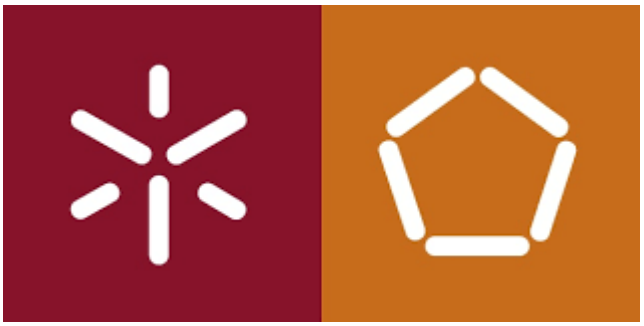


UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

PROJETO DE
SISTEMAS OPERATIVOS

GRUPO
José Carvalho (a94913)
Miguel Silva (a97031)
Ana Rita Poças (a97284)

ÍNDICE

Introdução

No âmbito do projeto a desenvolver na Unidade Curricular de Sistemas Operativos foi-nos proposto a implementação de um servidor que permite aos seus utilizadores o aplicar certas operações relativas a ficheiros de forma eficiente e segura.

O conjunto das operações são relacionadas a compressão e cifragem de ficheiros.

É possível haver vários pedidos a serem atendidos ao mesmo tempo e é permitido aos clientes consultarem o estado do servidor.

Estruturas Utilizadas

De forma a representarmos os dados necessários para conseguirmos implementar as funcionalidades necessárias, implementamos 4 estruturas de dados:

- ***MAXOPERATION :*** estrutura criada a fim de nos ser possível controlar o número de operações a decorrer no nosso programa, de forma a não exceder o número máximo de cada operação. Desta forma, temos uma string *operation* de tamanho fixo 20 que irá conter o nome da operação, um inteiro *number* (número atual de execuções de cada operação) e um inteiro *max* (número máximo que o servidor irá permitir para cada operação).

- ***EXECSTATUS*** : estrutura criada de forma a controlar o estado do nosso programa, isto é, guardar os pedidos a serem executados pelo servidor num determinado instante. Desta forma, temos uma string que guarda a string do pedido, *pedido*, de tamanho fixo 1024, um inteiro *nrpedido* (o número da operação) e um apontador para o pedido seguinte. Esta estrutura é representada sob a forma de uma lista ligada (*).

- ***OPERATION:*** estrutura criada com o objetivo de controlar o estado das operações do nosso programa. Assim, temos uma variável do tipo *EXECSTATUS* que guarda o estado atual do programa, um inteiro *numtasks* (o número de tarefas atuais) e um array de tamanho fixo 7 (devido a termos apenas 7 operações) do tipo *MAXOPERATION*.

- **TASK:** estrutura cuja função é representar um pedido, conta com três campos, duas strings de tamanho fixo 1024, uma dessas para armazenar por extenso o pedido, outra que armazena o nome do pipe para onde o servidor irá ter que mandar mensagens para o cliente relativas ao estado do seu pedido, e por último um inteiro chamado argumentos que tem o número de argumentos do pedido, semelhante ao argc de uma função main.

- **WAITQUEUE:** esta estrutura foi a última que criamos mas é bastante importante no projeto pois é através dela que representamos a nossa fila de espera de pedidos, sobre a forma de uma lista ligada (*). Esta estrutura conta com os seguintes campos, um apontador para uma string de tamanho fixo 1024, que irá servir como um array de strings, chamado pedido, uma string

de tamanho fixo 1024 chamada pedidob, três inteiros, um chamado espacos. o segundo file, e o último prioridade, um array de inteiros de tamanho fixo 7 chamado array, uma variável do tipo long chamada time, e por último, um apontador para o próximo pedido na fila de espera. O apontador pedido, como já foi referido, irá ser um array de strings em que cada posição é uma parte do pedido (é exatamente a mesma coisa que a componentes da função main do programa ./sdstored), pedidob é uma cópia da string do pedido que acaba por ser útil depois no status do servidor, temos esta cópia pois ao fazer *strsep* para criar a tal 'componentes' na função main, os apontadores irão mudar da mesma, logo, desta forma não temos problema no acesso a campos por mudança das suas posições de memória, espacos tem o número de argumentos do pedido, file armazena o descritor do pipe com nome do cliente, o array de inteiros o número de operações de cada tipo em execução, o tempo representa o instante em que o pedido foi submetido na fila e a prioridade representa a prioridade do pedido.

*****Nós nas estruturas **WAITQUEUE** e ***EXECSTATUS*** usamos listas ligadas devido ao facto que em ambas as situações irão ser realizados tantas remoções como inserções e como em arrays as remoções são uma dor de cabeça em termos de eficiência, optamos por a inserção ser mais lenta, mas em compensação a remoção ser muito mais eficiente, que é o caso das listas ligadas.

Implementação das funcionalidades básicas

Inicialização Servidor

Para inicializar o servidor, é preciso invocar o executável ./sdstored. Este executável tem que receber 2 argumentos, o nome do ficheiro que contém o número máximo de operações que podemos fazer num determinado instante e um caminho para onde se encontram os executáveis das operações que poderemos realizar.

Nós criamos uma estrutura chamada ***'OPERATION'*** e de seguida abrimos o tal ficheiro texto que contém o número máximo de cada operação e depois realizamos parse a cada linha.

Para guardar o caminho dos executáveis, criamos uma variável global do executável ./sdstored denominada de ***'path'*** em que iremos guardar o tal caminho sob a forma de string.

Depois de criar a tal estrutura e guardar o caminho dos executáveis, o servidor cria um pipe com nome, denominado de ***'tmp/cliente_server'***. Este pipe será usado para a receção de pedidos.

Depois destes processos, o servidor entra em modo operacional.

Envio de um pedido (./sdstore)

Neste momento iremos explicar o funcionamento do executável ./sdstore.

Quando queremos mandar um pedido para o servidor, temos que recorrer ao executável ./sdstore.

Para isto o ./sdstore usa como auxilio estrutura ***'TASK'*** que vai servir de mensagem do pedido.

Para não perder a ligação de servidor cliente e de o servidor não mandar mensagens sobre o estado do pedido para o cliente errado, pensamos em criar pipes com nome únicos para cada pedido. Estes pipes serão usados para o servidor mandar mensagens para o devido cliente sobre o estado do seu pedido. Para criarmos pipes com nome diferentes pensamos em usar o número de processo do executável ./sdstore, fazendo assim a diferenciação de diferentes pedidos. Para isto usamos a função ***'getpid'***, convertemos o valor de retorno da mesma para uma string e de seguida criamos o pipe com nome com o tal número de processo.

Depois é colocado na estrutura ***'TASK'*** no campo 'pedido' uma string, que contém todos os argumentos do executável ./sdstore, separando cada um por espaços e também é colocado o nome do pipe único do cliente numa string e o número de argumentos deste executável.

Neste momento é feita uma escrita no pipe com nome ***'tmp/cliente_server'*** desta tal estrutura auxiliar, usando obviamente o descritor de escrita. Depois o servidor irá ter este pipe com nome aberto em modo leitura e assim irá receber a tal estrutura, que contém o que já foi referido anteriormente.

Sobre o processamento concreto do pedido dentro do servidor iremos abordar mais à frente neste relatório.

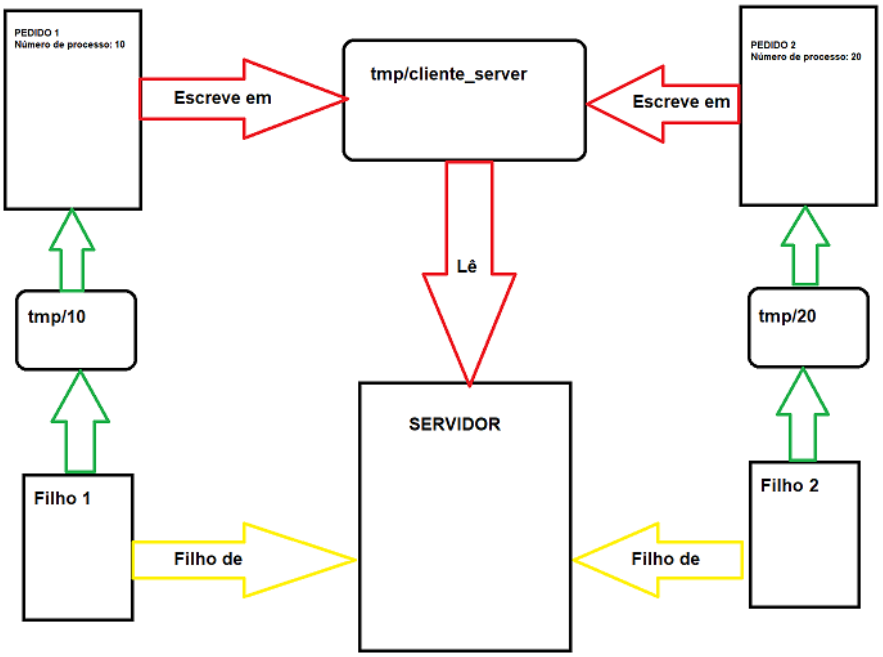
O servidor nesta altura irá abrir o tal pipe com nome único do cliente sob o modo escrita e o cliente sob o modo leitura e o servidor começa a mandar as mensagens sobre o estado do pedido. Quando o servidor não tem mais nada para mandar é feito o fecho do descritor de escrita do pipe com o nome do pedido no servidor, e o ./sdstore irá perceber que as mensagens para si acabaram devido ao facto de a função read retornar 0 pois não há descritor de escrita.

O ./sdstore neste momento vai também fechar o descritor de leitura do pipe único que o mesmo criou e vai verificar se o mesmo foi processado com sucesso ou não. Isto irá depender diretamente do número de mensagens enviadas do servidor para o cliente e do número de argumentos do próprio executável. Se o pedido for inválido o ./sdstore só irá receber 2 mensagens 'Pending' e 'Pedido Inválido' então não é feito mais nada neste caso e o mesmo se aplica quando o pedido for para consultar o status do servidor.

Caso o número de mensagens enviadas pelo servidor seja maior que 2 e o número de argumentos do ./sdstore, o executável do cliente abre novamente o ***'tmp/cliente_server'*** para informar o servidor que o mesmo já terminou com a mensagem 'acabei (pedido)'. Pensamos nesta ligação pois quem irá processar o pedido vai ser um processo filho do servidor, e o processo pai do servidor perde a conexão com o cliente, pois esta relação só está estabelecida para o tal filho. Esta escrita serve sobretudo para o servidor saber que tem x operações que podem ser libertadas e que estarão disponíveis para pedidos na fila de espera ou para e futuros pedidos.

Para acabar o ./sdstore faz unlink do pipe com nome que o mesmo criou, visto que não será utilizado por mais ninguém.

Apresentamos agora uma imagem a esquematizar o funcionamento desta parte. A imagem não apresenta o processo de forma cronológica.



Receção e processamento de um pedido (./sdstored)

Como já foi referido na secção anterior os pedidos são enviados para o pipe com nome ***'tmp/cliente_server'***. O servidor vai ter então ter este pipe aberto e no processo de leitura irá preencher os campos de uma ***'TASK'***.

Neste momento o servidor separa a string por strings, usando como fator de divisão o caractere ' '. Depois de efetuar esta divisão primeiramente o servidor verifica se é um pedido a informar que um determinado cliente já foi atendido até ao fim, ou se é um pedido novo.

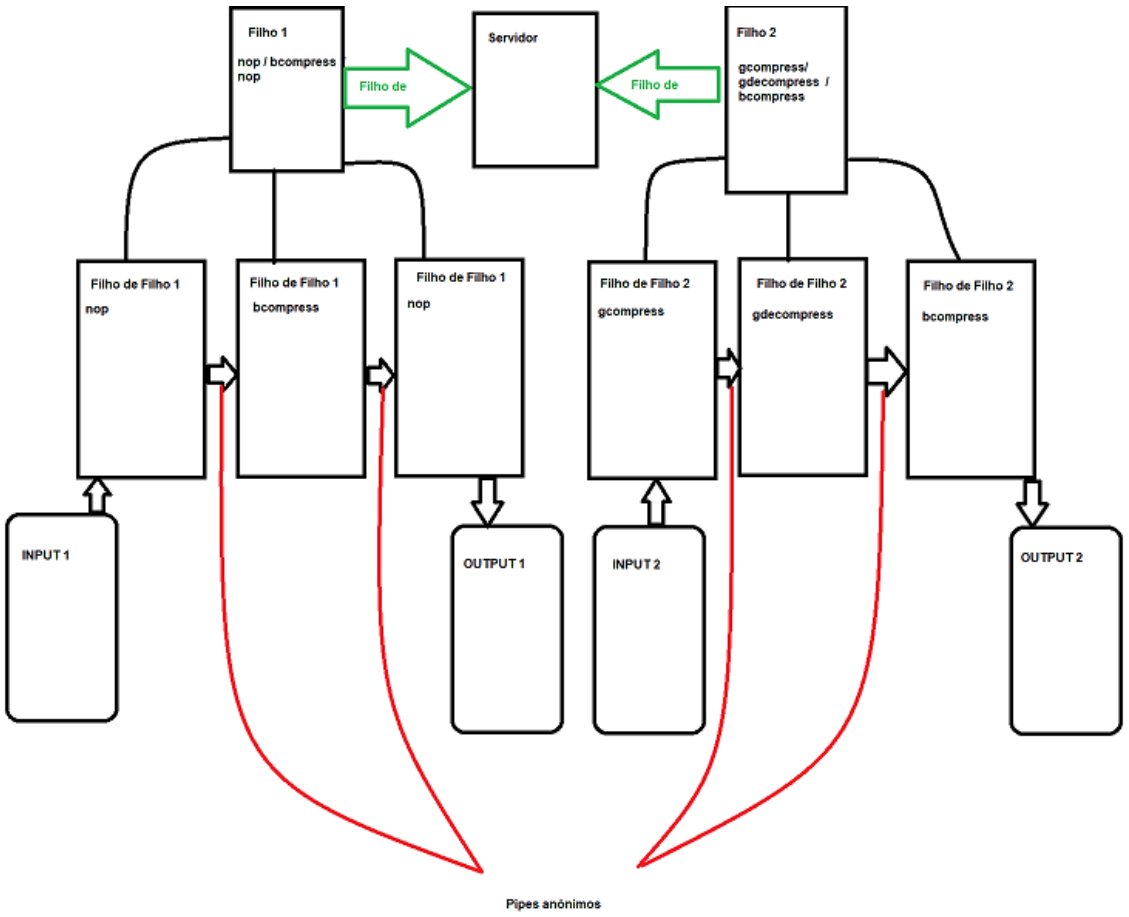
Se for a informar o fim de um pedido, o servidor primeiramente reduz na estrutura 'OPERATIONS' as operações do devido pedido e vai consultar a fila de espera (a fila de espera iremos explicar mais à frente como funciona, devido à prioridade influenciar as colocações dos pedidos na mesma). Se a queue tiver elementos o servidor primeiramente irá verificar quais dos mesmos poderá executar. Se um pedido da queue não poder ser executado, este continua na fila e o servidor continua a procurar no resto dos elementos da fila de espera até chegar ao fim. Se um pedido da fila poder ser executado o servidor incrementa as devidas operações do pedido e cria um filho para executar o mesmo e continua a ver na fila há mais pedidos que podem ser executados.

No outro caso, ao informar que é um pedido novo, o servidor primeiramente verifica se o pedido é válido, isto é, o número de ocorrências da operação x têm que ser menor ou igual ao número máximo de ocorrências da operação x. Se um pedido for invalido o servidor manda a mensagem para o devido cliente a informar de tal e o pedido não é executado e é fechado o descritor de escrita do pipe da comunicação servidor cliente. Por outro lado, se o pedido for válido, o servidor neste momento verifica se tem disponibilidade para executá-lo, se tiver cria um filho que irá ser responsável por executá-lo. Caso contrário adiciona o pedido na fila de espera e em ambos os casos, o servidor volta à parte de fazer read do

pipe com nome *'**tmp/cliente_server**'*.

Se um pedido entra em execução, primeiramente é criado um array com pipes anônimos. Este array vai ter comprimento variável, dependendo do número de operações do respetivo pedido e estes pipes irão servir para a comunicação de processos entre as diversas operações. Neste momento é criado um filho por cada operação em que cada um vai ler do pipe a si associado o input (descritor 0) e vai escrever para o pipe do próximo filho (descritor 1), salvo a exceção do último filho que irá escrever para o ficheiro resultante e o primeiro que irá ler do ficheiro de input. Estas associações são feitas com base em mudanças dos descritores 0 e 1 para os respetivos descritores de leitura e escrita dos pipes. É importante salientar que este processo é feito de forma concorrente, ou seja, primeiramente são criados n filhos e depois é que começa a execução dos mesmos. No final o processo pai associado à execução do respetivo pedido irá esperar que todos os seus filhos acabem primeiro e depois parte para uma parte que conta o número de bytes do ficheiro de input e output (iremos explicar melhor a forma como resolvemos essa parte mais à frente) e depois termina com um valor de exit 0.

Uma imagem a demonstrar de como é feita a execução de 2 pedidos em que um tem como ficheiro de input : "INPUT 1" e ficheiro de output: "OUTPUT 1" e o 2º pedido têm os mesmos nomes para os ficheiros de input e output, substituindo apenas o 1 por 2.



Status

Para representar o nosso estado de execução, usamos como referido na primeira secção a estrutura **EXECTATUS**, que é trabalhada na função *PrintStatus* do *sdstored.c*. Esta função basicamente escreve no pipe com nome do cliente associado, a informação atual guardada na estrutura.

A referida função é invocada quando recebemos um pedido do tipo *./sdstore status* ou *./sdstore*, escrevendo então a informação do estado do servidor.

Já a adição e remoção de pedidos é resolvida pelas funções *addPedidoOperation* e *removePedidoOperation* respetivamente, ambas do *sdstored.c*.

A informação escrita para o cliente surge no seguinte formato:

- Um seguimento de linhas de texto com a estrutura "Task %d: %s\n", onde o valor numérico refere-se ao número do pedido e a string ao estado de execução do pedido (campo *execstatus* da estrutura).
- Sete linhas no formato "Transf: %s (%d/%d) (running/max)\n", onde a primeira string representa o tipo de operação e os dois números o número de operações desse tipo em execução e o número máximo de operações desse tipo.

Implementação das funcionalidades avançadas

Contagem de bytes

Relativamente à funcionalidade avançada de contar o número de bytes, primeiramente é feito o open de dois ficheiros x e y, de seguida é feita a contagem dos bytes de do ficheiro x e depois é que começa a contagem do y. A contagem dos bytes têm por base invocar várias vezes a função read cujo valor de retorno é o número de bytes lidos e a cada read que realizamos somamos a uma variável que têm o número de bytes lidos dos reads feitos para trás.

Esta funcionalidade é 100% executada de forma sequencial por um único processo. Escolhemos fazer assim devido ao facto de que poderemos ter já muitos processos criados do processador, pois para executar um pedido são criados n + 1 processos, em que n é o número de operações. Nestes casos em que temos muitos processos, se criarmos ainda mais processos, a performance da máquina decai de forma exponencial devido ao facto de na memória haver muito mais ocorrências de swaps / troca de páginas pois processos distintos não partilham memória e também não esquecer o facto de haver muito mais trabalho para haver escalonação dentro do CPU o que resulta menos tempo de CPU para cada processo.

É importante relembrar que esta operação não é feita pelo processo principal do servidor, mas sim pelo filho que executou o respetivo pedido em que teve como ficheiro de input x e ficheiro de output y.

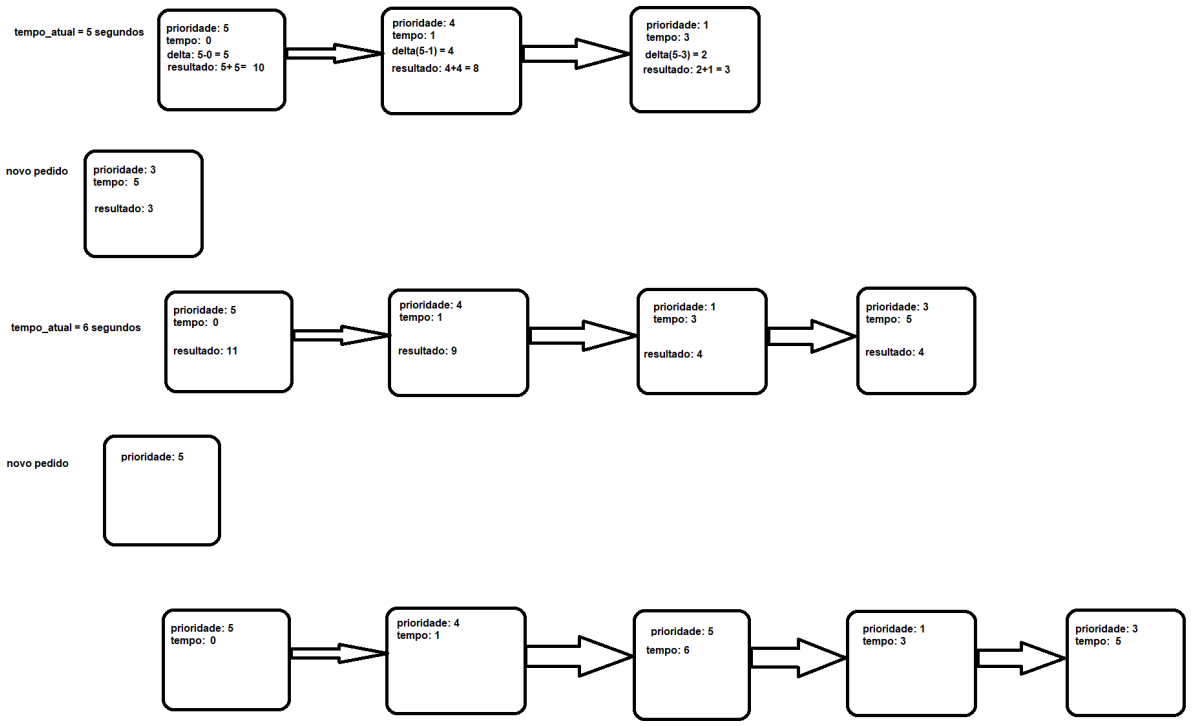
Prioridades e fila de espera

No servidor, as prioridades são importantes para o posicionamento na fila de espera. Se quisermos inserir um pedido em que a prioridade seja inferior em relação à prioridade de um elemento da queue, obrigatoriamente esse pedido irá ser colocado depois desse tal elemento. O mesmo não acontece sempre, na relação inversa, isto é, inserir na fila um elemento com maior prioridade do que outro que lá está, pois tivemos em consideração o tempo em que o pedido foi submetido. O nosso algoritmo de comparação é básico, como já foi referido, têm em conta o tempo de submissão e a prioridade. Ele realiza a diferença do tempo atual com o tempo em que o pedido foi colocado na fila, em segundos, e depois de ter esse valor soma à prioridade do respetivo pedido e dá um resultado. Este resultado é comparado diretamente com a prioridade do pedido que chegou e se esse resultado for menor do que a prioridade do novo pedido, então este é inserido na posição em questão, à frente do pedido que foi comparado. Caso contrário, isto é, o resultado da soma for maior ou igual do que a prioridade, o pedido não é inserido na atual posição. Se todos os elementos da fila de espera tiverem um resultado do algoritmo maior ou igual do que a prioridade do novo pedido, este é colocado no fim da fila.

Achamos por bem ter em conta o tempo em que o pedido se encontra na queue para evitar aqueles casos em que um pedido está sempre na fila.

De relembrar que depois quando o servidor consulta a queue para verificar quais pedido é que pode executar, o servidor percorre do início ao fim, ou seja, os pedidos que se encontram nas primeiras posições têm uma maior chance de entrar primeiro em execução do que os restantes. Embora possam ocorrer situações como poder haver um pedido no topo da fila em que o servidor ainda não esteja disponível para o executar, no entanto encontra-se disponível executar outro pedido da fila, então esse tal pedido mais à frente na fila é executado primeiro.

Uma imagem a simplificar a forma como são inseridos os pedidos na fila de espera:



Sinais para o servidor terminar de forma graciosa

Para o servidor terminar de forma graciosa, é necessário enviar um sinal do tipo ‘SIGTERM’. Para este caso criamos uma função que vai primeiramente fechar o servidor a novos pedidos, isto é, meter a variável global ***‘signal’*** com valor 0. Estes pedidos podem ser para ver o estado do servidor, ou com a opção ‘proc-file’. O servidor só acaba quando a fila de espera estiver vazia e não houver nenhum pedido em execução. Quando o servidor é fechado é feito o unlink do pipe com nome ***‘tmp/cliente_server’***. É importante referir, que o servidor pode ter recebido o sinal e ele rejeita novos pedidos, mas no entanto o servidor ainda continua disponível para receber pedidos do tipo ‘acabei’ de forma a submeter os pedidos que estão na fila de espera em execução e também perceber se há mais algum pedido em execução, e para isto é necessário o pipe com come ***‘tmp/cliente_server’***.

Conclusão

Para concluir, nós achamos que conseguimos atingir os objetivos deste projeto de Sistemas Operativos, pois fomos capazes de arranjar uma solução que trabalhe com aspetos como concorrência de processos, o uso de pipes anônimos, pipes com nome e sinais. Não quisemos estar a complicar muito o programa, mas pensamos em certas coisas que poderíamos melhorar ou fazer de outra forma. Uma que pensamos foi, em vez do ./sdstore mandar a mensagem ‘acabei’ para informar o processo pai do servidor que pode libertar x operações, poderíamos usar sinais de forma a que fossem os processos filhos do servidor a informá-lo que estes já acabaram e assim o programa ./sdstore só tinha que se preocupar em mandar o pedido uma vez e receber as mensagens do servidor.