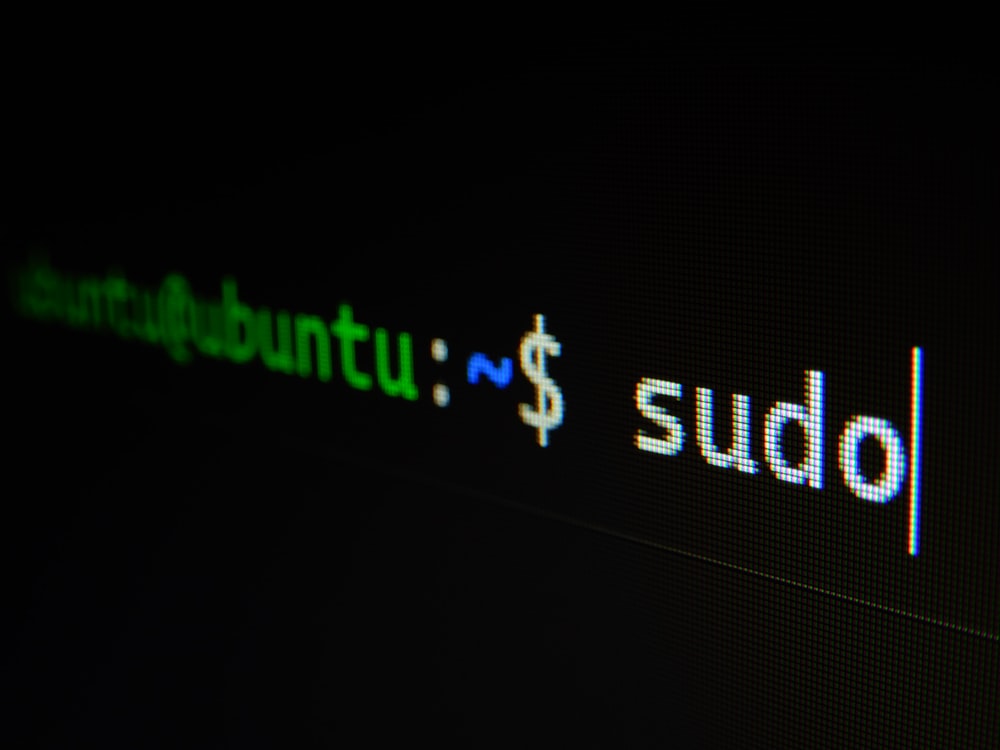
Relatório Trabalho Prático 

Grupo 99:

João Paulo Machado Abreu | a91755

Ricardo Cardoso Sousa | a96141

Tiago Luís Pereira Ferreira | a97141

Índice

[Introdução 1](#_Toc103544905)

[Comunicação entre cliente e servidor 1](#_Toc103544906)

[Estruturas de dados utilizadas 2](#_Toc103544907)

[Struct “operation” 2](#_Toc103544908)

[Struct “message\_cs” 2](#_Toc103544909)

[Struct “node” 4](#_Toc103544910)

[Struct “list” 4](#_Toc103544911)

[Programa Servidor 4](#_Toc103544912)

[Processo pai 5](#_Toc103544913)

[Processo filho 6](#_Toc103544914)

[Programa Cliente 6](#_Toc103544915)

[Operações 6](#_Toc103544916)

[Função “status” 7](#_Toc103544917)

[Função “proc-file” 7](#_Toc103544918)

[Funcionalidades avançadas 8](#_Toc103544919)

[Prioridades 8](#_Toc103544920)

[Tamanho dos ficheiros de input e output 8](#_Toc103544921)

[Sinal SIGTERM 9](#_Toc103544922)

# 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.

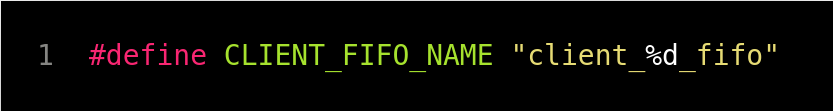
# Comunicação entre cliente e servidor

A comunicação entre clientes e servidor é efetuada através de *named pipes* (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.

Uma imagem com texto

Descrição gerada automaticamente

Figura

A resposta efetuada pelo servidor, se fosse escrita num FIFO comum a todos os clientes poderia gerar uma leitura não pretendida, por parte dos 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 o criou. Desta forma, garantimos unicidade na resposta dada a cada cliente.

Figura

# 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.

## Uma imagem com texto Descrição gerada automaticamenteStruct “operation”

Figura

Estrutura que contém 7 inteiros, cada um referente a uma transformação. É utilizada de forma a quantificar o máximo de transformações concorrentes para cada transformação que o servidor permite realizar, a quantidade de transformações a ser 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”

Uma imagem com texto

Descrição gerada automaticamente

Figura

Estrutura utilizada na comunicação cliente-servidor, que contém uma struct operation (que corresponde à quantidade de transformações de cada tipo que o cliente pretende efetuar), um pid\_t, correspondente ao id do processo que efetuou o pedido, um 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 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 restante 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

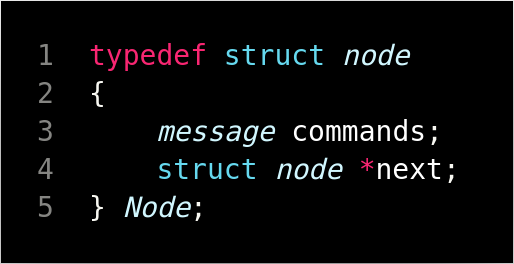
priority: 6

n\_args: 0

commands: “”

}

## Struct “node”



Figura

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 pedidos que apresentem a mesma prioridade e tenham sido adicionados anteriormente. Quanto à remoção de pedidos, é sempre removido aquele que se encontra presente à cabeça da lista.

## Struct “list”

Uma imagem com texto

Descrição gerada automaticamente

Figura

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 a 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 ficheiro de configuração e para a pasta onde os executáveis responsáveis pelas transformações estão guardados, respetivamente). Depois de verificar se o ficheiro de configuração e a pasta 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 zero (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.

## 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.

Uma imagem com texto

Descrição gerada automaticamente

Figura

A seguir, é executado o ciclo acima apresentado, que está encarregue, basicamente, de ler o que os clientes escrevem no FIFO do servidor e verificar se a sequência de transformações (operação) que o cliente pede para aplicar 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 dados 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 do 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 a um cliente abrir o descritor de escrita) e, fazemos também open do mesmo FIFO em modo **O\_WRONLY**, com o objetivo de, aquando de uma chamada read, nunca retornar EOF, à semelhança do que fazemos no servidor. Criamos um ciclo que faz read 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 estão guardadas o máximo de transformações concorrentes para cada transformação e a estrutura onde está guardada a quantidade de transformações que estão a ocorrer em tempo real para cada uma das transformações. 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 estão guardados 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, e 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 do ficheiro de entrada e de saída e fecha o descritor.

# Funcionalidades avançadas

## Prioridades

As prioridades são geridas pela fila de espera 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 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 STDIN.

## Sinal SIGTERM

Para implementar o sinal SIGTERM, declaramos os dois descritores de ficheiros (que abrem, no 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 correr. 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 e depois faz exit também, terminando assim a execução do servidor.