



SDStore: Armazenamento Eficiente e Seguro de Ficheiros

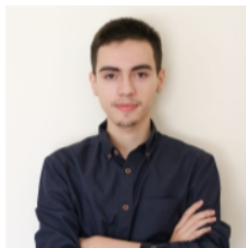
Trabalho Prático - Sistemas Operativos

Grupo 19

Nuno Guilherme Cruz Varela - a96455

Gabriela Santos Ferreira da Cunha - a97393

Miguel de Sousa Braga - a97698



a96455
Guilherme Varela



a97393
Gabriela Cunha



a97698
Miguel Braga

1 Introdução

Este trabalho consiste na implementação de um serviço que permita aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço de disco. Para tal, o serviço disponibiliza funcionalidades de compressão e cifragem dos ficheiros a serem armazenados.

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

2 Comunicação cliente-servidor

Neste trabalho, pretendemos que o servidor acolha pedidos de vários clientes de forma concorrente. Para isso, criamos um pipe com nome (FIFO) no servidor, partilhando-o pelos diferentes clientes. Este FIFO é definido com o nome “REQUEST” e guardado na pasta “tmp”. Existe um outro FIFO presente na comunicação cliente-servidor, o FIFO “INFO”. Através deste canal de comunicação, o servidor é avisado para pedidos de status do servidor, pedidos que chegam ao servidor ou pedidos que terminam de executar.

2.1 A estrutura Request

```
typedef struct request {
    pid_t id;
    char priority;
    char input_file[100];
    char output_file[100];
    char transforms[20];
    int fd;
    int task_number;
} *Request;
```

Figura 1: Estrutura Request.

Cada cliente vai submeter pedidos para serem tratados pelo servidor. Deste modo, decidimos criar uma *struct* que representa este pedido, sendo criada a partir de um *parsing* de argumentos no cliente. Esta estrutura contém um inteiro (ID do processo do cliente) - para que o servidor saiba o FIFO para onde tem de enviar as informações de volta para o cliente -, a prioridade atribuída ao pedido, representada através de um *char* - de modo a minimizar a utilização de memória -, uma string com o ficheiro de *input* sobre o qual vão ser aplicadas as transformações, outra com as respetivas transformações e, ainda, outra com o

ficheiro de *output* onde deve ser guardado o resultado final. Por último, contém ainda um inteiro que representa o descritor do FIFO que é utilizado sempre que o servidor envia informações para o cliente e um inteiro que é o número do pedido no servidor (para efeitos de *status*).

2.2 Criação de um pipe (com nome) único a cada cliente

Como a comunicação cliente-servidor é bidirecional, é necessário que o servidor seja capaz de comunicar com cada um dos clientes.

A existência de um outro FIFO partilhado por todos os clientes não é solução, uma vez que cada cliente desconhece se a mensagem enviada pelo servidor é para si, ou não. Uma opção seria adicionar à mensagem recebida pelo cliente um identificador do cliente que inicialmente enviou pedido para o servidor. No entanto, essa solução seria ainda bastante ineficiente, uma vez que o cliente só consegue saber se a mensagem é a si destinada depois de ler do FIFO, obrigando a que a mensagem seja enviada novamente para o FIFO. O mesmo acontece se for enviado um pedido *status*, o cliente vai ler deste FIFO a *string* que traduz o estado do servidor naquele momento.

De modo a resolver este problema, definimos que cada cliente seria responsável pela criação de um pipe único, caracterizado pelo ID do processo cliente. Toda a informação vinda do servidor seria, assim, enviada de forma inequívoca para o cliente respetivo, tendo o cliente apenas que ler a informação no extremo de leitura do FIFO.

Mais tarde, após o processamento do pedido, o cliente é informado pelo servidor (pelo FIFO único a cada cliente) em três situações: o pedido fica em fila de espera, o pedido entra em execução e o pedido termina a execução, sendo neste último caso reportado o número de bytes lidos e escritos de/para ficheiro.

3 Arquitetura do servidor

O nosso servidor consiste na existência de dois processos concorrentes: o processo-pai (servidor) que recebe vários tipos de pedidos (*status*, pedidos de transformação, pedidos que deixaram de estar em execução, etc.) e o processo-filho que aplica as transformações. A existência destes dois processos permite que sejam recebidos novos pedidos ao mesmo tempo que os pedidos em transformação são executados.

A arquitetura do servidor pode ser sintetizada pelo seguinte esquema:

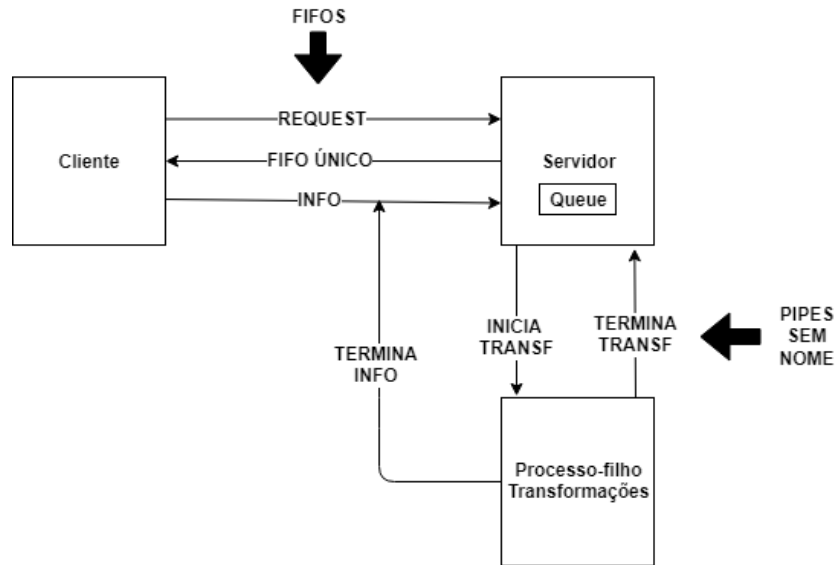


Figura 2: A arquitetura implementada.

3.1 Tipos de pedidos/informações

Existem diversos tipos de pedidos/informações que podem chegar ao servidor, através do pipe "INFO".

O primeiro - e o mais óbvio - é a chegada de pedidos de sequências de transformações do lado dos clientes. Estes pedidos são identificados com o inteiro -1 e sinalizam o servidor para ir verificar um outro pipe, de nome "REQUEST", onde são enviados os pedidos sucessivos que chegam dos clientes.

Um outro pedido que pode chegar dos clientes é um pedido de *status*, indicando que o cliente deseja saber o estado interno do servidor. Como este pedido não apresenta informação adicional, reservamos toda a gama dos inteiros positivos para poder ser indicado simultaneamente qual é o ID do processo do cliente que pediu o estado do servidor. Este inteiro servirá para, mais tarde, conseguirmos ter acesso ao FIFO exclusivo a esse cliente.

Internamente ao servidor, há ainda um outro tipo de pedido que pode ser lido. Neste caso, o identificador -2 indica ao servidor que um pedido deixou de executar as transformações a ele destinadas. Esta informação é importante, pois permite que, caso a fila esteja cheia, novos pedidos possam entrar em transformação, visto que já existem mais recursos livres (transformações).

A existência de um único pipe que recebe vários tipos de “sinais” tornou-se importante, pois permitiu agregar toda a lógica de controlo num único processo (*buffers* de controlo, filas de espera, etc). Inicialmente, pensamos em ter diferentes FIFOs para receber pedidos de *status* e pedidos de transformações. No entanto, isso iria requerer a necessidade de criar ainda mais mecanismos de comunicação entre processos, para além de estar mais sujeito a erros e a situações de *deadlock*, devido ao aumentar da concorrência. Julgamos desnecessária tal complexidade.

3.2 Chegada de um pedido ao servidor

A partir do momento em que um cliente envia para o pipe “INFO” o sinal -1 e para o pipe “REQUEST” o pedido correspondente, o servidor pode ler esse pedido e processá-lo. Duas situações podem ocorrer:

- O pedido pode começar já a executar;
- O pedido fica em fila de espera.

Esta segunda situação ocorre quando os pedidos que estão a ser tratados atingem o número máximo de operações concorrentes dentro do servidor.

Depois de decidir o que fazer com o pedido que chega, o servidor passa para a próxima iteração do ciclo, bloqueando em espera passiva (no *read*) à espera de um novo sinal, seja pedido que entra ou pedido que termina.

3.3 Filtragem de pedidos

No arranque, o servidor é configurado para um número máximo de operações que podem ocorrer em simultâneo de cada tipo. O ficheiro de configuração é lido e convertido para um *array* de inteiros com 7 posições, a indicar o número máximo de transformações concorrentes permitidas para aquele tipo.

Assim sendo, podem chegar ao servidor pedidos que nunca irão chegar a executar. Por exemplo, se o máximo de “nops” permitidos for 2 e um dado pedido contiver 3, este não poderá executar. Estes pedidos são assim filtrados na chegada ao servidor, pois a sua entrada para a fila de espera iria resultar numa estadia permanente desse pedido nessa fila de espera, algo que não desejamos.

3.4 A fila de espera

De modo a cumprir o requisito das prioridades, adicionamos uma fila de espera gerida por nós para os pedidos que não podem entrar imediatamente em execução. Como estrutura de dados escolhemos uma *heap* implementada como um *array*. Esta estrutura de dados garante, para além de fornecer sempre o

elemento com maior prioridade, tempos logarítmicos no número de elementos lá armazenados, tanto para a inserção como para a remoção do elemento máximo (pedido com maior prioridade, neste caso).

3.5 Envio de pedidos para o processo das transformações

De modo a assegurar a comunicação entre os dois processos internos ao servidor, criamos dois pipes anónimos (um em cada sentido). Assim, quando um pedido é mandado executar, o processo-pai (*main*), envia para o pipe “send_request“, criado no arranque do servidor e visível para ambos os processos, o pedido correspondente. Após este pedido ter terminado a execução, o processo-filho, que alberga a execução das transformações, envia para o processo-pai o pedido terminado através do pipe “send_concluded_request“, para que possam ser atualizados os *buffers* de controlo do número de transformações atualmente a executar.

3.6 Processo que trata das transformações

Tal como pedido no enunciado do trabalho prático, o servidor recebe o pedido do cliente e aplica as devidas transformações, de maneira a não criar ficheiros temporários. Desta forma, as transformações são feitas em memória, com o uso de pipelines.

Assim, estamos a criar um processo novo para cada pedido, responsável por executar as transformações. São criados $N - 1$ pipes sem nome - em que N é o número de transformações do pedido -, de onde vão ser lidos e escritos os resultados da execução das transformações, através do redirecionamento de ficheiros. No final, o *output* da última execução é escrito no ficheiro destino passado como parâmetro.

De forma a tornar o código mais modulado e perceptível, optamos por criar um módulo *transformations* que é responsável por interpretar o array das transformações e aplicar o executável respetivo, sendo que estes já tinham sido disponibilizados. Uma vez que o redirecionamento é diferente durante as transformações iniciais, intermédias e finais temos de fazer a distinção entre estas 3 fases.

Os executáveis são chamados com recurso à *system call* “exec“, que substitui o processo atual pelo executável indicado. Como cada um dos executáveis disponibilizados apenas opera sobre o *standard input* e sobre o *standard output*, tivemos que redirecionar, através da *system call* “dup2“, dependendo das circunstâncias, estes descritores de ficheiros para pipes ou para ficheiros abertos anteriormente (*input* e *output*).

3.7 Bytes lidos e escritos

Outra das funcionalidades implementadas foi a indicação ao cliente, por parte do servidor, do nº de *bytes* lidos do ficheiro de *input* e do nº de *bytes* escritos no ficheiro de *output*. Para isto, consideramos, inicialmente, adicionar uma variável que fosse acumulando o nº de *bytes* lidos/escritos por cada operação de *read/write* realizada nos ficheiros correspondentes. Esta operação teria de ser feita no processo que executa as transformações. Decidimos, no entanto, usar um simples “lseek” no ficheiro inicial e no ficheiro final, depois de aplicadas todas as transformações. Os valores obtidos são depois enviados para o cliente pelo pipe com nome (FIFO) já existente.

3.8 Terminação de forma graciosa

Para implementar este requisito avançado no nosso trabalho, recorreremos ao uso de sinais (tal como dito no enunciado), definindo uma função de tratamento do sinal SIGTERM.

Quando recebe este sinal específico, a função altera uma variável global, visível em todo o programa, que indica se o servidor pode ou não receber mais pedidos. Também remove o pipe “REQUEST” que recebe pedidos do cliente, para estes saberem se o servidor já terminou ou não a sua execução.

Contudo, o servidor pode ainda ter pedidos a executar, pelo que o pipe que recebe “infos” não pode ser fechado. A condição para continuarmos a ler nesse FIFO é (*is_closed* == 0 || *num_conc_requests* > 0), isto é, continuamos a ler até que a flag *is_closed* esteja a 1 e o número de processos em processamento seja 0.

4 Sistema de Debug

Neste trabalho, uma das tarefas, ao início, mais difíceis de realizar, foi o *debug*. O simples facto de ser uma aplicação concorrente com uma arquitetura cliente-servidor faz com que seja difícil identificar problemas como esperas indesejadas por um determinado recurso (pipe, FIFO, ficheiro) - *deadlocks*, por exemplo.

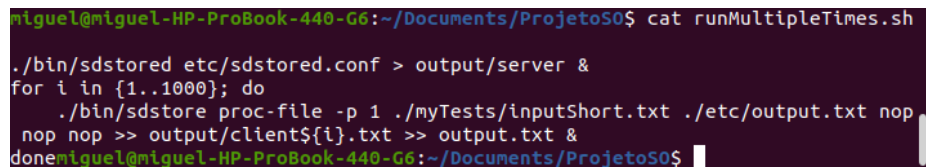
De modo a facilitar o *debug* do programa, criamos então um simples sistema de *debug* com vários níveis (*info*, *warning* e *error*) que pode ser ativado/desativado conforme desejado. Consideramos que este mecanismo contribuiu de forma importante para a identificação da razão para alguns problemas no código e conseqüente resolução.

5 Testes

Um dos aspetos essenciais de qualquer projeto de programação é a efetuação de testes ao código. Num sistema tão sujeito a falhas como é o que apresentamos, o processo de testagem e correção de erros ganha ainda maior importância. Assim, fizemos um conjunto de testes, de modo a apurar quer a correção do programa, quer o seu desempenho.

5.1 Teste 1

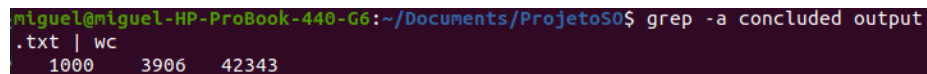
Um dos testes realizados colocou à prova a capacidade do servidor de receber múltiplos pedidos de forma concorrente, dando resposta a todos atendendo à sua ordem de chegada e prioridade correspondentes. Assim, elaboramos o seguinte *script*:



```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoS0$ cat runMultipleTimes.sh
./bin/sdstored etc/sdstored.conf > output/server &
for i in {1..1000}; do
  ./bin/sdstore proc-file -p 1 ./myTests/inputShort.txt ./etc/output.txt nop
  nop nop >> output/client${i}.txt >> output.txt &
done
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoS0$
```

Figura 3: Script de envio concorrente de pedidos para o servidor.

Após a execução do *script* verificamos o número de pedidos que terminaram a execução com o seguinte comando:



```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoS0$ grep -a concluded output
.txt | wc
 1000   3906   42343
```

Figura 4: 1000 pedidos terminaram a execução.

5.2 Teste 2

Outro aspeto a analisar é o desempenho de um único pedido no servidor, comparativamente com um pedido que é executado com recurso aos pipes disponibilizados pela *bash*.

```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoS0/bin/sdstore-transformation
s$ time ./bcompress < ../../myTests/inputLong.txt | ./bdecompress > out.txt

real    0m21,371s
user    0m25,523s
sys     0m0,168s
```

Figura 5: Tempo de execução de um pedido com os pipelines da *bash*.

```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoS0$ time ./bin/sdstore proc-f
ile myTests/inputLong.txt myTests/outputLong.txt bcompress bdecompress
pending
processing
concluded (bytes-input: 105056280, bytes-output: 105056280)

real    0m21,511s
user    0m0,003s
sys     0m0,000s
```

Figura 6: Tempo de execução de um pedido no servidor.

Neste caso, podemos concluir que os valores obtidos estão dentro do esperado, visto que há uma ligeira variação entre os tempos de execução do pedido na *bash* e o tempo de execução do pedido no servidor. Essa diferença é explicada pelo tempo adicional que o pedido gasta dentro dos FIFOs e dos pipes na nossa aplicação, de processo para processo.

```
total 467M
-rw-rw-r-- 1 miguel miguel 101M mai  3 14:10 inputLong.txt
-rw-rw-r-- 1 miguel miguel   3M mai  1 12:29 inputShort.txt
-rw-r--r-- 1 miguel miguel 217M mai  3 19:04 inputVeryLong.txt
-rw-r----- 1 miguel miguel   0M mai  6 12:20 output1.txt
-rw-r----- 1 miguel miguel   0M mai  6 12:20 output2.txt
-rw-r----- 1 miguel miguel 101M mai 26 22:04 outputLong.txt
-rw-r----- 1 miguel miguel   3M mai 26 21:17 outputShort.txt
-rw-r----- 1 miguel miguel   0M mai  5 23:33 output.txt
-rw-r----- 1 miguel miguel  50M mai 26 22:44 outputVeryLong.txt
```

Figura 7: Tamanhos dos ficheiros de teste em Mb.

Realizando um teste semelhante mas com um ficheiro de *input* maior:

```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoSO/bin/sdstore-transformations$ time ./bcompress <
../../myTests/inputVeryLong.txt | ./bdecompress | ./encrypt | ./decrypt | ./nop | ./nop | ./gcompress
s | ./gdecompress | ./bcompress > out.txt
real    0m31,843s
user    1m17,288s
sys     0m1,387s
```

Figura 8: Tempo de execução de um pedido maior com os pipelines da bash.

```
miguel@miguel-HP-ProBook-440-G6:~/Documents/ProjetoSO$ time ./bin/sdstore proc-
file myTests/inputVeryLong.txt myTests/outputVeryLong.txt bcompress bdecompress
encrypt decrypt nop nop gcompress gdecompress bcompress
pending
processing
concluded (bytes-input: 227322162, bytes-output: 51559640)
real    0m31,889s
user    0m0,001s
sys     0m0,003s
```

Figura 9: Tempo de execução de um pedido maior no servidor.

Continuamos a ver uma pequena diferença nos tempos de execução dos 2 programas. Contudo, verificamos que essa diferença não aumentou com o tamanho do *input*, pelo que a diferença deve-se aos tempos de espera e do trânsito do pedido desde que este sai do cliente, até que termine as transformações e comunique os resultados ao cliente. Também testamos se o resultado dos dois ficheiros de *output* coincidiam, algo que pudemos verificar com o comando “diff”.

6 Conclusão

Neste trabalho prático realizado na Unidade Curricular de Sistemas Operativos, tivemos uma primeira abordagem prática aos conceitos de concorrência na execução de pedidos (*tasks*) numa arquitetura cliente-servidor. Também complementamos os conceitos aprendidos quer nas aulas teóricas quer nas aulas práticas, nomeadamente os conceitos de escalonamento, processo, pipeline, ficheiro (abertura e operações básicas), entre muitos outros. Os resultados obtidos foram bastante satisfatórios, pois acreditamos que o nosso programa cumpre todos os requisitos.