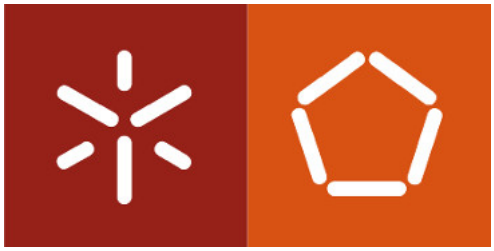

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Operativos

Ano Letivo de 2021/2022

SDStore: Armazenamento Eficiente e Seguro de Ficheiros

Grupo 103

A96326 Bernard Ambrósio Georges
A96434 Francisca Quintas Monteiro de Barros
A97642 José Rafael Cruz Ferreira

31 de maio de 2022

1 Introdução

No âmbito da cadeira de Sistemas Operativas, foi-nos proposto implementar um serviço que permitisse aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço de disco. Para tal, o serviço disponibilizará um conjunto de transformações que permitem comprimir e cifrar os ficheiros a serem armazenados. Existem 7 transformações disponíveis:

• **bcompress / bdecompress.** Comprime / descomprime dados com o formato bzip.

• **gcompress / gdecompress.** Comprime / descomprime dados com o formato gzip.

• **encrypt / decrypt.** Cifra / decifra dados.

• **nop.** Copia dados sem realizar qualquer transformação.

O serviço permite a submissão de pedidos para processar e armazenar novos ficheiros bem como recuperar o conteúdo original de ficheiros guardados previamente. É possível ainda consultar as tarefas de processamento de ficheiros a serem efetuadas num dado momento e estatistícas sobre as mesmas.

Para implementar este serviço, foram criados dois programas - *sdstore* e *sdstored*, que fazem o papel de cliente e servidor, respetivamente.

O programa *sdstore* permite enviar pedidos ao servidor através dos seus argumentos. Além disso, vai informando o utilizador do estado dos seus pedidos (no *standard output*).

O programa *sdstored* vai respondendo aos pedidos dos clientes.

Em primeiro lugar, é necessário inicializar o servidor. Este vai receber dois argumentos: o primeiro corresponde ao número máximo de operações concorrentes (de cada tipo) que este consegue executar e o segundo corresponde ao lugar onde se encontram os executáveis das transformações a fazer.

O servidor processa esta informação e está pronto a receber pedidos de clientes.

O servidor termina a sua execução se receber o sinal *SIGTERM*, acabando antes os pedidos em processamento ou pendentes, mas rejeitando a submissão de novos pedidos.

2 Estratégias utilizadas

2.1 Inicialização do servidor

Tal como foi referido anteriormente, o servidor recebe dois argumentos. Quando este é iniciado, é guardada a informação dada por esses argumentos.

O primeiro argumento contém o caminho para o ficheiro com a informação sobre o número máximo de instâncias, de uma certa transformação, que podem executar concorrentemente. Esta informação é guardada em memória. Optámos por ter dois arrays de tamanho 7, um guarda o número máximo de instâncias e o outro quantas estão disponíveis. Cada posição do array está associada a uma transformação específica (existe uma função que associa a transformação à sua posição). É considerado que o ficheiro que é passado deve definir valores para todas as transformações e apenas um valor por transformação, caso isto não se verifique, o servidor não é criado.

O segundo argumento contém o caminho onde estão os executáveis e é guardado numa string que contêm mais espaço para, posteriormente, ser acrescentada a transformação pedida para a poder executar.

De seguida, é criado um pipe com nome cujo objetivo será receber os pedidos dos clientes (é portanto aberto para leitura, no servidor). Neste ponto, o servidor fica à espera de pedidos, visto que fica bloqueado no *open* do pipe.

Nota: o pipe é depois também aberto em modo escrita no servidor, para permitir que este não termine a sua execução após um pedido.

2.2 Enviar pedidos para o servidor

O programa *sdstore* recebe nos seus argumentos o pedido a fazer ao servidor. Para enviar o pedido para o servidor é criada um string que contenha toda a informação necessária para o executar.

Quando é pedido o estado do servidor (comando *status*), a string criada vai ter um identificador deste tipo de pedido - o carater 's', no início - , seguido do identificador do pipe utilizado para a comunicação servidor $\rightarrow$ cliente (explicado na próxima secção).

Quando é pedido o processamento de um ficheiro (comando *proc-file*), a string vai ter a prioridade no início (se não for especificada fica 0), seguida do identificador do pipe utilizado para a comunicação servidor $\rightarrow$ cliente, seguida do ficheiro a ser processado, o caminho onde deve ser guardado a nova versão do ficheiro, e finalmente as várias transformações a fazer. Todas estes argumentos ficam separados por espaço.

Nota: no final das strings enviadas com os pedidos está sempre um \n, para ser possível distinguir onde é que o pedido acabou.

2.3 Resposta do servidor

Como há necessidade do servidor enviar informação ao cliente, é preciso haver um pipe que permita esta operação, no entanto, não basta criar um pipe para essa função, visto que vão ser executados pedidos concorrentemente e se só houvesse um pipe para todos os clientes, o servidor não controlaria para que cliente é que estava a mandar a mensagem. Assim, é necessário criar um pipe por cliente. Isto traz um possível problema: o nome do pipe, é necessário arranjar algo que seja único para cada cliente para nomear o pipe. Optámos por escolher o pid do cliente.

É então criado um pipe, no cliente, cujo nome corresponde ao seu pid. O pipe é aberto para leitura e vai escrevendo no *standard output* a informação que é para lá enviada.

Nota: quando o pedido é *proc-file* o pipe é também aberto no modo escrita, visto que o pedido pode ficar pendente e o servidor fecha o descritor que têm aberto em modo escrita, o que acabaria com o ciclo que está a ler. Este descritor é fechado quando o cliente recebe a informação de que o pedido está concluido ou é inválido.

2.4 Comando proc-file

Quando é enviado um pedido de processamento de um ficheiro, a primeira coisa que o servidor faz é o parse do string enviada, retirando o nome do pipe utilizado para a comunicação servidor $\rightarrow$ cliente, o ficheiro a ser processado e o caminho onde deve ser guardado a nova versão do ficheiro. De seguida, verifica-se se é possível executar o pedido (de acordo com as limitações definidas na inicialização do servidor).

Existem então três cenários possíveis: é possível executar o pedido, este fica pendente ou é detetado algum erro no pedido enviado.

Se for detetado um erro, é enviada a mensagem de erro correspondente ao cliente e o pedido não é processado.

Se for possível executar o pedido, este é acrescentado a uma lista (ligada) de pedidos a ser processados e é criado um novo processo-filho que o vai fazer.

Se não for possível, o pedido é acrescentado a um fila de espera (lista ligada de pedidos).

2.5 Prioridades

A prioridade dos pedidos, tal como foi referido anteriormente, aparece no início do pedido. Esta é apenas utilizada quando o pedido fica pendente: quando o pedido é acrescentado à fila de espera é colocado de acordo com a sua prioridade, ou seja, a lista que representa os pedidos pendentes está ordenada por ordem decrescente de prioridade.

No momento em que se vai verificar se já é possível executar algum dos pedidos, os primeiros a serem verificados são os mais prioritários.

2.6 Número de bytes do ficheiro de entrada e saída

Quando o processamento de um ficheiro está concluido, além de enviar esta informação para o cliente, informa-o também do número de bytes do ficheiro inicial e do ficheiro final. Para tal, é utilizada a chamada ao sistema *lseek*, para o final do ficheiro.

Outra solução seria usar o comando "*wc -c*" para o ficheiro e em vez de enviar o resultado para o standard output, enviar a informação para um pipe anónimo, por exemplo.

2.7 Controlo de número de operações concorrentes

Tal como foi explicado na secção do comando *proc-file*, é feita um verificação antes de executar o pedido. Se se concluir que é possível executar o pedido, é alterado o array que contém informação sobre o número de instâncias disponíveis, de acordo com as transformações necessárias.

Quando o pedido é concluído ele envia uma mensagem para o pipe normalmente usado para os pedidos (pipe da comunincação cliente $\rightarrow$ servidor). Esta mensagem serve para avisar o servidor de que aquele pedido foi concluido. Contém um identificador do tipo de mensagem que está a ser enviada - o carater 'c', no ínicio - , seguido do identificador do pedido (pid do cliente que o enviou). Quando o servidor recebe este tipo de mensagens procura na lista dos pedidos a ser executados o pedido e repõe o número de instâncias disponíveis para cada transformação.

De seguida, percorre a lista dos pedidos pendentes, com o objetivo de ver se já existem pedidos que possam ser executados.

2.8 Comando status

Quando o servidor recebe um pedido *status*, envia para o cliente (através do pipe com nome criado no cliente) informação sobre quais os pedidos que estão pendentes e a ser executados, bem como o estado de utilização das transformações.

2.9 Desligar servidor

O servidor é desligado quando recebe o sinal SIGTERM.

Para tal, foi criada uma variável global que indica se o sinal já foi enviado. Quando o sinal é enviado, esta variável inicialmente com o valor 1, passa a 0 e é enviada uma mensagem para o servidor (através do pipe dos pedidos) a avisar que foi enviado o sinal. Se já não houverem pedidos a executar nem pendentes, o servidor é fechado, se não, ele continua a executar até que as listas de pedidos estejam vazias. Se entretanto receber pedidos estes são rejeitados sendo enviada uma mensagem de erro para o cliente.

3 Conclusão

Durante a realização do trabalho foram usadas diferentes estratégias para guardar e representar os pedidos. Inicialmente, as listas onde eram guardados os pedidos eram simplesmente arrays de strings, onde cada string era um pedido. No entanto, optámos por alterar esta estratégia, para evitar estar sempre a fazer parsing do pedido. Assim, foi criada uma estrutura para representar cada pedido. Esta estrutura contêm informação sobre: o pid do processo do cliente, a prioridade do pedido, o número de vezes que é preciso executar cada operação e as operações a fazer, por ordem.

Optámos por guardar estes pedidos em listas ligadas, visto que não sabemos qual a quantidade de pedidos que será necessário guardar. Era possível criar um array estático de pedidos, no entanto, se este fosse demasiado pequeno podia não aguentar com todos os pedidos e se fosse demasiado grande iria ocupar muito espaço na memória. Outra solução, seria usar um array dinâmico, que seria mais eficiente no que diz respeito à localidade espacial, no entanto, numa lista ligada é mais fácil de inserir e retirar elementos, porque não é preciso fazer shifts.

Em termos gerais, pensamos que conseguimos cumprir todos os objetivos pedidos.