



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

João Augusto Macedo Moreira a93326
Pedro Miguel Marques Ferreira a93303
Teresa Costa Pires Gil Fortes a93250

Grupo 77

29 de maio de 2022

SO

Índice

1	Introdução	1
1.1	Apresentação do Caso de Estudo	1
1.2	Principais objetivos	1
1.3	Estrutura do Relatório	1
1.4	Funcionalidades Avançadas	2
2	Arquitetura	3
2.1	Estrutura de dados	3
2.2	Validação Input	3
2.3	Comunicação Cliente-Servidor	3
2.4	Variáveis Globais	4
3	Programa Cliente	5
3.1	Validação dos argumentos	5
3.2	Comunicação por fifos	5
4	Programa Servidor	6
4.1	Execução	6
4.2	Prioridades	7
4.3	Sinais	8
4.4	Alarme	8
4.5	Finalização Graciosa	9
5	Testes	10
6	Conclusão	13

Lista de Figuras

1	<i>Struct</i> Pedido criada.	3
2	Variáveis globais do programa servidor	4
3	Método de inserção e de análise de pedidos pendentes	7
4	Teste 1	10
5	Teste 2	11
6	Teste 3	12

1 Introdução

1.1 Apresentação do Caso de Estudo

O intuito do trabalho prático revolve do princípio em implementar um serviço que permita aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura eficiente, poupando espaço do disco. Este serviço possui, portanto, diversas funcionalidades como exemplo, compressão e cifragem dos ficheiros a serem armazenados. Assim sendo, deverá ser desenvolvido um cliente que ofereça uma interface com o utilizador via linha de comando.

O utilizador poderá agir sobre o servidor. Deverá ser também desenvolvido um servidor, mantendo em memória a informação relevante para suportar as funcionalidades implementadas.

1.2 Principais objetivos

Com a realização deste trabalho prático de Sistemas Operativos foram-nos evidenciados os seguintes objetivos a alcançar:

1. Desenvolver a capacidade de interação e trabalhar em grupo para a concretização do trabalho prático;
2. A implementação de programas cliente e servidor de uma forma completa e pertinente;
3. Obter novos conceitos e domínio de ferramentas que poderão ser úteis na cadeira como também na vida futura;
4. Promover um espírito crítico e analítico com base nos resultados obtidos e resultados esperados.

1.3 Estrutura do Relatório

O presente relatório está estruturado em 5 secções principais: **Introdução, Arquitetura, Programa Cliente, Programa Servidor e Testes.**

A **Introdução** expõe uma pequena apresentação do paradigma em que revolve este trabalho prático, as suas funcionalidades e o seu objetivo.

No caso da **Arquitetura**, descrevemos a abordagem que tivemos na implementação do que nos foi pedido. Então, como o servidor foi feito e todas as outras características que o programa tem; a comunicação entre o cliente e o servidor e as estruturas de dados que criamos de maneira a auxiliar a implementação do programa.

O **Programa Cliente** especifica a parte do código onde existe a interface a ser usada pelo utilizador para que este possa agir sobre o servidor mediante argumentos na linha de comando.

Também, o **Programa Servidor** descreve o servidor onde o utilizador age e local que possui as funcionalidades implementadas no programa de forma a tornar possível o objetivo do trabalho prático.

Em relação aos **Testes**, demonstramos a execução do programa e como este reage com base na sua utilização por um cliente.

1.4 Funcionalidades Avançadas

Todas as funcionalidades avançadas referidas no relatório foram implementadas no trabalho, esta foram as seguintes:

- Número de bytes de input/output após a execução de uma tarefa do tipo *proc-file*.
- Terminação de forma graciosa por parte do servidor aquando o sinal *SIGTERM*, deixando primeiro terminar os pedidos em processamento ou pendentes, mas rejeitando a submissão de novos pedidos.
- Diferentes prioridades (0 a 5) para diferentes tarefas do tipo *proc-file* para priorizar as tarefas com maior prioridade.

2 Arquitetura

2.1 Estrutura de dados

Para facilitar a estruturação e o acesso à informação importante para realizar as tarefas pretendidas decidimos criar uma *struct* chamada **Pedido**.

Esta *struct* possui um array com os argumentos necessários para o servidor executar a sua função, possui também um campo que tem o número de elementos do *array* em questão; aquando a aplicação das opções avançadas no programa decidimos que seria uma solução rápida criar mais um campo para a prioridade, mas nos pedidos do tipo *status* este campo acaba por ser inútil.

De salientar que o *array* com a informação na primeira posição (*request* → *args[0]*) está sempre o *pid* do processo que efetua o pedido ao servidor.

```
typedef struct pedido{
    int elems;
    int priority;
    char args[22][222]; //max de 22 palavras de de 222 de lenght de cada palavra
}*Pedido;
```

Figura 1: *Struct* Pedido criada.

2.2 Validação Input

A validação do input inserido pelo cliente é feita no ficheiro "sdstore.c". Como é passada a *struct* já com a informação devidamente validada o programa do servidor não necessita de fazer uma nova validação dos argumentos.

2.3 Comunicação Cliente-Servidor

O cliente e o servidor comunicam entre si através de *pipes* com nome, a informação que passa por esses *pipes* ou são as *structs* anteriormente descritas ou *strings*.

A criação de todos os *fifos* implica a criação de ficheiros temporários, pelo que esses ficheiros são sempre criados na pasta 'tmp', na **terminação graciosa** do programa todos os ficheiros temporários criados tanto pelo cliente como pelo servidor são eliminados.

2.4 Variáveis Globais

A utilização de variáveis globais surge com o intuito de ter informação armazenada em memória enquanto o servidor está a correr para este poder efetuar as suas funções.

Tentamos que estas fossem auto explicativas e é através delas que manipulamos toda a informação relativa ao servidor. Tivemos sempre o cuidado de só alterar o conteúdo destas variáveis apenas quando estávamos no processo do servidor e nunca em processos filhos, pois qualquer alteração das variáveis nos processos filhos seriam irrelevantes pois eram um mera cópia da informação original.

```
char* transformacoesNome[]={
    "nop",
    "bcompress",
    "bdecompress",
    "gcompress",
    "gdecompress",
    "encrypt",
    "decrypt",
    NULL
};

char* transformations_folder;

int config[TRANS_NR]; //nr maximos de cada tipo de filtros que podem executar ao mesmo tempo
int using[TRANS_NR]; //nr de filtros de cada tipo que estao a executar no momento presente

int pendingRequestsIdx[100][2]; //pedidos em espera          idTask | fifoU
int nrpendingRequests; //numero de pedidos em espera

int runningRequestsIdx[100]; //pedidos a correr
int nrrunningRequests; //numero de pedidos a correr

int waiting[100][2]; // pid | idxTask
int nrwaitingProcess;

Pedido tasks[MAX_STORE]; //todos os pedidos enviados para o servidor
int tasksNr;

pid_t pidServidor;
int backbone;
int alarm;

int accStarvation;
```

Figura 2: Variáveis globais do programa servidor

3 Programa Cliente

3.1 Validação dos argumentos

O programa do cliente("sdstore") começa por fazer uma verificação do input introduzido pelo utilizador segundo os seguintes critérios:

- | | | |
|-----------------------------|----|---|
| ▪ Número de argumentos == 2 | | ▪ Número de argumentos >= 6 |
| ▪ argv[1] == "status" | ou | ▪ argv[1] == "proc-file" |
| | | ▪ $0 \leq \text{atoi}(\text{argv}[2]) \leq 5$
(Prioridade) |

3.2 Comunicação por fifos

Após a verificação o programa começa por criar um fifo que será único para o processo em questão, o nome do fifo criado obedece à sintaxe 'fifoReadXX' onde 'XX' representa o pid do processo. Por exemplo um processo que esteja a executar o programa do cliente com um **pid** de valor **12345** irá criar um fifo com o nome '**fifoRead12345**'.

De seguida o cliente abre o fifo do servidor que tem um nome fixo ('fifoWrite') em modo de escrita e após povoar a struct com a informação necessária envia a struct através do *file descriptor* aberto anteriormente.

Após enviar o pedido o programa abre o fifo antes criado pelo mesmo em modo de leitura e fica à escuta através desse *file descriptor* até receber *End Of File (EOF)* e vai imprimindo para o *standard output* toda a informação que lê desse *fifo*. Para finalizar de forma graciosa, quando recebe *EOF*, elimina o *fifo* criado anteriormente.

4 Programa Servidor

4.1 Execução

O programa do servidor(*"sdstored"*) recebe 3 argumentos, o executável do servidor, o caminho para o ficheiro de configuração e o caminho da pasta onde estão as transformações. Assim, lê o ficheiro das configurações (existe um *"config.txt"* na pasta *'etc'*) e começamos por criar o *fifo* de escrita *"fifoWrite"*, para o qual todos os clientes escrevem.

Depois, abre o *fifo* em modo de leitura à espera que os clientes escrevam, ficando este bloqueado até que um cliente abra a outra ponta do *fifo* em modo de escrita. Quando é aberto o *fifo* no cliente, o servidor abre um *backbone*(file descriptor de backup) do mesmo *fifo* em modo escrita.

De seguida, cria o ciclo do alarme que está constantemente a enviar o sinal *SIGUSR1* ao processo do servidor. Dá-se início, então, ao ciclo de leitura de pedidos, onde através *file descriptor* do servidor este lê uma *struct* de cada vez. De referir, que a leitura do *"fifoWrite"* no ciclo *while* nunca fecha, porque o *backbone* se encontra aberto.

O servidor lê uma *struct* enviada pelo cliente e abre o *fifo* criado pelo servidor(*READ_NAME+pid-Cliente*). O servidor verifica se essa *struct* tem 2 elementos. Se tiver, significa que é feito um pedido de *status*, executa-o e o *fifo* imprime a informação do estado atual do servidor. Depois, fecha o descritor do cliente associado. Quando termina esta ação volta ao ciclo *while* de leitura de pedidos.

Caso não a *struct* que recebeu não tenha 2 argumentos, significa que se trata de um pedido de transformação de ficheiros e adiciona essa *struct* ao *array* onde todos os pedidos já recebidos estão registados.

De seguida, verifica se o pedido pode executar nesse momento, através da função *"goPendingOrNot"*(que devolve -1(ERRO) caso não possa fazer). Nesse caso, isto é, se ficar pendente, adiciona-o à fila dos pedidos pendentes, através da função *"addPending"*(situação das prioridades referida mais à frente neste relatório) onde esta função coloca o pedido numa dada posição consoante a sua prioridade. No caso do pedido de transformação de ficheiro possa ocorrer no momento o servidor adiciona a essa tarefa às tarefas que estão a executar, incrementando os filtros em uso e executando um duplo *fork*.

O 2º *fork*, que faz espera ativa, avisa o cliente que o seu pedido já começou, redireciona o input/output e aplica as transformações devidas. Já o 1º *fork* não realiza espera ativa, para o servidor não entrar em pausa.

Quando o 2º *fork* termina, vai buscar as *stats* dos ficheiros onde aplicou os filtros e envia para o cliente a informação necessária, notificando também o servidor que já vai passar a ter filtros livres(*SIGUSR1*). Ainda no 1º *fork*, devido à *flag WNOHANG* na função *'waitpid'*, sabemos que se o resultado for 0, é porque o 2º *fork* ainda não tinha terminado e, então faz *"addWaiting"* do pedido, ou seja não se liberta os filtros. É então necessário esperar que o *pid* desse *fork* retorne

alguma coisa útil. Caso contrário, se tiver executado o pid do processo filho que acabou (diferente de 0), faz "removeRunning", onde liberta espaço nos filtros. Acaba então a manutenção de um pedido, o servidor pode fechar o descritor do cliente pois caso ainda seja necessário utilizar o *fifo*, o processo filho possui uma cópia do descritor e pode usá-la, envia também o sinal **SIGUSR1** para o servidor e regressa ao ciclo *while* de leitura de pedidos dos clientes. Quando sai desse ciclo na forma "GRACIOSA", fecha o *fifo* de leitura e apaga-o o *fifo*.

4.2 Prioridades

Numa versão mais anterior onde não tratávamos das prioridades das tarefas, o algoritmo de escalonamento dos processos pendentes era o *First In First Out (FIFO)*. À medida que os processos ficavam pendentes eram colocados no fim de um *array* das tarefas pendentes e esse *array* era lido da esquerda para a direita.

Quando decidimos implementar esta funcionalidade avançada a única diferença na implementação foi a posição onde o pedido é inserido na *array* das tarefas pendentes e a ordem pelo que os pedidos são testados.

Os pedidos começam por ser verificados se já podem executar do fim para o início do *array* dos processos dependentes e um pedido com uma certa prioridade é colocado depois dos pedidos com prioridades inferiores mas antes dos pedidos com prioridade igual ou superior ao processo em questão.

Na figura abaixo apresentamos um esquema desta solução de inserção e testagem das tarefas no *array*.

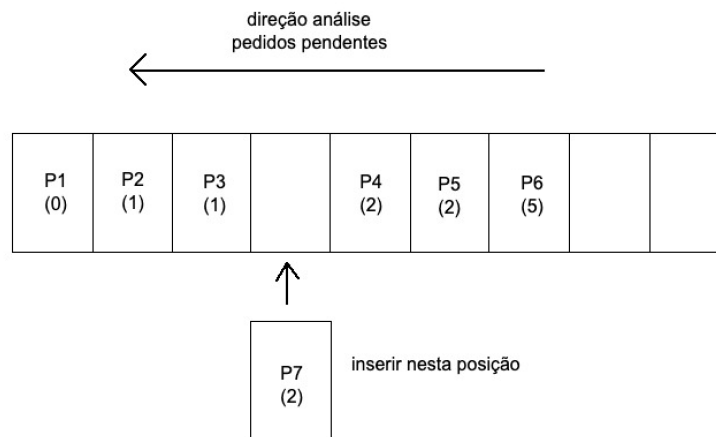


Figura 3: Método de inserção e de análise de pedidos pendentes

Esta solução poderia levar à *Starvation*, pelo que para prevenir esta situação a cada 11 vezes que o programa trate o sinal *SIGUSR1* incrementa a prioridade de todos os pedidos pendentes em 1 unidade até um valor máximo de 5.

O único problema que pode dar-se é se um pedido pendente necessitar de utilizar vários filtros, mas outros pedidos pendentes necessitem de um número razoavelmente menor, assim, primeiramente era sempre testado o processo mais antigo(maior prioridade), mas como nunca haveria a quantidade de filtros total necessária, os outros pedidos com menos prioridade/mais recentes como necessitam de menos filtros conseguem começar a executar.

Uma forma de evitar esta situação seria por exemplo de X em X tempo não aceitar mais pedidos até que a fila de pedidos pendentes estivesse vazia.

4.3 Sinais

Neste programa são utilizados 3 tipos de sinais, o **SIGALRM**, o **SIGTERM** e o **SIGUSR1**. Este último faz com que o servidor tente libertar filtros e colocar processos pendentes em execução. Os outros dois sinais serão abordados de seguida.

4.4 Alarme

O servidor necessita de efetuar regularmente verificações para saber se já pode executar algum pedido que esteja pendente, pela finalização de outras execuções, leva a que a razão da utilização de alguns filtros diminua e certas tarefas podem passar de pendentes para a executar.

Para tal criamos um *fork* que irá infinitamente enviar o sinal **SIGUSR1** para o *pid* do processo onde o processo do servidor está a correr.

Apesar de no fim de cada execução de uma tarefa o sinal *SIGUSR1* seja enviado para o processo do servidor, a fila de sinais apenas suporta um sinal de cada tipo de cada vez, o que pode levar a que certos sinais sejam ignorados.

Devido a este facto, é necessário assegurar que constantemente os processos são reavaliados para garantir que nenhum fique esquecido.

4.5 Finalização Graciosa

Um sinal é utilizado para notificar um processo de um evento síncrono ou assíncrono. Quando um sinal é enviado, o sistema operativo interrompe o fluxo normal de execução do processo de destino para entregar o sinal. O sinal *SIGTERM* é um sinal genérico usado para causar o término do programa. O comando *shell kill* gera *SIGTERM* por predefinição.

Para garantir a finalização graciosa, sabemos que se receber o sinal *SIGTERM*, o servidor deve deixar primeiro terminar os pedidos em processamento, notificar os processos pendentes que o seu pedido foi abortado e rejeitar a submissão de novos pedidos.

De seguida apresentamos as várias partes da nossa solução para a terminação graciosa do servidor.

Parar a rotina do alarme, enviando o sinal *SIGKILL* para o processo que está a executar o ciclo dos alarmes.

```
kill(alrm, SIGKILL);
```

Fechar o *file descriptor* criado no início do programa que após fechado faz com que o ciclo *while* que estava a ler do '*fifoWrite*' saia do ciclo pois ao tentar ler dá EOF.

```
close(backbone);
```

Informar todos os processos pendentes que foram abortados(o seu pedido não irá ser realizado) e fechar a conexão com esses clientes.

```
while (nrpendingRequests>0){
    int fifoU = pendingRequestsIdx[0][1];
    write(fifoU, SERVICE_ABORTED, strlen(SERVICE_ABORTED));
    close(fifoU);
    removePending(0);
}
```

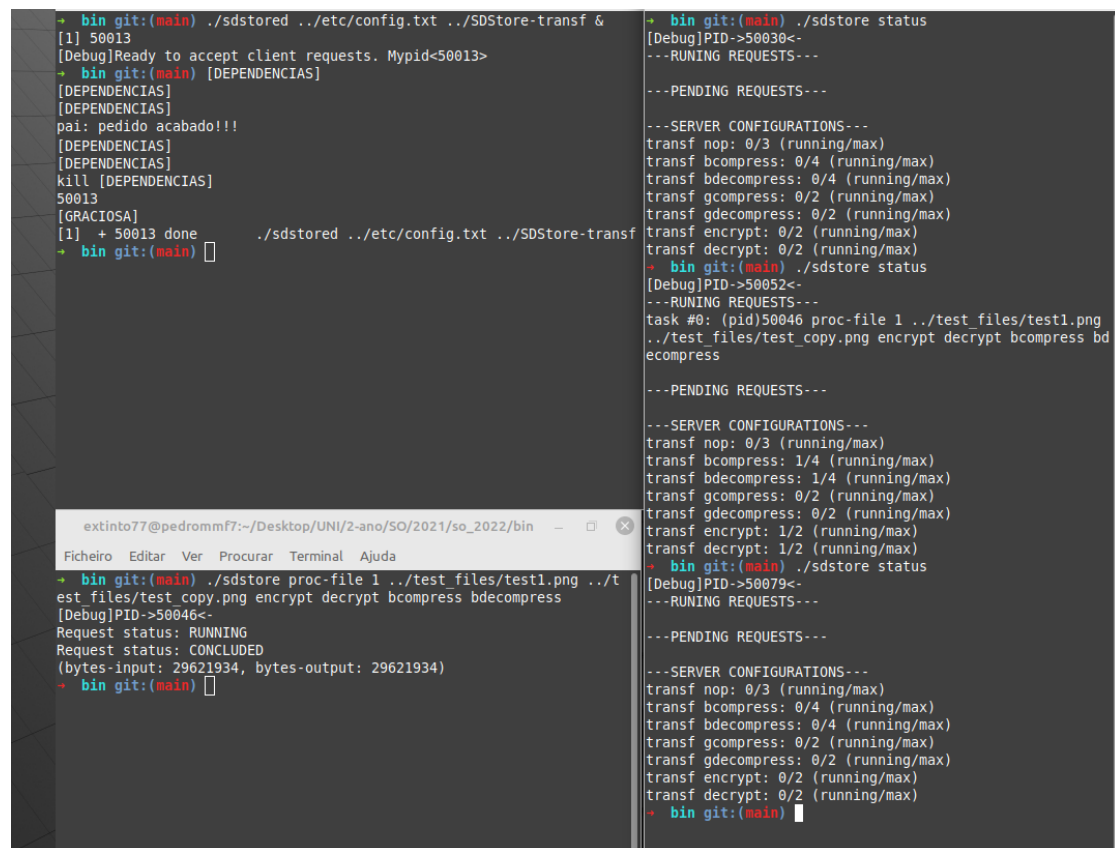
Esperar que todos os processos que estão a correr acabem de executar.

```
while (nrrunningRequests>0 || nrwaitingProcess>0){
    int status;
    for (int i = 0; i < nrwaitingProcess; i++){
        int pid = waiting[i][0];
        if(waitpid(pid, &status, WNOHANG)!=0){
            int idxTask = waiting[i][1];
            removeWaiting(pid);
            removeRunning(idxTask);
            i--;
        }
    }
}
```

5 Testes

Na realização dos testes tentamos abordar as várias funcionalidades do trabalho, de seguida iremos apresentar algumas mas estaremos disponíveis para realizar mais alguns testes mais específicos na defesa do trabalho prático.

Neste primeiro teste temos a janela dos clientes à direita e à esquerda em baixo, a janela na esquerda em cima é onde o programa do servidor está a correr. Podemos observar o estado de ocupação antes durante e depois de um pedido do tipo *proc-file*. Podemos também encontrar o fecho do servidor de forma graciosa(onde nada aconteceu).



```

+ bin git:(main) ./sdstored ../etc/config.txt ../SDStore-transf &
[1] 50013
[Debug]Ready to accept client requests. Mypid<50013>
+ bin git:(main) [DEPENDENCIAS]
[DEPENDENCIAS]
[DEPENDENCIAS]
pai: pedido acabado!!!
[DEPENDENCIAS]
[DEPENDENCIAS]
kill [DEPENDENCIAS]
50013
[GRACIOSA]
[1] + 50013 done      ./sdstored ../etc/config.txt ../SDStore-transf
+ bin git:(main)

+ bin git:(main) ./sdstore status
[Debug]PID->50030<-
---RUNING REQUESTS---
---PENDING REQUESTS---
---SERVER CONFIGURATIONS---
transf nop: 0/3 (running/max)
transf bcompress: 0/4 (running/max)
transf bdecompress: 0/4 (running/max)
transf gcompress: 0/2 (running/max)
transf gdecompress: 0/2 (running/max)
transf encrypt: 0/2 (running/max)
transf decrypt: 0/2 (running/max)
+ bin git:(main) ./sdstore status
[Debug]PID->50052<-
---RUNING REQUESTS---
task #0: (pid)50046 proc-file 1 ../test_files/test1.png
../test_files/test_copy.png encrypt decrypt bcompress bd
ecompress
---PENDING REQUESTS---
---SERVER CONFIGURATIONS---
transf nop: 0/3 (running/max)
transf bcompress: 1/4 (running/max)
transf bdecompress: 1/4 (running/max)
transf gcompress: 0/2 (running/max)
transf gdecompress: 0/2 (running/max)
transf encrypt: 1/2 (running/max)
transf decrypt: 1/2 (running/max)
+ bin git:(main) ./sdstore status
[Debug]PID->50079<-
---RUNING REQUESTS---
---PENDING REQUESTS---
---SERVER CONFIGURATIONS---
transf nop: 0/3 (running/max)
transf bcompress: 0/4 (running/max)
transf bdecompress: 0/4 (running/max)
transf gcompress: 0/2 (running/max)
transf gdecompress: 0/2 (running/max)
transf encrypt: 0/2 (running/max)
transf decrypt: 0/2 (running/max)
+ bin git:(main)

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore proc-file 1 ../test_files/test1.png ../t
est_files/test_copy.png encrypt decrypt bcompress bdecompress
[Debug]PID->50046<-
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 29621934)
+ bin git:(main)

```

Figura 4: Teste 1

Neste segundo teste podemos ver de forma semelhante à anterior o servidor no canto superior esquerdo, tudo o resto são clientes. É possível ver que houve clientes que executaram sem problemas, clientes que executaram mal enviaram o pedido e também clientes que tiveram o seu pedido pendente mas depois acabou por ser efetuado. Podemos também ver que no encerramento do servidor de forma graciosa o cliente no canto inferior esquerdo viu o seu pedido a ser abortado pois aquando o envio do do sinal SIGTERM, o seu pedido ainda não tinha iniciado.

```

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstored ../etc/config.txt ../SDStore-transf &
[1] 54496
[Debug]Ready to accept client requests. Mypid<54496>
+ bin git:(main) [DEPENDENCIAS]
[DEPENDENCIAS]
pai: pedido acabado!!!
[DEPENDENCIAS]
pai: pedido acabado!!!
[DEPENDENCIAS]
pai: pedido acabado!!!
[DEPENDENCIAS]
kill[DEPENDENCIAS]
[DEPENDENCIAS]
kill 54496
[GRACIOSA]
[1] + 54496 done      ./sdstored ../etc/config.txt ../SDStore-transf
+ bin git:(main)

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore proc-file 1 ../test_files/test1.png ../test_files/test_copy.png encrypt decrypt encrypt decrypt
[Debug]PID->54536<-
Request status: PENDENT
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 29621934)
+ bin git:(main)

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore proc-file 1 ../test_files/test1.png ../test_files/test_copy.png encrypt encrypt encrypt encrypt
[Debug]PID->54512<-
Request status: PENDENT
Request status: ABORTED
+ bin git:(main)

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore status
[Debug]PID->54515<-
---RUNING REQUESTS---
---PENDING REQUESTS---
pending #0: (pid)54512 proc-file 1 ../test_files/test1.png
../test_files/test_copy.png encrypt encrypt encrypt encrypt
---SERVER CONFIGURATIONS---
transf nop: 0/3 (running/max)
transf bcompress: 0/4 (running/max)
transf bdecompress: 0/4 (running/max)
transf gcompress: 0/2 (running/max)
transf gdecompress: 0/2 (running/max)
transf encrypt: 0/2 (running/max)
transf decrypt: 0/2 (running/max)
+ bin git:(main) ./sdstore status

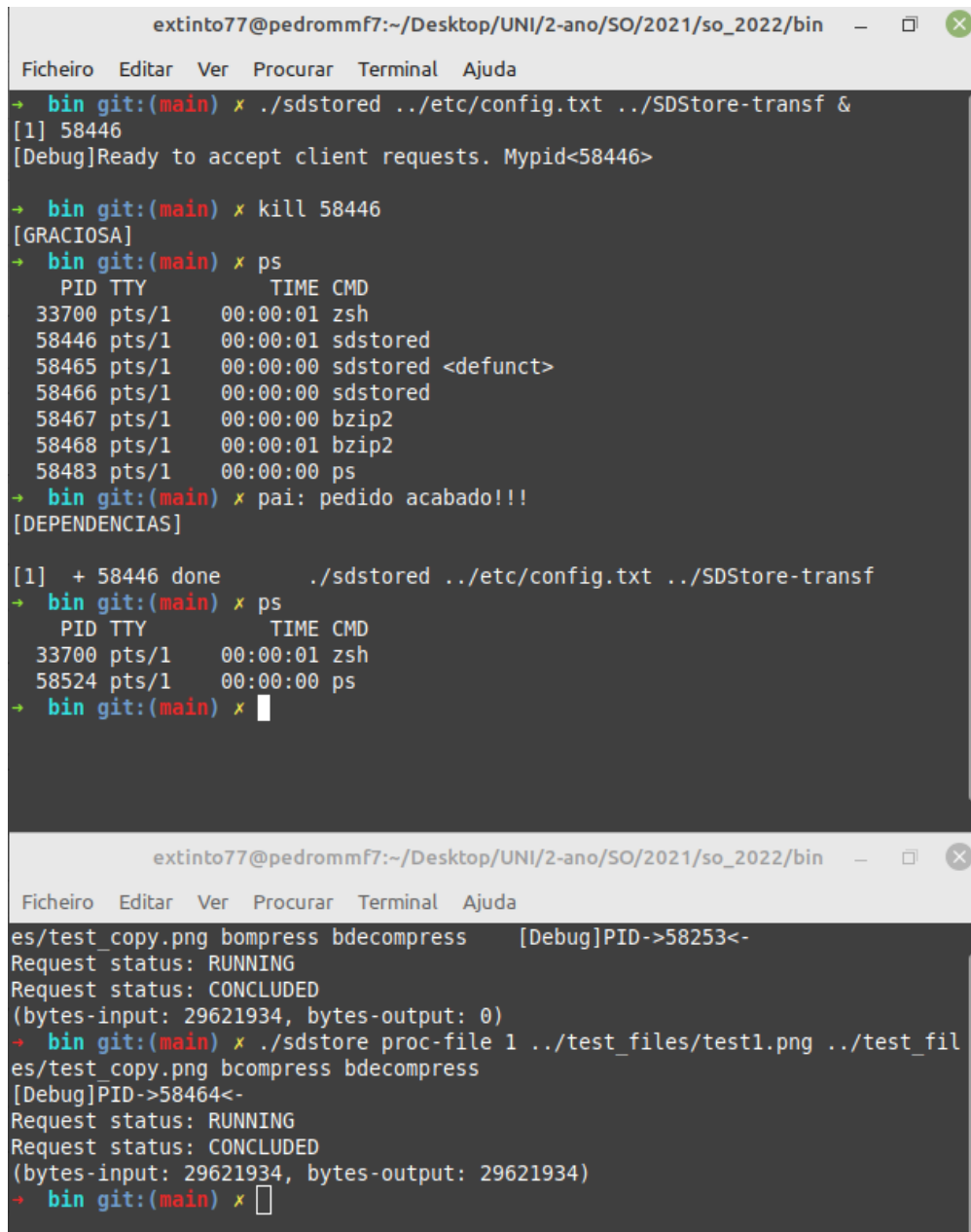
extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore proc-file 5 ../test_files/test1.png ../test_files/test_copy1.png encrypt decrypt encrypt decrypt
[Debug]PID->54564<-
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 29621934)
+ bin git:(main)

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
+ bin git:(main) ./sdstore proc-file 1 ../test_files/test1.png ../test_files/test_copy.png encrypt encrypt encrypt encrypt
[Debug]PID->54529<-
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 29621934)
+ bin git:(main)

```

Figura 5: Teste 2

Por último podemos ver que não existem processos *zombies* esquecidos e que o servidor espera pelo fim de todos os seus processos filhos para terminar o seu processo.



```
extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
→ bin git:(main) x ./sdstored ../etc/config.txt ../SDStore-transf &
[1] 58446
[Debug]Ready to accept client requests. Mypid<58446>

→ bin git:(main) x kill 58446
[GRACIOSA]
→ bin git:(main) x ps
  PID TTY          TIME CMD
 33700 pts/1        00:00:01 zsh
 58446 pts/1        00:00:01 sdstored
 58465 pts/1        00:00:00 sdstored <defunct>
 58466 pts/1        00:00:00 sdstored
 58467 pts/1        00:00:00 bzip2
 58468 pts/1        00:00:01 bzip2
 58483 pts/1        00:00:00 ps
→ bin git:(main) x pai: pedido acabado!!!
[DEPENDENCIAS]

[1] + 58446 done      ./sdstored ../etc/config.txt ../SDStore-transf
→ bin git:(main) x ps
  PID TTY          TIME CMD
 33700 pts/1        00:00:01 zsh
 58524 pts/1        00:00:00 ps
→ bin git:(main) x

extinto77@pedrommf7:~/Desktop/UNI/2-ano/SO/2021/so_2022/bin
Ficheiro Editar Ver Procurar Terminal Ajuda
es/test_copy.png bcompress bdecompress [Debug]PID->58253<-
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 0)
→ bin git:(main) x ./sdstore proc-file 1 ../test_files/test1.png ../test_fil
es/test_copy.png bcompress bdecompress
[Debug]PID->58464<-
Request status: RUNNING
Request status: CONCLUDED
(bytes-input: 29621934, bytes-output: 29621934)
→ bin git:(main) x
```

Figura 6: Teste 3

6 Conclusão

O tempo investido neste trabalho prático ajudou-nos a tocar na superfície do que é Sistemas Operativos e, deste modo, o grupo passou a ser capaz a manipular ficheiros quer seja na sua compressão ou descompressão, quer seja na sua cifragem ou decifragem, a criar um programa cliente servidor onde o cliente pode agir sobre o servidor através de argumentos especificados na linha de comando.

Além disso, também permitiu um melhor aprofundamento no manuseamento da linguagem em que o programa foi desenvolvido, ou seja, em C.

Terminado este trabalho, e tendo em conta todos estes aspetos, consideramos que realizamos um projeto que satisfaz o que foi pedido no enunciado, que coincide com a matéria lecionada nas aulas teóricas e práticas.

Do ponto de vista apreciativo, pensamos que fomos bem-sucedidos durante o desenrolar deste projeto e que adquirimos capacidades fundamentais não só para o nosso futuro académico como profissional.