamente ligada. A tabela hash permite o acesso rápido aos itens na cache, enquanto a lista duplamente ligada facilita a operação de mover um item para a cabeça da lista sempre que é acessado, indicando que é o mais recentemente utilizado. Isso torna as operações de adição e remoção muito rápidas, com complexidade constante O(1), o que é crucial para a eficiência do sistema.

Inicialmente foi considerada a estratégia do relógio (Clock) como forma de aproximar o comportamento da LRU com menor custo computacional. No entanto, a sua implementação exigiria uma estrutura circular e uma marcação de bits adicionais por entrada, o que complicaria a gestão da cache no nosso contexto. Para manter a simplicidade e garantir acessos e atualizações em tempo constante, decidimos em vez disso utilizar uma lista duplamente ligada como uma espécie de priority queue — onde os elementos mais recentemente usados são movidos para o início da lista e os menos usados vão sendo empurrados para o fim. Esta abordagem é mais intuitiva, fácil de manter e mais adequada à escala e complexidade do nosso projeto.

Contudo, a LRU apresenta a desvantagem de, em caches pequenas, remover itens que foram acessados com frequência, mas que, por alguma razão, não são mais necessários a curto prazo. Como a política LRU remove os itens menos recentemente acessados, mesmo que um item tenha sido muito utilizado no passado, basta que o cliente aceda a outros itens para que o item originalmente muito acessado seja removido da cache. Isso pode resultar em uma cache subótima, onde itens que seriam úteis a longo prazo acabam sendo descartados prematuramente. Apesar dessa limitação, a LRU foi considerada adequada para o projeto, pois sua simplicidade e eficácia a tornam uma solução prática para a maioria dos casos de uso.

Além da LRU, também consideramos a implementação da política LFU (Least Frequently Used), que é baseada na ideia de remover os itens menos frequentemente acessados. A vantagem do LFU é que ele preserva itens que são frequentemente utilizados, mesmo que não sejam os mais recentes. Porém, uma desvantagem significativa do LFU é que itens que acumulam alta frequência de acesso, mas que depois deixam de ser acessados, não são removidos da cache, o que pode levar ao acúmulo de dados obsoletos e reduzir a eficiência. Além disso, o LFU exige uma estrutura de dados mais complexa para acompanhar a frequência de acesso, o que acaba resultando em um maior uso de memória e em um processamento mais lento, impactando negativamente o desempenho.

Por fim, também considereramos uma solução híbrida, o LRFU (Least Recently/Frequently Used), que combina as vantagens do LRU e do LFU, utilizando um processo de decaimento para ajustar a frequência dos itens ao longo do tempo. Embora essa abordagem ofereça uma maior flexibilidade e potencial de eficiência, ela exige um nível de complexidade maior tanto na

implementação quanto na manutenção da estrutura de dados. O LRFU requer mais recursos computacionais para manter os registros de frequência e os mecanismos de decaimento, o que aumenta a carga sobre o sistema e pode resultar em uma desaceleração, especialmente quando a cache cresce em tamanho.

Dado o equilíbrio entre simplicidade, eficácia e consumo de recursos, a implementação da LRU foi a escolha mais adequada para o projeto, proporcionando uma gestão eficiente da cache sem comprometer o desempenho geral.

Persistencia:

O meu grupo decidiu implementar a persistência de maneira otimizada ao ter como chave dos ficheiros indexados, ordem de chegada. Essa abordagem permite uma leitura eficiente e direta do disco. Como os itens são indexados pela ordem de chegada, cada item possui uma chave que corresponde à sua posição relativa no arquivo binário. Para recuperar um item, podemos calcular diretamente a posição utilizando a função lseek, multiplicando o tamanho da estrutura do índice pela chave, que é a ordem de chegada do item. Isso permite que a busca pelo índice seja realizada em tempo constante O(1), sem a necessidade de percorrer o arquivo inteiro.

Essa estratégia elimina a necessidade de múltiplos reads para localizar um item específico, o que seria muito lento, especialmente em arquivos grandes. Com a abordagem de lseek, o acesso direto ao disco torna-se muito mais eficiente, minimizando o número de acessos ao disco e melhorando significativamente o desempenho.

Em relação à deleção de itens no disco, a solução adotada foi substituir o índice do item a ser excluído por um DeletedIndex. Este é basicamente um índice marcado com a ordem de chegada ajustada para -1, indicando que o item foi eliminado. Embora essa técnica tenha sido uma solução prática dada a limitação de tempo, sabemos que não é a ideal, pois o DeletedIndex ainda ocupa espaço no disco, o que leva a uma utilização ineficiente do armazenamento. Embora esse método resolva o problema de maneira rápida, ele deixa registros que não são mais úteis, o que poderia ser melhorado com uma abordagem que realmente liberasse o espaço de forma mais eficaz. No entanto, a implementação atual foi suficiente para os requisitos do projeto, dado o tempo disponível.

Testes executados:

Para avaliar o ganho de desempenho na pesquisa paralela de documentos, foi desenvolvido um primeiro script que mede os tempos de execução do client com a opção -s, utilizando diferentes níveis de paralelismo. O número de processos a serem executados em paralelo é aumentado gradualmente, e os tempos de execução (real, user e sys) são registados num ficheiro CSV (tempos.csv).

Adicionalmente, foi criado um segundo script que simula múltiplos pedidos aleatórios de consulta ao servidor. Este teste permite avaliar o desempenho da cache em comparação com a persistência em disco. Ele analisa como a cache lida com a frequência de acesso aos dados e como o desempenho é impactado pela necessidade de ler informações diretamente do disco, em vez de utilizar a cache, e vice-versa. Isso proporciona uma visão clara sobre o impacto da política de caching implementada no sistema.

Nos testes de desempenho da pesquisa paralela de documentos, observa-se no primeiro gráfico que, sem paralelização, o tempo de execução ultrapassa os 2 segundos. Com paralelização, os tempos mantêm-se sempre abaixo de 1 segundo. Nota-se ainda que o tempo de execução diminui com o aumento do número de threads, até atingir o limite de threads do computador, a partir do qual estabiliza.

Teste com o primeiro elemento não tendo pesquisa paralelizada:

Anexo1

Teste onde todos os elementos foram paralelizados:

Anexo2

De seguida, testamos a cache utilizando o segundo script, que realiza um número de consultas ao sistema à nossa escolha. Para isso, inicializamos a cache com diferentes tamanhos, assim como com a cache desativada, utilizando apenas a persistência, para analisar as diferenças de desempenho. Essas diferenças só puderam ser analisadas corretamente com um número elevado de consultas, devido à grande eficiência da implementação da persistência.
Executamos os testes com um total de 1645 ficheiros (número de ficheiros inseridos pelo script fornecido pelos professores). O grupo decidiu realizar testes com 10 000 consultas (primeiro gráfico) e 12 000 consultas (segundo gráfico).

Relativamente aos dados obtidos, realizámos testes nas seguintes configurações:

- Sem cache;
- Com cache de tamanho 400;

- Com cache de tamanho 800;
- Com cache de tamanho 1200;
- Com cache de tamanho 2000 (neste caso, todos os ficheiros cabem em cache desde o início).

Observou-se que os resultados para os cenários sem cache e com cache de tamanho 400 foram bastante semelhantes, tanto no teste de 10 000 como no de 12 000 consultas. Esta semelhança deve-se ao facto de que, com apenas 400 entradas para 1645 ficheiros disponíveis, a maior parte das consultas à cache acabam por falhar (cache misses). Consequentemente, o sistema precisa de aceder ao disco na maioria das vezes, criando overhead adicional: primeiro há o custo de verificar a cache sem sucesso, seguido do acesso à persistência. Assim, o benefício da cache é praticamente anulado neste caso.

No entanto, para configurações com cache de 800, 1200 e especialmente 2000 entradas, notaram-se diferenças significativas no desempenho. À medida que a cache cobre uma proporção maior do total de ficheiros, a taxa de cache hits aumenta, reduzindo substancialmente o número de acessos ao disco e, portanto, melhorando o tempo de resposta.

Em números concretos:

- No teste com **10 000** consultas, registou-se um aumento de desempenho de aproximadamente **12%** entre o cenário sem cache e o cenário com cache de tamanho 2000.
- No teste com **12 000** consultas, o ganho subiu para cerca de **15%**, reforçando que, quanto maior o número de consultas realizadas, maior a justificação e o impacto positivo do uso da cache.

Estes resultados demonstram claramente que a cache se torna progressivamente mais relevante à medida que a carga de trabalho aumenta, validando assim a decisão de implementar mecanismos de caching eficientes no sistema.

Teste com 10000 consultas:

Anexo3

Teste com 12000 consultas:

Anexo4

Conclusao:

O projeto desenvolvido permitiu a criação de um servidor multi-cliente eficiente, suportando operações assíncronas através da utilização de fork() e comunicação por FIFOs, garantindo que múltiplos pedidos pudessem ser tratados simultaneamente sem bloqueios.
Implementou-se uma cache baseada na política LRU, equilibrando simplicidade e desempenho, o que proporcionou ganhos claros em cenários de elevado número de consultas, sem sacrificar a leveza do sistema. A persistência dos dados foi também otimizada, usando o ordenamento natural dos índices para permitir acessos diretos em tempo constante (O(1)), minimizando o custo de interações com o disco.
Apesar de algumas limitações, como a gestão de remoções na persistência que poderia ser mais refinada, o sistema alcançou um desempenho robusto e escalável, comprovado através dos testes experimentais.
O projeto demonstrou a importância da eficiência em sistemas de I/O intensivo e destacou o valor de escolhas arquiteturais cuidadosas na otimização de recursos, garantindo assim uma solução funcional, rápida e preparada para cenários de carga variada.