

Universidade do Minho
Escola de Engenharia

Relatório do Trabalho Prático

Sistemas Operativos

2021/2022

Grupo 1

José António Alves de Matos Moreira, A95522

Santiago Vicente Ferreira Fernandim Domingues, A96886

Gonçalo Lobo Freitas, A96136

Conteúdo

1.Introdução	3
1.1 Descrição do Serviço	3
1.1.1 <i>SDSTORED</i>	3
1.1.2 <i>SDSTORE</i>	3
1.2 Transformações	3
2. Solução	3
3. Implementação	4
3.1 Estruturas de Dados	4
3.1.1 Estrutura Line	4
3.2.2 Estrutura Operation	5
3.2 Parsing	5
3.3 Processamento dos ficheiros	6
3.4 Processamento concorrente de pedidos	6
3.5 Bytes dos ficheiros	6
3.6 Status	7
3.7 <i>Sigterm</i>	7
4. Problemas Estruturais	7
4.1 <i>Queue</i> de pedidos	8
4.2 Status incompleto	8
4.3 Estado pendente	8
4.4 <i>Sigterm</i>	8
5. Conclusão	8

1.Introdução

O presente documento serve como base ao trabalho prático realizado no âmbito da unidade curricular de Sistemas Operativos, no ano letivo 2021/2022, cujo objetivo passou por desenvolver um serviço que permitisse aos utilizadores armazenar uma cópia dos seus ficheiros, bem como implementar um conjunto de transformações de compressão e encriptação para uso do utilizador.

1.1 Descrição do Serviço

De modo a ser possível a projeção do serviço foi necessário a criação de programas que fornecessem uma interface com o utilizador e que fossem capazes de realizar as operações pretendidas. Para isso foram criados dois programas distintos.

1.1.1 *SDSTORED*

Servidor. O núcleo do projeto, capaz de realizar diversas transformações, de criar e armazenar ficheiros a mando do utilizador.

1.1.2 *SDSTORE*

Cliente. Responsável por oferecer uma interface para o utilizador conseguir comunicar com o servidor.

1.2 Transformações

O cliente dispõe das seguintes transformações:

- `bcompress / bdecompress`: comprime / descomprime dados com o formato bzip;
- `gcompress / gdecompress`: comprime / descomprime dados com o formato gzip;
- `encrypt / decrypt`: cifra / decifra dados;
- `nop`: copia dados sem realizar qualquer transformação;

2. Solução

Como forma de darmos resposta ao enunciado do trabalho começamos por criar (em linguagem C) dois programas distintos cuja comunicação entre si seria assegurada através de *named pipes*. No programa do servidor foram criadas estruturas de dados capazes de armazenar as linhas de comando proveniente do input do utilizador contendo os ficheiros de origem e destino tal como as transformações a serem realizadas. Para tal foi criada uma função responsável por fazer o *parse* dos dados a ser armazenados na respetiva estrutura. É

também no servidor que se encontram as funções responsáveis pelo processamento dos ficheiros e do status, responsável por indicar ao utilizador o estado do servidor.

Visto que a maioria das operações é realizada no servidor foi necessário estabelecer uma conexão entre os programas de modo a ser possível o *parse* correto dos dados, para isso foi criado no servidor um *named pipe*, que será aberto em modo de escrita pelo cliente e em modo de leitura pelo servidor, deste modo conseguimos garantir que o servidor consegue armazenar corretamente os dados provenientes do pedido do utilizador. No entanto, para garantir que é possível o utilizador realizar vários pedidos, quer concorrentemente quer separadamente, foi necessário estabelecer um *named pipe* responsável por cada pedido do cliente, sendo estes criados no programa do cliente e formatados com o nome “proc” seguido do pid do processo em questão (obtido através de uma chamada à função `getpid()`).

3. Implementação

3.1 Estruturas de Dados

Estruturas de dados no programa servidor:

```
typedef struct line {
    char* operation;
    int limit;
    int using;
} *Line;

typedef struct operation {
    char* fifo;
    char* source;
    char* destination;
    char** operations;
    int opN;
} *Operation;
```

Figura 1- Estruturas de Dados

3.1.1 Estrutura Line

Nesta estrutura armazenamos o input do servidor. A variável (`char*`) *operation* é responsável por armazenar a operação do respetivo ficheiro de operações, a variável (`int`) *limit* armazena o limite máximo da respetiva transformação, ou seja, o número de vezes que o servidor permite realizar aquela transformação e a variável (`int`) *using* representa a quantidade de transformações daquele tipo que está a ser executada no momento pelo servidor.

3.2.2 Estrutura Operation

Nesta estrutura armazenamos o input do cliente. A variável (char*) *fifo* é responsável por armazenar o pipe (processo) do cliente que está a realizar o pedido, a variável (char*) *source* armazena o ficheiro que vai ser processado, isto é, o ficheiro de origem, a variável (char*) *destination* armazena o ficheiro destino, ou seja, o ficheiro resultante da transformação do ficheiro original, uma estrutura (char**) *operations*, um array responsável pelo armazenamento do conjunto de operações que o utilizador pretende realizar, de acordo com o limite máximo estabelecido e um inteiro *opN* que indica o número de operações a ser realizadas

3.2 Parsing

Tendo já definido um conjunto de estruturas de dados responsáveis pelo armazenamento do input do cliente e do servidor foi então necessária a criação de uma função que fosse capaz de dividir os respetivos inputs, alocando-os nos espaços corretos da estrutura de dados.

```
Operation toOperation(char* file){
    Operation ret = (Operation)malloc(sizeof(*ret));
    ret->fifo = strdup(strsep(&file, " "));
    ret->source = strdup(strsep(&file, " "));
    ret->destination = strdup(strsep(&file, " "));
    ret->operations = malloc(256);
    int index = 0;
    while(file != NULL){
        ret->operations[index++] = strdup(strsep(&file, " "));
    }
    ret->opN = index;
    return ret;
}

Line toLine(char* line){
    Line ret = (Line)malloc(sizeof(*ret));
    ret->operation = strdup(strsep(&line, " "));
    ret->limit = atoi(strdup(strsep(&line, " ")));
    ret->using = 0;

    return ret;
}
```

Figura 2- Funções de Parsing

Exemplo input do utilizador no programa cliente:

```
$ ./sdstore proc-file repos.csv final1 nop encrypt
```

Figura 3- Input do utilizador

Exemplo input no programa servidor:

```
$ ./sdstored config/config.txt execs/
```

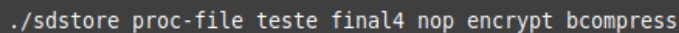
Figura 4- Input do programa servidor

3.3 Processamento dos ficheiros

Nesta fase, após a correta alocação dos ficheiros e respetivas transformações nas estruturas de dados adequadas, implementamos as funções base do projeto, responsáveis pelo processamento dos ficheiros.

De forma a sustentar a multiplicidade de transformações em cada pedido, foi usada uma técnica de encadeamento de *pipes* para a comunicação entre processos pai e filho, de modo a ser possível a execução das transformações de modo coerente, evitando simultaneamente a criação de ficheiros intermédios, um dos requisitos do trabalho.

Foram aplicadas técnicas de redirecionamento de descritores, com o intuito de utilizar de forma correta as chamadas ao sistema para a execução das transformações, redirecionando a execução do *standard output* para os *pipes*, de modo a ser possível o seu encadeamento.



```
./sdstore proc-file teste final4 nop encrypt bcompress
```

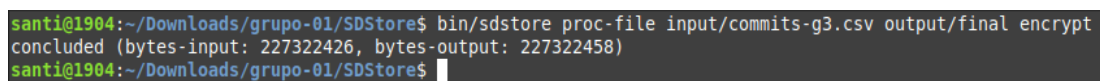
Figura 5- Encadeamento de transformações

3.4 Processamento concorrente de pedidos

Com o objetivo de respeitar a concorrência de pedidos a execução de cada um é feita em processos filho distintos. O processo do servidor consegue manter a execução simultânea de pedidos devido ao facto dos pedidos serem executados em processos filho. Com isto o processo pai não necessita de esperar a saída de cada filho, sendo possível executar outro pedido.

3.5 Bytes dos ficheiros

No final de cada pedido é impresso o estado do pedido como concluído e o número de bytes do ficheiro de origem e do ficheiro de destino após as transformações. Para tal foi criada uma função que executa num processo filho os comandos “wc” “-c” no ficheiro de origem, possível através do redirecionamento do standard output para o descritor do ficheiro. Sendo posteriormente lido pelo processo pai e retornando o número de bytes. A comunicação entre os processos é assegurada por um *pipe*, cujo extremo de escrita é usado pelo processo filho e o extremo de leitura é usado pelo processo pai.



```
santi@1904:~/Downloads/grupo-01/SDStores$ bin/sdstore proc-file input/commits-g3.csv output/final encrypt
concluded (bytes-input: 227322426, bytes-output: 227322458)
santi@1904:~/Downloads/grupo-01/SDStores$
```

Figura 6- Bytes dos ficheiros de origem e destino

3.6 Status

A qualquer instante o utilizador deve ser capaz de utilizar o comando status para averiguar o funcionamento do servidor, contendo todos os processos em execução bem como o número de vezes que uma transformação está a ser executada e o seu limite máximo.

Devido a algumas inconsistências na implementação do nosso projeto não foi possível executar o status na sua plenitude, sendo apenas possível ao utilizador verificar o número de transformações que foram usadas desde a inicialização do servidor. Inconsistências essas que serão discutidas posteriormente neste relatório.

```
nop - 0 times
bcompress - 0 times
bdecompress - 0 times
gcompress - 0 times
gdecompress - 0 times
encrypt - 0 times
decrypt - 0 times
```

Figura 7- Status inicial

```
nop - 1 times
bcompress - 1 times
bdecompress - 0 times
gcompress - 0 times
gdecompress - 0 times
encrypt - 1 times
decrypt - 0 times
```

Figura 8- Status após execução

3.7 Sigterm

O servidor é terminado caso ocorra algum erro durante a sua execução, como por exemplo na abertura de um *named pipe* e também através do comando “sigterm”, input do utilizador na linha de comandos.

```
santi@1904:~/Downloads/grupo-01/SDStore$ bin/sdstore sigterm
```

Figura 9- Comando sigterm

```
santi@1904:~/Downloads/grupo-01/SDStore$ bin/sdstore config/configs.txt bin/SDStore-transformations/
Terminated
santi@1904:~/Downloads/grupo-01/SDStore$
```

Figura 10- Terminação do servidor após sigterm

4. Problemas Estruturais

Apesar de termos desenvolvido um programa funcional é impossível omitir determinados erros que limitam de certa forma a resposta do programa a alguns dos requisitos necessários para este trabalho.

4.1 Queue de pedidos

Um dos problemas do projeto insere-se na ausência de um método que assegure o armazenamento consecutivos dos pedidos numa *queue*. Esta foi a técnica pensada pelo grupo cuja implementação não foi de todo exequível. O principal objetivo deste método passaria por alocar os pedidos num buffer antes de serem executados, pois seria acrescentada uma condição que verificaria o estado de utilização de determinada transformação e a partir daí determinava se era possível ou não realizar a transformação. A implementação deste método era também fundamental para a execução do status.

4.2 Status incompleto

A não implementação do método em questão desencadeou alguns problemas na execução de alguns requisitos do programa. O status do programa apenas permite ao utilizador visualizar o número total de transformações efetuadas até um determinado momento, durante o funcionamento do servidor, não sendo possível averiguar quais os pedidos em curso nem as transformações a serem executadas no momento.

4.3 Estado pendente

Quando um pedido não pudesse ser executado devido ao limite máximo de determinada transformação ter sido atingido, deveria ser impresso como output no cliente o estado “pending”, como forma de indicar ao cliente que o seu processo não pode ser executado instantaneamente e por esse motivo ficará em espera. A ausência de filas de espera de pedidos impossibilita esta opção.

4.4 Sigterm

Sempre que o utilizador decide usar o comando “sigterm” o servidor é terminado, no entanto os processos que ainda estejam a ser executados também o são, pelo que não os deixa terminar.

5. Conclusão

Face à proposta de trabalho apresentada o grupo projetou métodos coerentes para o desenvolvimento do programa, sendo capaz de ultrapassar a maioria das adversidades. Posto isto, conseguimos até um certo ponto desenvolver um serviço que permite ao utilizador realizar diversas operações de encriptação, compressão, relativas ações opostas e de cópia sobre ficheiros, sendo também capaz de armazenar cópias dos mesmos.