Universidade do Minho

Licenciatura em Engenharia Informática

Sistemas Operativos

SDStore: Armazenamento Eficiente e Seguro de Ficheiros

Grupo 48
Ana Murta (A93284)
Ana Henriques (A93268)
Rui Coelho (A58898)

maio, 2022

Conteúdo

Capítulo 1

Introdução

O presente projeto centra-se na implementação de um serviço cujo objetivo central consiste na compressão e cifragem de ficheiros. Este serviço visa permitir aos utilizadores armazenar cópias seguras dos seus ficheiros, sendo segundo um modelo de cliente-servidor – onde o cliente executa pedidos, estando o servidor encarregue de atender e satisfazer os pedidos recebidos. Assim sendo, o serviço a implementar deve permitir executar operações de compressão e cifragem de ficheiros a pedido do cliente, sendo, também, capaz de recuperar o conteúdo original de ficheiros previamente transformados, caso o cliente assim o deseje.

Aquando da submissão de um pedido, o servidor encontra-se encarregue de o gerir, colocando-o em execução caso todos os recursos necessários se encontrem disponíveis, ou colocando-o em espera na eventualidade de os recursos estarem ocupados – sendo o pedido tratado aquando da libertação de recursos usados para responder a outros pedidos. Adicionalmente, o cliente pode consultar as estatísticas referentes ao trabalho em curso por parte do servidor, obtendo uma descrição da utilização dos recursos disponíveis e das tarefas que estão presentes em sistema.

Em suma, o trabalho desenvolvido, com recurso à linguagem de programação imperativa *C*, centra-se no desenvolvimento de dois programas que comunicam entre si: *sdstore*, correspondente ao cliente que efetua pedidos, e *sdstored*, sendo o servidor que responde aos pedidos.

Capítulo 2

Arquitetura cliente-servidor

O serviço implementado segue um modelo cliente-servidor. Estas arquiteturas são caracterizadas pela existência de duas entidades distintas, cliente e servidor, que comunicam entre si, efetuando e respondendo a pedidos. O cliente – *sdstore* no caso aqui apresentado, efetua pedidos de transformações a ficheiros ao servidor. O servidor – *sdstored* – encarrega-se de responder a esses pedidos, efetuando as transformações solicitadas pelo cliente, transformando o ficheiro de origem.

Um paradigma cliente-servidor permite, primeiramente, a divisão clara entre as responsabilidades dos sistemas envolvidos o que, adicionalmente, facilita a manutenção e melhoramento dos sistemas. A qualquer momento tanto cliente como servidor podem ser modificados, do ponto de vista de funcionamento interno, sem que a comunicação entre ambos seja prejudicada – assumindo, naturalmente, a manutenção dos meios de comunicação. Esta modularidade permite que o trabalho executado seja centralizado no servidor, tornando a manutenção do sistema como um todo mais simples.

2.1 Cliente

O cliente do serviço, *sdstore*, efetua pedidos ao servidor. Os pedidos a ser efetuados possuem uma estrutura específica que deve ser respeitada para que um pedido siga, com sucesso, para o servidor. O cliente pode efetuar dois tipos distintos de pedidos. A execução, no terminal, do comando *status* permite ao cliente consultar as estatísticas do servidor, i.e., é efetuado um pedido ao servidor para consultar o estado de ocupação dos seus recursos e a lista de tarefas (em execução e pendentes) registadas no servidor. O comando abaixo apresentado exemplifica o pedido explicado, não sendo necessário argumentos adicionais para além do *status*.

```
$ ./sdstore status
```

Adicionalmente, o cliente pode ainda efetuar um pedido de transformação de um ficheiro, *proc-file*, sendo necessário um conjunto de argumentos adicionais para que o pedido seja válido. Um pedido desta natureza pode ter uma prioridade associada – sendo a prioridade caracterizada por um inteiro com o valor mínimo de 0 (baixa prioridade) e valor máximo de 5 (prioridade máxima). Este argumento é de caráter opcional e caso o cliente não especifique um valor, foi convencionado que o pedido terá o valor mais baixo de prioridade, ou seja, 0.

Os pedidos de processamento de ficheiros necessitam, obrigatoriamente, da especificação do caminho dos ficheiros de *input* e de *output* – correspondendo, respetivamente, ao ficheiro de entrada a ser transformado e ao ficheiro de saída, resultante da aplicação das transformações. Seguindo a estes argumentos, deve ser especificadas as transformações a ser executadas ao ficheiro – sendo necessário, no mínimo, a listagem de uma transformação. O comando abaixo apresentado exemplifica a estrutura de um pedido de processamento de um ficheiro.

```
$ ./sdstore proc-file [-p priority] input-file output-file trans1 trans2 ...
```

Os pedidos de transformação de ficheiros podem especificar tipos diferentes de transformações: compressão/descompressão (*bcompress*, *bdecompress*, *gcompress* e *gdecompress*); cifragem/decifragem (*encrypt* e *decrypt*); cópia (*nop*).

Com vista a não sobrecarregar o servidor com pedidos inválidos, o programa *sdstore* opera inicialmente sobre o pedido que pretende efetuar. Apenas pedidos válidos são comunicados ao servidor. Assim sendo, um pedido é considerado válido se: (i) é usado o comando *status* sem quaisquer argumentos adicionais; (ii) é usado o comando *proc-file* com os argumentos obrigatórios – sendo que as transformações a serem pedidas devem ser suportadas pelo servidor. Na eventualidade do pedido ser inválido é fornecida uma mensagem de erro, para informar o motivo pelo qual o comando não pode ser enviado ao servidor.

2.2 Servidor

O servidor do serviço, *sdstored*, encontra-se encarregue de responder aos pedidos recebidos pelos clientes. O comando abaixo apresentado permite que o servidor arranque e carregue as configurações necessárias para o seu funcionamento. Como tal, é necessária a inserção do caminho para o ficheiro de configurações (*config-filename*) e o caminho para a pasta que contém as transformações suportadas pelo servidor (*transformations-folder*).

```
$ ./sdstored config-filename transformations-folder
```

O servidor atende, em simultâneo, pedidos de diferentes clientes, operando sobre os mesmos em função da disponibilidade dos seus recursos, executando pedidos para os quais existam recursos disponíveis e colocando em espera os pedidos que necessitam de recursos que se encontram ocupados – executando-os posteriormente.

2.3 Comunicação

Tal como foi anteriormente mencionado, o servidor é capaz de atender vários clientes em simultâneo. Para tal, foi criado um canal de comunicação principal – através de um *named pipe* – que se encontra disponível para receber mensagens dos clientes. Cada cliente conecta-se ao canal principal, enviando ao servidor o identificador único que o caracteriza – sendo este identificador correspondente ao *pid* do processo que descreve o cliente. Após o envio desta mensagem, cliente e servidor usam canais de comunicação específicos, também *named pipes*, para trocar mensagens[1]. O canal de comunicação é fechado pelo servidor quando se verificam um de dois acontecimentos: (i) o pedido requisitado pelo cliente foi satisfeito; (ii) não é possível executar o pedido devido à ocorrência de um erro.

A criação de um canal principal de comunicação, apenas para primeiro contacto entre servidor e cliente, visa, essencialmente, permitir que o servidor dedique canais de comunicação específicos para cada cliente. Deste modo, cada cliente possui um canal próprio de comunicação com o servidor, não havendo possibilidade de os pedidos efetuados serem misturados e/ou corrompidos aquando da leitura de um *FIFO* único e partilhado por todos os clientes.

[1]Por pedido, são criados dois canais de comunicação aos quais cliente e servidor se conectam: um destinado a escrita e outro a leitura.

Capítulo 3

Serviço *SDStore*

3.1 Estruturas de dados

O funcionamento correto do serviço implementado assenta, em parte, na criação de estruturas de dados que permitam armazenar, em memória, as informações necessárias para que o servidor possa operar. A estrutura *Transformation*, que corresponde a uma lista ligada, armazena a informação acerca das diversas operações suportadas pelo servidor. Aquando da leitura do ficheiro de configurações do servidor[1], são carregados para memória o nome das operações e o número máximo de instâncias simultâneas possíveis. Uma vez que é necessário armazenar o número de instâncias atualmente em uso, foi adicionada a variável *currently_running*, a cada transformação, para o efeito.

```
typedef struct Transformation{
    char operation_name[SMALL_BUFF_SIZE];
    int max_operation_allowed;
    int currently_running;
    struct Trans * next;
} * Trans;
```

Do mesmo modo, é igualmente importante o servidor armazenar a informação acerca dos pedidos que recebe e, para tal, foi criada a estrutura *Task*, que também consiste numa lista ligada. Cada tarefa em sistema é caracterizada pelo *pid* do processo que efetuou o pedido – sendo, aquando da execução – sendo adicionado o *pid* do processo encarregue de executar a tarefa. Adicionalmente, uma tarefa possui ainda o comando executado pelo cliente, i.e., o pedido em si, a sua prioridade, o descritor do ficheiro usado para comunicar com o cliente e o estado atual do pedido.

```
typedef struct Task{
    pid_t pid_request;
    pid_t pid_executing;
    int fd_writter;
    int priority;
    char command[MID_BUFF_SIZE];
    char status[SMALL_BUFF_SIZE];
    struct Task * next;
} * Task;
```

Com vista a facilitar o desenvolvimento aplicacional, foram criadas duas listas de tarefas: uma referente às tarefas em execução e outra para armazenar os pedidos pendentes, i.e., pedidos que se encontram à espera de recursos, que se encontram no momento ocupados, para poderem ser

[1] através da função *loadServer* (Anexo A.1).

executados. Uma vez que associado às tarefas se encontra o conceito de prioridade, o armazenamento das tarefas em ambas as listas toma segue este critério para a sua ordenação: as tarefas organizam-se da mais prioritária, para a menos prioritária.

3.2 Pedidos

3.2.1 Tratamento e organização

Aquando da receção de um pedido por parte do cliente, pelos meios de comunicação anteriormente descritos, o servidor determina o tipo de pedido recebido (*status* ou *proc-file*) para poder operar sobre o mesmo.

Caso o pedido recebido seja do tipo *status*, o servidor efetua a *system call fork* para criar um processo filho encarregue de enviar a informação referente ao estado do servidor para o cliente[2]. Uma vez que este tipo de pedidos não consome recursos do servidor – no que diz respeito às transformações – não é associado a nenhuma lista de tarefas.

Por outro lado, se o pedido recebido pelo servidor for referente a um *proc-file*, é acionado um conjunto de operações por parte do servidor que objetivam responder ao pedido.

Em primeira instância é efetuada uma validação do pedido que visa determinar se é futuramente possível executar o pedido recebido, i.e., verifica se o número de recursos disponibilizados pelo servidor é suficiente para executar o pedido, uma vez que um pedido apenas é executado quando todos os recursos necessários para a sua conclusão se encontrem livres. Esta verificação visa impedir que pedidos que exijam um número de transformações superior às permitidas pelas configurações do servidor fiquem, eternamente, em sistema à espera de ser executados – por exemplo, se o servidor permitir 2 instâncias simultâneas da transformação *nop* e for recebido um pedido que pretenda executar 3 vezes esse comando, o pedido ficará pendente e nunca será executado, ficando cliente à espera da conclusão do pedido, e servidor sem recursos para tal.

Uma vez validado o pedido, o servidor verifica se possui, atualmente, livres os recursos necessários para a sua execução[3]. Se os recursos se encontrarem ocupados, o pedido é colocado na lista de tarefas pendentes, sendo o cliente informado que o pedido encontra-se em lista de espera. Por outro lado, se estiverem livres os recursos necessários para satisfazer o pedido, a tarefa é adicionada à lista de execução, os recursos são dados como ocupados, e o cliente é informado de que o seu pedido encontra-se em tratamento.

A terminação da execução de um pedido, por parte do servidor, desencadeia a fase final[4] de comunicação com o cliente: os recursos utilizados são libertados e o cliente é informado da conclusão da tarefa, recebendo uma mensagem de conclusão, com o tamanho, em *bytes*, dos ficheiros de *input* e *output*.

3.2.2 Execução

A execução das tarefas é efetuada com recurso à função *executeTask* (Anexo A.5) que recebe como argumentos uma lista de *tokens*[5] e o número total de elementos da lista. Com vista a permitir que o servidor continue a atender pedidos, a execução de tarefas é do encargo de processos-filho do processo principal, criados através da *system call fork*. Para que, posteriormente, seja efetuado o levantamento dos processos que terminaram de ser executados, a função *executeTask* retorna o valor do *pid* do processo encarregue de efetuar as transformações do pedido – sendo este identificador posteriormente associado à estrutura da tarefa, no campo *pid_executing*.

A gestão e execução das tarefas, por parte de um processo-filho, decorre num processo iterativo, cujo trajeto depende do número de transformações a executar[6]. No caso mais simples, o pedido

[2]Através da função *sendStatus* (Anexo A.2).

[3]Sendo usada a função *evaluateResourcesOcupation* para o efeito (Anexo A.3).

[4]Através da função *cleanFinishedTasks* – (Anexo A.6).

[5]Com a informação providenciada pelo cliente acerca da prioridade (caso tenha sido especificada), dos ficheiros e transformações a executar.

[6]Todas as transformações executadas recorrem à *system call execl*.

em tratamento possui um pedido apenas (sendo o processo iterativo executado uma única vez). Caso tal se verifique, é o próprio processo-filho que estará encarregue de a executar, efetuando os redirecionamentos necessários de descritores de ficheiros – através da *system call dup2* – para produzir o resultado esperado.

Um modo de operação diferente é necessário quando o pedido recebido contém mais do que uma transformação. Neste caso, em cada passo do processo iterativo, é criado um novo processo para executar a transformação[7], sendo a comunicação entre os diversos processos efetuada através de *pipes* anónimos – criados pelo processo pai, encarregue de gerir o fluxo de execução do servidor, mas abertos apenas pelo processo-filho[8] encarregue de gerir o processo de execução das tarefas. A criação destes canais de comunicação visa, essencialmente, permitir a comunicação entre os diversos processos que vão sendo gerados no decurso da execução iterativa das transformações – sendo, para tal, necessário efetuar o redirecionamento correto dos descritores de ficheiro, dos *pipes*, para que a comunicação ocorra.

Findo o processo de execução iterativa das transformações o (primeiro) processo-filho, criado pelo processo principal do programa, espera pelo término de execução dos processos-filho que criou, i.e., pelo fim da execução das transformações, terminando o seu próprio processo através da função *_exit*.

3.3 Sinais

O controlo do fluxo de execução do servidor é auxiliado pelo uso dos sinais. Aquando do arranque, o servidor define um conjunto de rotinas para tratamento de sinais, fazendo uso da *system call signal*, tal como pode ser observado no excerto de código abaixo apresentado.

```
signal(SIGTERM, termiation_handler);
signal(SIGINT, termiation_handler);
signal(SIGCHLD, sigchild_handler);
signal(SIGALRM, sigalarm_handler);
```

Os sinais *SIGTERM* e *SIGINT* são tratados pela rotina *termiation_handler*. Esta rotina visa terminar, de forma graciosa, o servidor. Aquando da receção de um destes sinais, o servidor fecha o canal de comunicação principal, deixando de estar disponível para responder a pedidos de novos clientes. Adicionalmente, o servidor executa continua a executar todas as tarefas que se encontram em curso e, ainda, executa as tarefas pendentes – terminando apenas quando não existirem tarefas em memória.

```
void termiation_handler(int sig){
    close(channel);
    unlink(fifo);
    while(executing_tasks != NULL || pending_tasks != NULL)
        pause();
}
```

A rotina *sigchild_handler* define o conjunto de operações a executar quando é recebido um sinal *SIGCHLD*. Como é observável no código abaixo apresentado, o *handler* extrai o *pid* do processo que terminou e é efetuado o tratamento final do pedido, com recurso à função *cleanFinishedTasks* (Anexo A.6) – onde são libertados os recursos ocupados pela tarefa e é enviada a mensagem de terminação ao cliente (com o tamanho de *input* e *ouput* dos ficheiros).

```
void sigchild_handler(int signum){
    pid_t pid;
```

[7] Note-se que a execução da *system call execl* substitui o código de execução do programa, pelo que a não criação de processos-filho impediria a continuidade de execução das transformações.

[8] Através da *system call pipe*.

```
    int status;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0){
        cleanFinishedTasks(pid, status);
    }
}
```

Por fim, *sigalarm_handler* permite ao servidor verificar, periodicamente, a viabilidade de execução das tarefas colocadas na lista de tarefas pendentes. Esta verificação é despoletada no arranque do servidor, com recurso à função *alarm*, sendo, após um segundo, enviado um sinal *SIGALRM*. A verificação da viabilidade de execução das tarefas pendentes é efetuada pela função *checkPendingTasks* (Anexo A.7). Assim que os recursos se encontrem disponíveis para uma dada tarefa, essa tarefa segue para a lista de tarefas em execução.

```
void sigalarm_handler(int signum){
    checkPendingTasks();
    alarm(1);
}
```

Capítulo 4

Teste de funcionalidades

A fim de testar as funcionalidades e o comportamento do serviço foi executado um conjunto variado de comandos para o efeito. Os comandos abaixo apresentados consistem num conjunto de pedidos válidos, aos quais o servidor é capaz de responder corretamente. Tal como esperado, no caso de os recursos se encontrarem ocupados, o servidor coloca as tarefas em lista de espera, executando-as assim que possível.

```
$ ./sdstore status
$ ./sdstore proc-file -p 2 ../docs/enunciado.pdf ../docs/teste.pdf nop
$ ./sdstore proc-file -p 3 ../docs/enunciado.pdf ../docs/teste1
        nop bcompress bdecompress encrypt decrypt
$ ./sdstore proc-file -p 1 ../docs/enunciado.pdf ../docs/teste2 encrypt bcompress
$ ./sdstore proc-file ../docs/teste2 ../docs/teste3.pdf nop bdecompress decrypt
$ ./sdstore proc-file ../docs/enunciado.pdf ../docs/teste4 gcompress nop bcompress
$ ./sdstore proc-file ../docs/enunciado.pdf ../docs/teste5 nop bcompress gdecompress
```

Por fim, foi ainda executado um conjunto adicional de comandos, para averiguar o comportamento do servidor aquando da presença de comandos inválidos. A execução destes comandos não levanta problemas para o servidor, uma vez que os pedidos inválidos (e.g, com número inválido de argumentos ou com transformações não suportadas pelo servidor) são filtrados antes de chegar ao servidor. Adicionalmente, caso o pedido seja válido mas, por exemplo, exceda as configurações permitidas pelo servidor ou seja fornecido um caminho para um ficheiro inexistente, o servidor responde adequadamente ao comando, informando o cliente do erro que o pedido originou.

```
$ ./sdstore proc-file ../docs/enunciado ../docs/teste6.pdf nop
$ ./sdstore proc-file ../docs/enunciado.pdf ../docs/teste7.pdf nada
$ ./sdstore proc-file ../docs/enunciado.pdf ../docs/teste8.pdf nop nop nop nop nop
$ ./sdstore proc-file ../docs/enunciado.pdf ../docs/teste9.pdf
$ ./sdstore sta
```

Capítulo 5

Conclusão

O trabalho desenvolvido centrou-se na criação de um servido que possibilita a implementação de operações de compressão e cifragem de ficheiros, segundo uma ótica de cliente-servidor. O programa *sdstore*, cliente, efetua pedidos de processamento de ficheiro ou de consulta de estatísticas. O programa *sdstored*, servidor, responde a esses pedidos, efetuando as operações solicitadas por parte do cliente.

Face aos objetivos estabelecidos e ao trabalho apresentado, a equipa considera ter alcançado com sucesso as metas esperadas, criando um programa funcional e simples que preenche os requisitos estipulados. Algumas funcionalidades futuras poderiam ser inseridas/melhoradas, de modo a fortalecer o programa desenvolvido e a aumentar o conjunto de funcionalidades que este fornece. A verificação da correção de comandos pode ser movida para a lógica do servidor, tornando as duas componentes mais independentes, i.e., do modo como o serviço foi desenvolvido alterações nas transformações fornecidas pelo o servidor requer manutenção do serviço tanto do lado do cliente, como do lado do servidor. Adicionalmente, de modo a tornar o processo de espera de tarefas mais "justo", deve ser adicionado algum algoritmo que procure inibir que as tarefas colocadas como pendentes fiquem, indefinidamente, à espera para ser executadas – algo passível de acontecer a uma tarefa usar muitos recursos e tiver baixa prioridade.

Apêndice A

Algoritmos

A.1 Carregamento de configuração servidor

```
bool loadServer(char * path[], Trans * tr){
    char buffer[MAX_BUFF_SIZE];
    Trans tmp_tr = NULL;
    int config_fd = open(path[1], O_RDONLY);

    if(config_fd < 0)
        return false;

    while(readLine(config_fd, buffer))
        tmp_tr = addTransformation(tmp_tr,buffer);

    close(config_fd);
    * tr = tmp_tr;

    return true;
```

A.2 Envio de estado do servidor

```
void sendStatus(Trans * tr, Task * executing, Task * pending, int writer){
    char buff[MAX_BUFF_SIZE] = "";
    int counter = 1, total_bytes = 0;

    while(* tr){
        total_bytes = sprintf(
            buff, "[Transformation] %s: %d/%d running/max\n",
            (*tr)->operation_name,
            (*tr)->currently_running,
            (*tr)->max_operation_allowed);
        write(writer, buff, total_bytes);
        buff[0]= '\0';
        tr = & ((*tr)->next);
    }

    buff[0]= '\0';

    while(* executing){
```

```
            total_bytes = sprintf(
                    buff, "[Task #%d] %d: %s\nPriority: %d | Status: %s\n",
                    counter++,
                    (*executing)->pid_request,
                    (*executing)->command,
                    (*executing)->priority,
                    (*executing)->status);
            write(writer, buff, total_bytes);
            buff[0]= '\0';
            executing = & ((*executing)->next);
        }

        buff[0]= '\0';

        while(* pending){
            total_bytes = sprintf(
                    buff, "[Task #%d] %d: %s\nPriority: %d | Status: %s\n",
                    counter++,
                    (*pending)->pid_request,
                    (*pending)->command,
                    (*pending)->priority,
                    (*pending)->status);
            write(writer, buff, total_bytes);
            buff[0]= '\0';
            pending = & ((*pending)->next);
        }
    }
```

A.3 Validação de recursos

```
bool evaluateResourcesOcupation(Trans * tr, char * transformations[], int nrTrans){
    int resources_needed;

    while(* tr){
        resources_needed = 0;

        for(int i = 2; i < nrTrans; i++){
            if(strcmp((*tr)->operation_name, transformations[i]) == 0)
                resources_needed++;
        }

        if((*tr)->max_operation_allowed < ((*tr)->currently_running + resources_needed))
            return false;

        tr = & ((*tr)->next);
    }

    return true;
}
```

A.4 Conclusão de pedidos

```
void cleanFinishedTasks(pid_t pid_ex, int status){
    int num_transformations;
    off_t size_input = 0, size_ouput = 0;
    char * transformationsList[SMALL_BUFF_SIZE];
    char concludedMessage[MID_BUFF_SIZE];
    Task * tmp = &executing_tasks;

    if(WIFEXITED(status)){
        while(* tmp && ((*tmp)->pid_executing != pid_ex))
            tmp = & ((*tmp)->next);

        if(* tmp){
            num_transformations = lineSplitter((*tmp)->command, transformationsList);
            freeResources(&sc, transformationsList, num_transformations);
            size_input = calculateSize(transformationsList + 1, 0);
            size_ouput = calculateSize(transformationsList + 1, 1);
            sprintf(concludedMessage,
                "concluded (bytes-input: %lld, bytes-output: %lld)\n",
                size_input,
                size_ouput);
            write((*tmp)->fd_writter, concludedMessage, strlen(concludedMessage));
            close((*tmp)->fd_writter);
            deleteTask_byRequestPID(&executing_tasks, (*tmp)->pid_request);
        }
    }
}
```

A.5 Execução de tarefas

```
int executeTask(char * args[], int size){
    int index, nr_transformations, start_index, priority, input, output;
    pid_t pid;

    index = 0;

    if ((priority = priorityCheck(args)) != -1)
        index = 2;

    nr_transformations = size - 2 - index;
    start_index = 2 + index;

    if((input = open(args[index],O_RDONLY, 0666)) == -1){
        printMessage(fileError);
        return -1;
    }

    if((output = open(args[index+1], O_CREAT| O_TRUNC | O_WRONLY ,0666)) == -1){
        printMessage(fileError);
        return -1;
    }
```

```
int pipes[nr_transformations][2], status[nr_transformations];
char full_path[MAX_BUFF_SIZE];
char * transf[nr_transformations];

switch(pid = fork()){
    case -1:
        printMessage(forkError);
        return -2;
    case 0:
        for(int i = 0; i < nr_transformations; i++){
            transf[i] = malloc(sizeof(char) * MAX_BUFF_SIZE);
            strcpy(transf[i], args[start_index + i]);
            strcpy(full_path, transformations_path);
            strcat(full_path, transf[i]);

            if(nr_transformations == 1){
                strcpy(full_path, transformations_path);
                strcat(full_path, transf[0]);
                dup2(input, STDIN_FILENO);
                dup2(output, STDOUT_FILENO);
                execl(full_path ,transf[0], NULL);
                _exit(0);
            }
            else{
                if (i == 0){
                    if (pipe(pipes[i]) != 0)
                        return -1;
                    switch(fork()){
                        case -1:
                            printMessage(forkError);
                            return -2;
                        case 0:
                            close(pipes[i][0]);
                            dup2(input, STDIN_FILENO);
                            dup2(pipes[i][1], STDOUT_FILENO);
                            close(pipes[i][1]);
                            execl(full_path, transf[i], NULL);
                            _exit(0);
                        default:
                            close(pipes[i][1]);
                    }
                }
                else if (i == nr_transformations -1){
                    switch(fork()){
                        case -1:
                            printMessage(forkError);
                            return -2;
                        case 0:
                            dup2(pipes[i-1][0], STDIN_FILENO);
                            dup2(output, STDOUT_FILENO);
                            close(pipes[i-1][0]);
                            execl(full_path, transf[i], NULL);
                            _exit(0);
                        default:
```

```
                            close(pipes[i-1][0]);
                        }
                    }
                    else{
                        if (pipe(pipes[i]) != 0)
                            return -1;
                        switch(fork()){
                            case -1:
                                printMessage(forkError);
                                return -2;
                            case 0:
                                close(pipes[i][0]);
                                dup2(pipes[i][1], STDOUT_FILENO);
                                close(pipes[i][1]);

                                dup2(pipes[i-1][0], STDIN_FILENO);
                                close(pipes[i-1][0]);

                                execl(full_path, transf[i], NULL);
                                _exit(0);
                            default:
                                close(pipes[i][1]);
                                close(pipes[i-1][0]);
                        }
                    }
                    full_path[0] = '\0';
                }
            }
            for(int i = 0; i < nr_transformations; i++){
                free(transf[i]);
                wait(&status[i]);
            }
            _exit(0);
        default:
            close(input);
            close(output);
    }

    return pid;
}
```

A.6 Tratamento de operações terminadas

```
void cleanFinishedTasks(pid_t pid_ex, int status){
    int num_transformations;
    off_t size_input = 0, size_ouput = 0;
    char * transformationsList[SMALL_BUFF_SIZE];
    char concludedMessage[MID_BUFF_SIZE];
    Task * tmp = &executing_tasks;

    if(WIFEXITED(status)){
        while(* tmp && ((*tmp)->pid_executing != pid_ex))
            tmp = & ((*tmp)->next);
```

```
        if(* tmp){
            num_transformations = lineSplitter((*tmp)->command, transformationsList);
            freeResources(&sc, transformationsList, num_transformations);
            size_input = calculateSize(transformationsList + 1, 0);
            size_ouput = calculateSize(transformationsList + 1, 1);
            sprintf(concludedMessage,
                "concluded (bytes-input: %lld, bytes-output: %lld)\n",
                size_input,
                size_ouput);
            write((*tmp)->fd_writter, concludedMessage, strlen(concludedMessage));
            close((*tmp)->fd_writter);
            deleteTask_byRequestPID(&executing_tasks, (*tmp)->pid_request);
        }
    }
}
```

A.7 Revisão de tarefas pendendes

```
void checkPendingTasks(){
    Task * tr = &pending_tasks;
    Task tmp_t = NULL;
    char tmp[MID_BUFF_SIZE];
    int num_transformations, executing_pid;
    char * transformationsList[SMALL_BUFF_SIZE];

    while(* tr){
        strcpy(tmp, (*tr)->command);
        num_transformations = lineSplitter(tmp, transformationsList);
        if(evaluateResourcesOcupation(&sc, transformationsList, num_transformations)){
            write((*tr)->fd_writter, processingStatus, strlen(processingStatus));
            occupyResources(&sc,transformationsList, num_transformations);
            executing_pid = executeTask(transformationsList + 1, num_transformations - 1);
            if(executing_pid == -1 || executing_pid == -2){
                freeResources(&sc,transformationsList, num_transformations);
                if(executing_pid == -1)
                    write((*tr)->fd_writter, fileError, strlen(fileError));
                else
                    write((*tr)->fd_writter, executingError, strlen(executingError));
                deleteTask_byRequestPID(&pending_tasks, tmp_t->pid_request);
                close((*tr)->fd_writter);
            }
            else{
                tmp_t = createTask((*tr)->command, (*tr)->pid_request, (*tr)->fd_writter);
                executing_tasks = taskJoiner(executing_tasks, tmp_t);
                deleteTask_byRequestPID(&pending_tasks, tmp_t->pid_request);
                updateStatusTaskByRequestPID(&executing_tasks, tmp_t->pid_request, "executing");
                updateExecPID(&executing_tasks, tmp_t->pid_request, executing_pid);
                return;
            }
        }
        tr = & ((*tr)->next);
    }
```

```
}
```