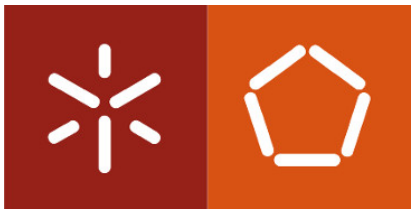

Universidade do Minho

Mestrado Integrado em Engenharia Informática
Licenciatura em Engenharia Informática

Sistemas Operativos - Trabalho Prático
SDStore: Armazenamento Eficiente e Seguro de Ficheiros
Ano Letivo 2021/2022
Grupo 69

Simão Cunha (a93262) João Loureiro (a97257)
Rafael Correia (a94870)

29 de maio de 2022

Resumo

Este relatório vem relatar a experiência na criação da aplicação `SDStore: Armazenamento Eficiente e Seguro de Ficheiros`, no âmbito do trabalho prático da UC de Sistemas Operativos.

Numa primeira instância, iremos efetuar uma pequena introdução do trabalho efetuado. De seguida, iremos descrever brevemente cada um dos módulos criados, assim como a respetiva API. Depois iremos explicar a comunicação entre o cliente e o servidor e, mais tarde, as funcionalidades básicas e eventuais funcionalidades extras. Terminaremos com uma breve conclusão acerca do trabalho efetuado, apontando possíveis erros identificados e trabalho futuro que poderá ser efetuado neste trabalho prático.

Conteúdo

1 Introdução

Este relatório surge no âmbito do trabalho prático da UC de Sistemas Operativos, onde o propósito principal do mesmo é desenvolver um serviço de armazenamento seguro e eficiente de ficheiros, através das programas de compressão como o `bcompress`, `bdecompress`, `gcompress` e `gdecompress` e através de programas de cifragem, tais como `encrypt` e `decrypt`.

O serviço terá de permitir que o cliente possa submeter os pedidos de tratamento de ficheiros e que também possa consultar o estado do servidor (tarefas a serem executadas de momento) e algumas estatisticas sobre estas.

Ao longo deste relatório, iremos efetuar uma descrição dos diversos módulos implementados na linguagem C, explicitar através de um esquema o modo de comunicação entre os clientes e o servidor, explicar as várias funcionalidades implementadas (básicas e EXTRAS), terminando com uma secção de conclusões, onde explicaremos as dificuldades que encontramos ao longo do relatório e eventual trabalho futuro que possa ser efetuado neste projeto.

2 Descrição dos módulos criados

2.1 `sdstore.c`

Este módulo refere-se à entidade cliente. Este apenas possui uma função `main` que efetua todo o trabalho. De seguida, segue um pequeno excerto em pseudo-código que exemplifica o que o cliente faz:

```
int main(int argc, char *argv[]){
    //cliente pode fazer status ou proc-file.
    //Deve ter atenção ao nº de argumentos que cada uma requer
    validacao_comandos();

    if(strcmp(argv[1], "status") == 0){
        cria_mensagem_para_enviar_para_servidor(); //aspeto: "[PID] S"

        //util para servidor saber dar a resposta ao cliente correto
        criacao_pipe(pid_cliente);

        abertura_fifo_servidor(); //comunicação cliente-servidor

        envio_mensagem_criada_para_servidor();

        fechar_fifo_servidor();

        //Cliente irá receber a resposta do servidor
        abertura_fifo_pid_cliente();

        escrever_resposta_no_stdout();

        fechar_fifo_pid_cliente(); //Cliente já recebeu toda a resposta

        apaga_fifo_pid_cliente(); //unlink do fifo em questão
    }
    else if(strcmp(argv[1], "proc-file") == 0){
        //aspeto: "[PID] P [input] [output] [binário 1] [binário 2] [...]"
        cria_mensagem_para_enviar_para_servidor();

```

```
        //util para servidor saber dar a resposta ao cliente correto
        criacao_pipe(pid_cliente);

        abertura_fifo_servidor(); //comunicação cliente-servidor

        envio_mensagem_criada_para_servidor();

        fechar_fifo_servidor();

        //Cliente irá receber a resposta do servidor
        abertura_fifo_pid_cliente();

        escrever_resposta_no_stdout();

        fechar_fifo_pid_cliente(); //Cliente já recebeu toda a resposta

        apaga_fifo_pid_cliente(); //unlink do fifo em questão
    }
    else
        reconhece_comando_desconhecido();
}
```

2.2 sdstored.c

Este módulo refere-se à entidade servidor. Este possui algumas funções auxiliares como:

```
//Inicializa o servidor, carregando o ficheiro de configuração e
//consequentemente a criação da lista ligada CONFIG
CONFIG start_server(char* config_filename, char* binarys_folder);

//Executa uma pipeline de binários que estão contidos no array de strings
int execute_commands_in_pipeline(CONFIG c, char* input, char* output,
                                 char** binaries_array,
                                 int number_of_commands);

//Função principal que cria a mensagem a ser enviada ao cliente
//através do comando status
void create_status_message(int fileDescriptor, CONFIG c, Queue q,
                           int vetor_instances_original[]);

//Fecha o servidor
void close_server(int signum);
```

Além disso, possui uma função `main` que efetua todo o trabalho. De seguida, segue um pequeno excerto em pseudo-código que exemplifica o que o servidor faz:

```
int main(int argc, char const *argv[]){
    inicializa_config();//Cria a lista ligada CONFIG
    inicializa_queue();//Cria a queue

    //útil para calcular nº de instâncias a serem usadas (comando status)
    cria_array_instancias_originais();

    criaçao_fifo_servidor();//pipe com nome de ligação servidor <-> cliente

    abertura_fd_comunicação();
    abertura_fdr_servidor_sobrevivencia();

    while((bytes_read = readln(fd_comunicacao,buffer,size_buff)) > 0){
        parse_modo_operação(); //S: status; P: proc-file

        if(deteta_status()){
            executa_status();
        }

        if(deteta_procFile()){
            executa_procFile();
        }

    }
    apaga_fifo_servidor();
    return 0;
}
```

2.3 config.c

Este módulo constitui a forma de armazenamento dos binários obtidos através dos programas fornecidos pela equipa docente.

```
typedef struct configuration{
    char* binary_name; //nome do binário a executar (ex. "nop")
    char* path_name; //path do ficheiro objeto (ex. "obj/nop")
    int max_instances; //número máximo de instâncias
    struct configuration *next;
}*CONFIG;

//Cria a lista ligada de configurações,
//fazendo parse de uma linha do ficheiro de configuração
CONFIG create_config (char* line, char* binarys_folder, CONFIG c);

//Carrega todas as configurações, lendo o ficheiro de configuração
CONFIG load_configurations (char* config_filename, char* binarys_folder);

//Tendo o nome de um binário, procura-o na lista ligada e executa-o
void execute_config (CONFIG cs, char* binary_name);
```

```
17
18  //Cria o path para um dado binário
19  char* get_binary_filename (char* binary_name, char* binary_folder);
20
21  //Devolve a cópia de um nodo que contém a configuração de um binário
22  CONFIG get_Config(char* binary_name, CONFIG cs);
23
24  //Incrementa (ou decrementa) as instâncias de um dado binário
25  void changeInstances(CONFIG c, char* binary, char* operation);
26
27  //Verifica se todos os binários do array podem ser executados
28  //ou seja, verifica se as instâncias são maiores que 0
29  int canExecuteBinaries(CONFIG c, char** binaries_array,
30                         int number_of_commands);
31
32  //Verifica se as instâncias de um dado binário são maiores que 0
33  int match_binary_with_instances(CONFIG cs, char* binary_name);
34
35  //Decrementa as instâncias de todos os binários do array
36  void request_enter(CONFIG cs, char** binaries_array,
37                     int number_of_binaries);
38
39  //Incrementa as instâncias de todos os binários do array
40  void request_out(CONFIG cs, char** binaries_array, int number_of_binaries);
41
42  //vetor contém as instâncias originais consoante a ordem na CONFIG
43  int get_original_inst(int indice, int vetor_instances_original[]);
44
45  //Cria a mensagem com os elementos da CONFIG que vai ser enviada ao cliente
46  //que efetuar o comando status
47  void get_status_from_config(int fileDescriptor, CONFIG c,
48                              int vetor_instances_original[]);
```

2.4 `task.c`

Este módulo constitui a forma de armazenamento dos pedidos de processamento de ficheiros enviados pelo cliente ao servidor que não poderam ser atendidos, pelo facto de pelo menos um dos executáveis ter atingido o nº máximo de instâncias que poderão estar a ser usadas ao mesmo tempo de um determinado binário.

```c
typedef struct task {
    int id; //Identificação das task
    char* file_input; //Ficheiro de input da task
    char* file_output; //Ficheiro de output da task
    char** binaries_to_execute; //Array de binários a executar
    int binaries_num;//Tamanho do array de binários a executar
    struct task* prox;
} *TASK;

typedef struct queue {
    struct task* inicio , *fim ;
} *Queue;

Queue init_queue(); //Inicializa a queue

//Adiciona uma task (com os elementos necessários) no fim da queue
//O ID da task vai ser do cliente que mandou o pedido
void add_task (Queue q, char* file_input,
               char* file_output, char** binaries_to_execute,
               int number_of_binaries, int client_pid);

//Remove a task que se encontra no inicio da queue
void remove_task (Queue q);

//Liberta a memoria utilizada por um array de strings
void freeArrayList(char** arrayStrings, int size);

//Função de debug: imprime o conteudo de uma queue
void printQueue (Queue q);

//Copia um array de strings para outro
char** arrayStrings_Copy (char** original, int array_size);

//Cria a mensagem com os elementos da queue que vai ser enviada ao cliente
//que efetuar o comando status
void get_status_from_queue(int fileDescriptor, Queue q);
```

Listing 1: API do módulo `task.c`

2.5 `utilities.c`

Este módulo é constituído por todas as funções auxiliares aos outros módulos da aplicação. Este possui várias funções úteis, tais como:

```c
//Lê uma linha inteira até ao caractere '\n'
size_t readln(int fd, char* line, size_t size);

//Verifica se um binário pode ser executado, ou seja,
//se possui o nome dos binários dados pela equipa docente
int validate_binary_to_execute (char* binary_to_execute);

```

```
8  //Converte um número n numa string
9  char* inttoString(int n);
10
11 //Adiciona uma string a um array de strings
12 char** add_string_to_array(char** array_of_strings,
13                            const char* string_to_add);
14
15 //Devolve quantos binários irão ser executados
16 int get_binaries_num(char* buffer);
17
18 //Cria um array de binários diretamente da mensagem enviada pelo cliente
19 char** create_binaries_array(char* buff, int number_of_binaries);
20
21 //Concatena a string 'a' com a string 'b', devolvendo o resultado em 'a'
22 void my_strcat(char* a, char* b);
```

3 Comunicação entre clientes e servidor

A comunicação entre o cliente e o servidor é feita através de um pipe com nome `(o FIFO)`. O servidor lê o pedido do cliente e executa conforme desejado.

Cada pedido do cliente é enviado para o servidor numa espécie de "sinal". Este sinal é uma string que contém a informação relacionada ao pedido do servidor, o seu pid, os campos do seu pedido necessários para a sua execução e uma "flag", que pode ser "P"e "S"(no caso do cliente) ou "1"e "0"(explicado mais a frente).

No caso de o cliente pedir o status, é enviada a flag "S"ao servidor, e este, utilizado as devidas funções, devolve ao cliente o status.

No caso do proc-file, é enviada a flag "P"ao servidor onde, através de um novo processo, verifica se é possivel executar o pedido naquele momento. Se não for possível, é adicionado a queue e o cliente recebe uma mensagem a dizer que o pedido ficou pendente, se for é executado naquele momento, é devolvida uma mensagem ao cliente e o servidor "envia para si mesmo a flag "0.

Esta flag "0"faz com que o servidor vá a queue verificar se o pedido á sua cabeça já pode ser feita. Se não for, termina aí. Se for, ele é removido da queue,é executado e é enviada uma mensagem ao seu cliente a informar-lo do desfecho. O servidor manda por fim a flag "1"a si mesmo.

Esta última flag, por sua vez, repete o processo de cima enquanto houver pedidos na queue para executar. Assim, aproveitemos para falar agora da queue.

No nosso projeto, a queue segue o algoritmo de escalonamento `FCFS`(First Come, First Served), ou seja, apesar dos pedidos que não vão para a queue serem executados de forma concorrente, os da queue, na verdade, não o são. Por exemplo, um pedido na queue que possa ser executado naquele momento, só o será quando os pedidos que estão à sua frente sejam executados. Implementamos este algoritmo porque, para além de ser o que consideramos mais justo, era também o que achamos mais simples de implementar e o que permitia ao servidor correr de forma mais consistente e coerente, o que facilitou imenso nos processos de debugging.

Terminado os pedidos, o servidor fica ligado, à espera de mais pedidos, até que o seu processo seja terminado.

4 Funcionalidades básicas

Neste trabalho prático, existem algumas funcionalidades mais básicas a apresentar. São elas a `proc-file` e a `status`.

4.1 Comando `proc-file`

Este comando consiste na aplicação de vários binários a executar a um ficheiro de *input*, guardando o resultado num ficheiro de *output*. De forma a permitir esta funcionalidade, foi criada a função `execute_commands_in_pipeline` (que pode ser encontrada no ficheiro `sdstored.c`), onde, em primeiro lugar, redirecionamos o ficheiro de *input* para o `Standard Input` e o ficheiro de *output* para o `Standard Output`; de seguida, vamos criando *pipes* e fechando os descritores desses pipes consoante a necessidade que temos em permitir o fluxo ao longo dos *pipes anónimos*. No final desta execução, o cliente poderá, então, consultar o ficheiro pretendido.

No entanto, o servidor tem executáveis com um número máximo de instâncias, pelo que, caso algum dos binários a executar tenha atingido este valor, é criada uma `task` que contém os dados necessários para executar este pedido e é adicionado numa *queue*. Sempre que acontece uma adição à *queue*, o respetivo cliente recebe a mensagem `"pending..."` e, quando é finalizado o seu pedido, recebe a mensagem `"done!"`.

Nota: Pressupomos que o cliente manda os seus pedidos com uma ordem válida.

4.2 Comando `status`

Outro comando que também disponibilizamos é o `status`. Este consiste em saber os detalhes da *queue* e da lista ligada `CONFIG`. Para tal, foi criada a função `create_status_message`, que está encarregue de ir à *queue* e escrever cada linha da mensagem no seguinte formato:

```
task \#[PID Cliente]: proc-file [ficheiro input] [ficheiro output] [binário 1] [binário 2]...
```

Depois, vai consultar a lista ligada `CONFIG`, gerando mensagens com o seguinte formato:

```
transf [nome do binário]: [instâncias no momento]/[instâncias máximas] (running/max)
```

Por fim, há que referir a nuance que, na função acima, é passada como argumento o descritor do cliente, pelo que efetuamos um `write` direto em vez de formar a *string* resultante do comando *status* e enviar toda de uma vez para o respetivo cliente (que foi a nossa primeira abordagem, mas tivemos problemas em criar a *string resultante*).

5 Funcionalidades extras

Além das funcionalidades básicas, existiam algumas extra pedidas pela equipa docente, que são:

1. Quando um pedido termina, deverá ser reportado ao cliente o número de bytes recebidos e o número de bytes produzidos por operação. Ou seja, para cada operação proc-file deverá ser reportado o número de bytes que foram lidos do ficheiro a ser processado, bem como o número de bytes que foram escritos no ficheiro final após aplicar as transformações.

2. Se receber o sinal SIGTERM, o servidor deve terminar de forma graciosa, deixando primeiro terminar os pedidos em processamento ou pendentes, mas rejeitando a submissão de novos pedidos.

3. Assuma que as operações proc-file podem ter diferentes prioridades. O servidor deverá dar prevalência a pedidos com maior prioridade. Para realizar esta funcionalidade, cada operação deve ser acompanhada da sua prioridade (identificadas como inteiros de 0 a 5, em que 5 atribui prioridade máxima)

No entanto, não implementámos nenhuma das enunciadas acima. Uma sugestão de trabalho futuro é criar estas funcionalidades.

6 Conclusão

Com este trabalho prático conseguimos explorar e entender melhor os mecanismos que compõem um sistema operativo. Consideramos que o produto final é aceitável e que executa as funcionalidades básicas de acordo como pedido no enunciado. Sabemos, também, que se tivessemos gerido melhor o nosso tempo académico, conseguiríamos efetuar todas as funcionalidades extras.

O maior ponto a melhorar no projeto é, a nosso ver, o tratamento da queue, o que faz com que os pedidos não sejam todos 100% concorrentes, como explicado em cima.

Apesar disso, conseguimos aplicar os conhecimentos que obtemos ao longo desta UC, conhecimentos tais que iremos aplicar e levar connosco para o nosso futuro académico e profissional.