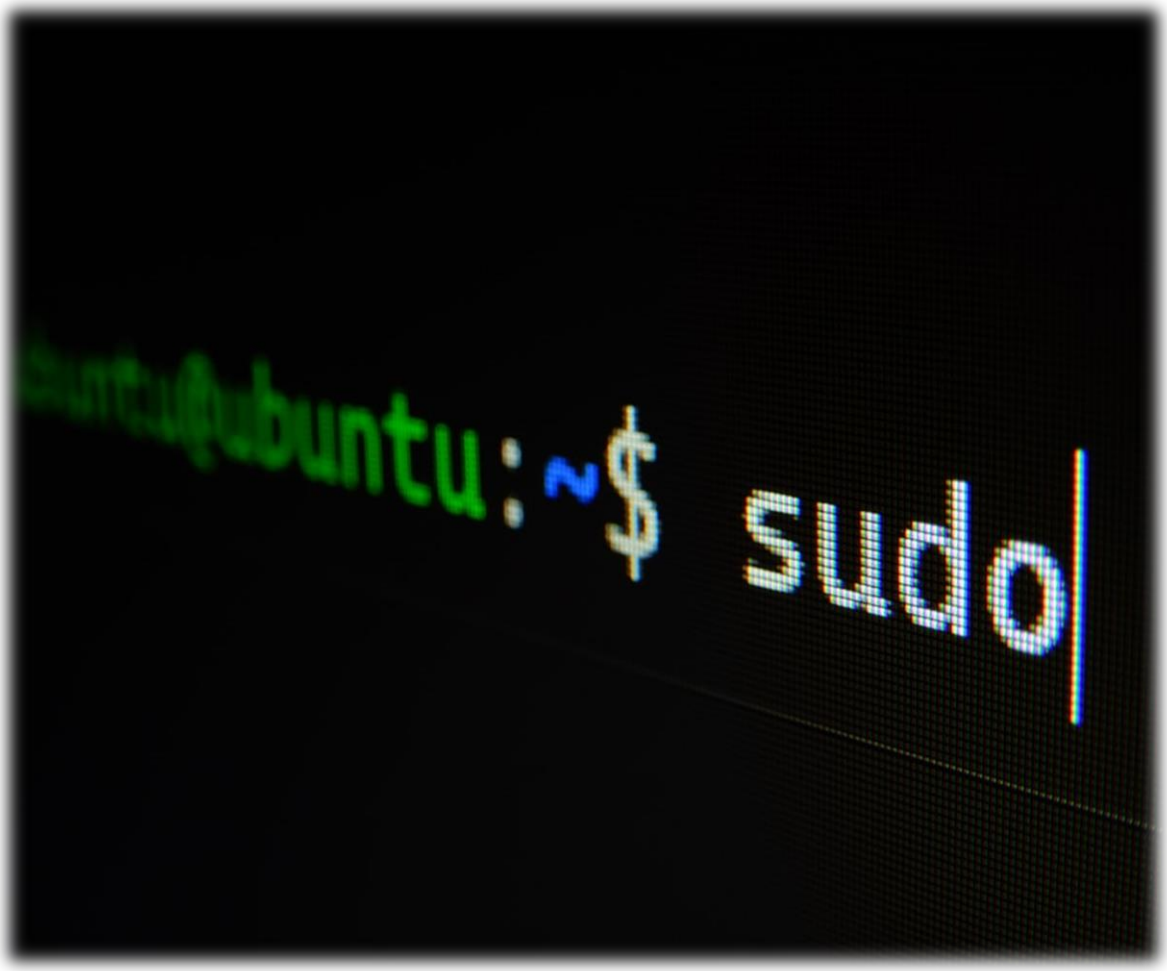
Universidade do Minho
Escola de Engenharia

Universidade do Minho
Sistemas Operativos
29 de maio de 2022

Relatório Trabalho Prático



Grupo 99:

João Paulo Machado Abreu | a91755

Ricardo Cardoso Sousa | a96141

Tiago Luís Pereira Ferreira | a97141

Índice

Introdução

Este projeto foi desenvolvido no âmbito da Unidade Curricular de Sistemas Operativos, tendo como objetivo, tal como refere o enunciado "implementar um serviço que permita aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço de disco". Para tal, é sugerida a implementação de um modelo cliente-servidor, que operem no mesmo sistema. Consequentemente, quer o servidor, quer os vários clientes são, então, processos de uma mesma máquina. No decorrer deste relatório, explicaremos de que forma os clientes comunicam com o servidor, como é efetuada a "resposta" do servidor a cada um dos clientes e, também abordaremos a arquitetura de cada programa.

Comunicação entre cliente e servidor

A comunicação entre clientes e servidor é efetuada através de pipes com nome (FIFOs). Todos os clientes têm acesso a um FIFO criado pelo servidor, cujo nome é o mesmo da macro "SERVER_FIFO_NAME", presente no header file "declarations.h", e escrevem nele os pedidos que desejam efetuar.

```
1  #define SERVER_FIFO_NAME "server_fifo"
```

Figura 1 - Macro com o nome do FIFO criado pelo servidor

A resposta efetuada pelo servidor, se fosse escrita num FIFO comum a todos os clientes poderia gerar uma leitura não pretendida, por parte de clientes, a respostas que não fossem direcionadas a eles, sendo que o cliente que deveria receber essa resposta, já não a poderá receber, devido ao facto de, num FIFO, as mensagens, para serem lidas, terem de ser consumidas. Então, resolvemos fazer com que cada cliente criasse o seu próprio FIFO, cujo nome é o mesmo da macro CLIENT_FIFO_NAME, presente no header file "declarations.h" como ilustrado na figura abaixo, onde o %d é substituído pelo pid do processo que cria o FIFO em questão. Desta forma, garantimos unicidade na resposta dada a cada cliente.

```
1  #define CLIENT_FIFO_NAME "client_%d_fifo"
```

Figura 2 - Macro com o nome do FIFO criado por cada cliente

Estruturas de dados utilizadas

Antes de explicar a arquitetura dos programas, iremos apresentar as estruturas de dados utilizadas no projeto, de forma a facilitar a compreensão por parte do leitor em relação ao funcionamento dos programas.

Struct "operation"

```
1  typedef struct operation
2  {
3      int nop, bcompress, bdecompress, gcompress, gdecompress, encrypt, decrypt;
4  } operation, *Operation;
```

Figura 3 - Campos da struct "operation"

Estrutura que contém 7 inteiros, cada um referente a uma transformação. É utilizada de forma a quantificar o número máximo de transformações concorrentes para cada tipo de transformação que o servidor permite realizar, a quantidade de transformações que são utilizadas num determinado momento e, também, no pedido enviado do cliente para o servidor, de forma a identificar qual a quantidade de transformações de cada tipo que o cliente pretende efetuar.

Struct "message_cs"

```
1  typedef struct message_cs
2  {
3      operation op;
4      pid_t client_pid;
5      int type;
6      int task_number;
7      int priority;
8      int n_args;
9      char commands[1024];
10 } message, *Message;
```

Figura 4 - Campos da struct "message_cs"

Estrutura utilizada na comunicação cliente-servidor, que contém uma struct "operation" (que guarda a quantidade de transformações de cada tipo que o cliente pretende efetuar), um pid_t, correspondente ao id do processo que efetuou o pedido, e um inteiro "type" associado ao tipo de pedido que o cliente pode fazer ao servidor, "proc-file" ou "status", associados aos números 0 e 1, respetivamente. Apresenta também um campo "task_number", que é da responsabilidade do servidor preencher conforme a ordem de chegada de pedidos ao mesmo, um campo "priority" que, tal como o nome indica, armazena a prioridade do pedido (0 a 5, caso seja um pedido "proc-file" ou 6, caso seja um pedido "status"), um array de caracteres que contêm os caminhos dos

ficheiros de entrada e saída e a sequência de transformações a aplicar ao ficheiro de entrada e, ainda, um inteiro ("n_args") correspondente ao número de argumentos presentes neste array.

Exemplos de utilização:

>> ./sdstore proc-file -p 3 Enunciado.pdf out nop bcompress encrypt

{

op: nop = 1; bcompress = 1, encrypt = 1 e restantes a 0

cliente_pid: pid do cliente

type: 0

task_number: 0 no cliente, no servidor ser-lhe-á atribuído outro valor

priority: 3

n_args: 5

commands: "Enunciado.pdf out nop bcompress encrypt"

}

>> ./sdstore status

{

op: tudo a 0

cliente_pid: pid do cliente

type: 1

task_number: 0 no cliente, no servidor ser-lhe-á atribuído outro valor

priority: 6

n_args: 0

commands: ""

}

Struct "node"

```
1  typedef struct node
2  {
3      message commands;
4      struct node *next;
5  } Node;
```

Figura 5 - Campos da struct "node"

Estrutura que representa a fila de espera do servidor. Consiste numa lista ligada, onde a cabeça da lista representa a próxima operação a executar. Na organização da lista ligada adotamos como primeiro critério a prioridade do pedido e, de seguida, a ordem de chegada do mesmo. Assim sendo, quando um pedido é adicionado, fica sempre à frente de qualquer outro com menor prioridade, mas atrás de todos os pedidos que apresentem maior prioridade, ou, igual prioridade e que tenham sido adicionados anteriormente. Quanto à remoção de pedidos, é sempre removido aquele que se encontra presente à cabeça da lista.

Struct "list"

```
1  typedef struct list
2  {
3      message commands;
4      struct list *next;
5  } List;
```

Figura 6 - Campos da struct "list"

Estrutura que representa a lista dos pedidos que estão em execução num dado momento. Em termos de dados é idêntica à struct "node", variando apenas na sua forma de funcionamento. Contrariamente à struct "node", a struct "list" adiciona sempre novos pedidos ao fim da lista, visto que a ordem com que se apresentam nesta estrutura de dados é irrelevante. Quanto à remoção, é passado como argumento o pid do cliente que efetuou o pedido a remover, percorre-se a lista até encontrar um pedido cujo pid guardado na sua estrutura (struct "message_cs") seja igual ao pid do argumento passado e remove-se esse pedido da lista.

Programa Servidor

O programa servidor recebe dois argumentos pela linha de comandos (caminhos para o ficheiro de configuração e para a diretoria onde os executáveis responsáveis pelas transformações estão armazenados, respetivamente). Depois de verificar se o ficheiro de configuração e a diretoria referida existem, é guardado numa estrutura de dados ("operation"), em memória, o número máximo de ocorrências concorrentes permitidas a cada transformação e, é criada uma nova estrutura de dados do mesmo tipo, onde a referência para cada transformação é inicializada a 0 (representa o número de ocorrências em tempo real). De seguida, criamos um pipe anónimo e um processo filho, encarregue de gerir a fila de espera e a lista dos pedidos "a executar", cujas funcionalidades explicaremos detalhadamente mais à frente.

Exemplo de ficheiro de configuração

O ficheiro de configuração recebido como argumento pelo programa servidor tem de seguir a seguinte estrutura:

- 7 linhas (uma por transformação);
- Cada linha deve apresentar o nome da transformação, seguido de um espaço e um inteiro correspondente ao número máximo de transformações concorrentes do tipo em questão, tal como na imagem abaixo apresentada;

Figura 7 – Estrutura do ficheiro de configuração do servidor

Processo pai

No processo pai é criado o FIFO anteriormente apresentado.

De seguida, fazemos open desse FIFO em modo **O_RDONLY** (que bloqueia até a um cliente abrir o descritor de escrita) e, fazemos também open, no servidor em modo **O_WRONLY**, de forma que a system call read nunca retorne EOF quando é chamada.

```c
while ((read_res = read(fiford, &messageFromClient, sizeof(message))) > 0)
{
    snprintf(client_fifo, 1024, CLIENT_FIFO_NAME, (int)messageFromClient.client_pid);

    if ((fd_client_fifo = open(client_fifo, O_WRONLY)) == -1)
        perror("open");

    if (isValid(&messageFromClient.op, maxOperations) && !sig_term)
    {
        task_n++;
        messageFromClient.task_number = task_n;
        write(fd_client_fifo, "pending", 8);
        close(fd_client_fifo);
        write(p[1], &messageFromClient, sizeof(message));
    }
    else
    {
        write(fd_client_fifo, "concluded", 10);
        close(fd_client_fifo);
    }
}
```

Figura 8 – Ciclo responsável pela leitura de pedidos submetidos pelos clientes

A seguir, é executado o ciclo acima apresentado, que está encarregue, basicamente, de ler aquilo que os clientes escrevem no FIFO do servidor e verificar se a sequência de transformações ("operation") que o cliente pede para executar respeita os limites impostos pelo ficheiro de configuração. Por exemplo, se o limite máximo de transformações "nop" que podem ser executadas concorrentemente for 4 e uma operação pedida pelo cliente apresentar mais de 4 "nops", então, não é uma operação válida. Caso a operação seja válida, é escrita no pipe criado anteriormente e, é escrito no FIFO respetivo ao cliente que efetuou o pedido a mensagem "pending". Caso contrário, é descartada a operação e, é escrito no FIFO respetivo ao cliente que efetuou o pedido a mensagem "concluded".

Processo filho

O processo filho é o responsável pela gestão da fila de espera e da lista dos pedidos que estão em execução. A estratégia aqui utilizada foi a seguinte: temos um ciclo que tem como função ler pedidos do pipe anónimo criado pelo pai. De seguida, verifica se esse pedido está na lista dos pedidos a executar. Se sim, retira-o da lista, decrementa os recursos utilizados pelo mesmo e volta ao estado de leitura (bloqueando até algo ser escrito no pipe). Se não, adiciona-o à lista de espera. A seguir, e ainda nesse ciclo, criamos outro ciclo, que corre enquanto a lista de espera não estiver vazia e o próximo comando a executar na lista de espera tiver disponíveis os recursos necessários à sua execução. Enquanto essas duas condições se verificam, criamos um novo processo (que trata da execução do pedido), incrementamos os recursos que serão utilizados por ele e retiramo-lo da fila de espera.

Esse novo processo, criado para a execução do pedido, depois de o executar escreve no pipe anónimo o pedido que executou, de forma a sinalizar ao processo responsável pela gestão das filas que o pedido já foi executado e já não está a utilizar recursos.

Programa Cliente

O programa cliente recebe como argumento o tipo de pedido que pretende enviar para o servidor ("proc-file" ou "status"). Depois de fazer algumas verificações sobre os argumentos introduzidos pelo utilizador, o cliente cria um FIFO, da maneira especificada no ponto "Comunicação entre cliente e servidor". De seguida, é criada uma struct "message_cs" (estrutura de dados utilizada para a comunicação cliente-servidor) e preenchida conforme os argumentos introduzidos pelo utilizador. Tendo já tudo verificado e preenchido, é submetido no FIFO do servidor o pedido efetuado pelo cliente. Por último, fazemos open do FIFO criado pelo cliente em modo **O_RDONLY** (que bloqueia até um cliente abrir o descritor de escrita deste) e, fazemos também open do mesmo FIFO em modo **O_WRONLY**, com o objetivo de, aquando de uma chamada read, esta nunca retornar EOF, à semelhança do que fazemos no servidor. Criamos um ciclo que lê do FIFO criado e que termina quando lê um array de caracteres que contenha como substring a mensagem "concluded".

Operações

Como temos acesso ao id do processo (cliente) que efetuou o pedido, antes de o executar, abrimos em modo **O_WRONLY** o seu FIFO e guardamos numa variável o seu descritor de ficheiro. Quer para a operação "status", quer para a "proc-file", desenvolvemos funções que iremos apresentar de seguida.

Função "status"

Recebe como argumento apontadores para a lista dos pedidos em execução, a estrutura de dados onde está guardado o número máximo de transformações concorrentes para cada tipo de transformação e a estrutura onde está guardada a quantidade de transformações de cada tipo que estão a executar em tempo real. Recebe também um descritor de ficheiro, correspondente ao FIFO do cliente que pediu a realização da operação "status".

Como esta operação apenas tem como objetivo informar o cliente sobre o estado do servidor, o seu funcionamento é o seguinte:

- Percorre a lista de pedidos a executar e escreve no descritor passado como argumento os pedidos que nela estejam contidos;
- Como tem informação sobre os recursos máximos a utilizar e os que estão a ser utilizados, escreve no descritor passado como argumento essa informação para cada uma das transformações.

Função "proc-file"

Recebe como argumento um array com as transformações a executar e ainda o caminho dos ficheiros de entrada e de saída. Para além disso, recebe o número de elementos deste array, uma string correspondente ao caminho onde os executáveis das transformações estão armazenados e também um descritor de ficheiro, correspondente ao FIFO do cliente que pediu a realização da operação proc-file.

Primeiramente, esta função escreve a mensagem "processing" no FIFO cujo o descritor foi passado como argumento. De seguida, cria a diretoria onde será guardado o ficheiro de saída, caso esta não exista, e, ainda, são abertos o ficheiro de entrada em modo leitura (**O_RDONLY**) e o ficheiro de saída com as flags **O_WRONLY**, **O_CREATE** e **O_TRUNC**. De seguida, o descritor do ficheiro de entrada é redirecionado para o STDIN e o descritor do ficheiro de saída para o STDOUT, e estes descritores são fechados. Caso o número de transformações a executar, que designaremos por n, seja superior a 1, é criada uma pipeline constituída por n-1 pipes. Posteriormente, são lançados n processos-filho (numerados de 0 a n-1) sequencialmente (mas que executam concorrentemente), sendo que em cada um destes processos é executada uma das transformações presentes no array recebido, pela ordem na qual estão dispostas no mesmo. Assim sendo, em cada processo filho, excetuando o primeiro e o último processos-filho lançados, são redirecionados o STDIN e STDOUT para a extremidade de leitura do pipe com número anterior ao número do processo em questão e para a extremidade de escrita do pipe com número igual ao do processo, respetivamente. Quanto ao primeiro processo filho lançado, é apenas redirecionado o STDOUT para a extremidade de escrita do primeiro pipe da pipeline, uma vez que a leitura efetuada neste processo é feita a partir do ficheiro de entrada. Do mesmo modo, no último processo lançado é apenas redirecionado o STDIN para a extremidade de leitura do último pipe da pipeline, uma vez que a escrita será realizada no ficheiro de saída.

Caso seja executada apenas uma transformação, não é necessária a utilização de pipes. Neste caso, e já efetuado o redireccionamento do STDIN e STDOUT para os ficheiros de entrada e saída,

respetivamente, tal como explicado acima, basta lançar um processo filho para executar a transformação desejada.

Por fim, é escrito no descritor passado como argumento a mensagem "concluded", juntamente com o número de bytes dos ficheiros de entrada e de saída e fechado o descritor.

Funcionalidades avançadas

Prioridades

As prioridades são geridas pela fila de espera implementada e, portanto, já foi explicada a sua implementação nos tópicos apresentados anteriormente.

Tamanho dos ficheiros de input e output

De modo a implementar esta funcionalidade avançada recorremos à system call lseek. Assim sendo, para obter o número de bytes do ficheiro de entrada, o cursor é deslocado para o final deste ficheiro, na função "proc-file", através da utilização da system call referida, com a flag **SEEK_END** e offset igual a 0 bytes. Deste modo, a system call retorna o número de bytes do ficheiro de entrada. De seguida, o cursor é novamente colocado no início do ficheiro de entrada, de modo que lhe a função "proc-file" lhe possa aplicar as transformações desejadas. Após ser gerado o ficheiro de saída, este é novamente aberto na função "proc-file" e o cursor é também deslocado para o fim do mesmo, através da system call lseek, com a flag **SEEK_END** e offset igual a 0 bytes. De modo similar ao ficheiro de entrada, a system call retorna agora o número de bytes do ficheiro de saída. Finalmente, o número de bytes dos ficheiros de entrada e de saída são colocados numa string e esta é escrita no FIFO do cliente que submeteu o processo em questão ao servidor, de modo que este possa receber esta mensagem e imprimi-la no seu STDOUT.

Sinal SIGTERM

Para implementar o término do servidor de forma graciosa quando este recebe o sinal SIGTERM, declaramos os dois descritores de ficheiros (que abrem, no próprio servidor, o FIFO criado pelo mesmo em modo de escrita e de leitura, respetivamente) como variáveis globais e, no handler do SIGTERM fechamos o descritor de escrita, de forma que o read que mantém o ciclo do processo pai, ilustrado na Figura 7, a correr retorne EOF e, consequentemente, pare. De seguida, escrevemos no pipe anónimo, que comunica com o processo filho, uma mensagem na qual o valor do campo "type" é igual a -1. Desta forma, o processo filho reconhece que o servidor recebeu um

SIGTERM, executa até a fila de espera ficar vazia, termina e faz exit. O processo pai espera que o filho acabe de executar, terminando assim a execução do servidor.

Conclusão

De um modo geral, conseguimos criar um programa capaz de satisfazer o objetivo principal deste trabalho prático: "armazenar cópias de ficheiros de forma segura e eficiente, de modo a poupar espaço em disco". Assim sendo, tentamos desenvolver um programa com o comportamento pedido no enunciado, implementando para tal todas as funcionalidades sugeridas, desde as mais básicas até às mais avançadas. Para realização deste trabalho prático, recorremos às funcionalidades lecionadas em cada um dos guiões das aulas práticas, sendo que este projeto, de maior dimensão, nos permitiu aprofundar os nossos conhecimentos e entender melhor o funcionamento de cada uma destas funcionalidades.