Sistemas Operativos

Grupo 107
62608 - Marco Sousa
93271 - José Malheiro

17 de Junho de 2021

Resumo

Este projeto permitiu desenvolver competências de **Engenharia de Software**, nomeadamente *gestão de processos e execução concorrente* de programas.

O objetivo foi implementar um serviço capaz de transformar ficheiros de áudio por aplicação de uma sequência de filtros. Para além deste, permite ainda consultar as tarefas em execução e número de filtros disponíveis e em uso.

1 Introdução

O presente relatório foi redigido no âmbito da unidade curricular (UC) Sistemas Operativos e remete-se à elaboração de um projeto na linguagem de programação C para um **Serviço de Processamento de Áudio**.

A construção do projeto teve como referência a orientação dos docentes da UC e principal objetivo de desenvolver conceitos de gestão de processos, criação de filhos, utilização de sinais, direcionamento de descritores *STDIN, STDOUT, STDERR*. Para além destes, permitiu o aprofundamento sobre o desenvolvimento de programas na linguagem C.

2 Estrutura do Projeto

A arquitetura utilizada foi uma estratégia modular e encapsulada que permitiu a criação de um sistema cliente-servidor fácil de manter e re-estruturar.

- Client
  - Processo que permite fazer pedidos ao servidor (cliente);
- Modules
  - Módulos de apoio ao desenvolvimento
  - Pool
    - Implementação do tipo *thread-pool*
- Server
  - Processo que recebe os pedidos e encaminha para a pool

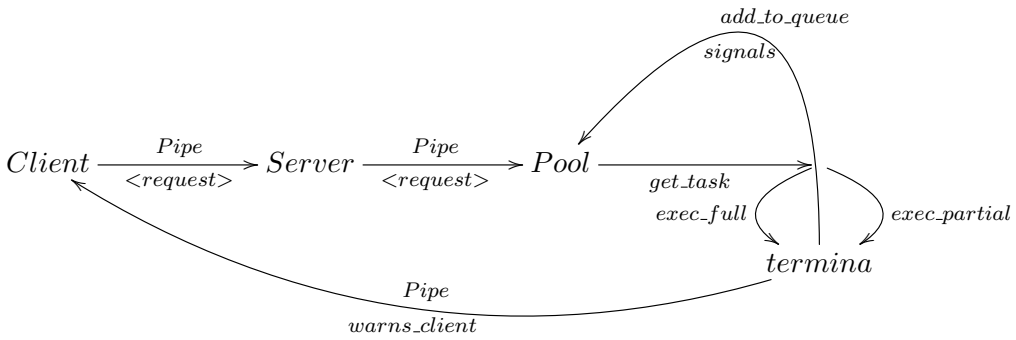
Por forma a verificar a funcionalidade do sistema, é necessário executar o processo servidor com o comando:

```
> bin/aurrasd <path_to_config> <path_to_filters>
```

Posteriormente, pode ser executado qualquer cliente de forma concorrente, utilizando um dos seguintes comandos:

```
> bin/aurras status
> bin/aurras transform <input_file> <output_file> <filter_1 ... filter_n>
```

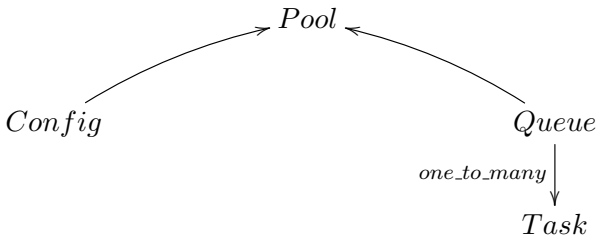
Para gerar os executáveis, deverá executar-se na *bash* o comando `make`.



2.1 Estruturas de Dados

As estruturas de dados desenvolvidas para auxiliar na gestão do sistema, foram *Config*, *Pool*, *Queue* e *Task*.

Cada uma destas estruturas Apresenta uma gestão própria, sendo-lhe disponibilizado uma API com funções para inicializar, adicionar, remover ou atualizar dados, etc. Estão devidamente documentadas para facilitar a uilização por outros para além do seu programador original.



2.1.1 Config

```
typedef struct config *Config;
typedef struct config_server *Config_Server;

struct config
{
    char *filter;
    char *file_name;
    int current, max;
} CNode;

struct config_server
{
    Config *config;
    char *configFolder;
    int total;
    int size;
} CSNode;
```

Estrutura utilizada para armazenar e gerir a configuração do servidor, nomeadamente caminho para os filtros, filtros disponíveis e respetivo número de instâncias em uso e máximas. Ver 4.1.

2.1.2 Task

```
typedef struct task *Task;

typedef enum taskstatus
{
    PENDING,
    WAITING,
    PROCESSING,
    FINISHED,
    ERROR
} Status;

typedef enum execute_type
{
    FULL,
    PARTIAL
} ExecuteType;

typedef struct filters
{
    char **filters; // filtros com a ordem pedida pelo utilizador
    int size;       // size of array
    int num_filters;
    int current_filter;
    char **filters_set; // set de filtros (sem repetições)
    int *filters_count; // lista de contadores (relacionado com filtro)
    int num_filters_set;
} * Filters, FNode;
struct task
{
    char *request;          // pedido completo
    char *command;          // transform vs status
    char *pid;              // pid do cliente que pediu
    int client_fd;          // fd to pipe com cliente
    ExecuteType execution; // tipo de execução (parcial ou completa)
    int executer_pid;       // pid do filho que está a executar
    Status status;          // estado do processamento
    char *input_file;       // path para file input
    char *output_file;      // path para file output
    Filters filters;
} TNode;
```

Cada pedido que o cliente efetua, corresponde a uma tarefa no servidor. Esta tarefa tem um estado próprio, que permite fazer a sua gestão. Ver 4.2

comunicar com cliente	pid do cliente descritor para pipe
execução do comando	filtros associados, por ordem pid do filho a executar comando tipo de execução (parcial ou completa) estado de processamento caminho para input caminho para output
gestão da execução	filtro em execução (execução parcial)

API adicionar filtros
obter número de filtros
obter próximo filtro a executar
obter filtro em execução set de filtros (sem repetições)
set de contador de instâncias necessárias

Assim, quando uma tarefa é iniciada, recebe como argumento o *request* realizado pelo cliente, com a seguinte estrutura:

```
$ <pid_cliente> <transform> <input_file> <output_file> <filtros>
$ <pid_cliente> <status>
```

A partir desta string, é extraída toda a informação necessária através do seu parsing, conforme descrito. É ainda verificado se o ficheiro de input efetivamente existe. Caso não exista, a tarefa não é adiciona à queue e é apresentado uma mensagem de erro no cliente, assim como fechado o seu descritor de escrita do pipe.

Para facilitar a gestão, foi criado duas estruturas do tipo

`enum`

, nomeadamente para saber qual o estado de processamento da tarefa e qual o tipo de execução.

Em relação a este último, salienta-se a sua importância pois o sistema desenvolvido permite dois tipos de excução: quando o servidor tem recursos suficientes para executar de uma só vez (através de uma *pipeline*) - execução completa ou quando tal não é possível, aplica um filtro de cada vez. Neste caso, é criado um ficheiro temporário para ser utilizado na iteração seguinte.

2.1.3 Queue

```
struct queue
{
    Task *tasks;
    int current;
    int pending;
    int total;
    int size;
} NQueue;
typedef struct queue *Queue;
```

O conceito inicial **Queue** seria uma estratégia do tipo FCFS (*First Come First Served*). No entanto, ao longo da realização do trabalho, tal como descrito na literatura, esta estratégia não é, de todo, eficiente. Este tópico será discutido na secção 2.1.5.

Através desta estrutura, é possível ter conhecimento de quantas tarefas estão pendentes, total (em precessamento e pendentes) e tarefa atual. Com a utilização da API, ver 4.3, é possível fazer a gestão da lista de espera através de funções para adicionar, atualizar, remover e obter tarefas.

2.1.4 Exec Helper

Com apenas duas funções, esta API permite a execução total ou parcial de uma tarefa.

É utilizada pela Pool (ver 2.1.5) para apoio na execução de tarefas pendentes, mediante o tipo de execução associada (*PARTIAL vs FULL*).

Para mais informação, ver API 4.4.

2.1.5 Pool

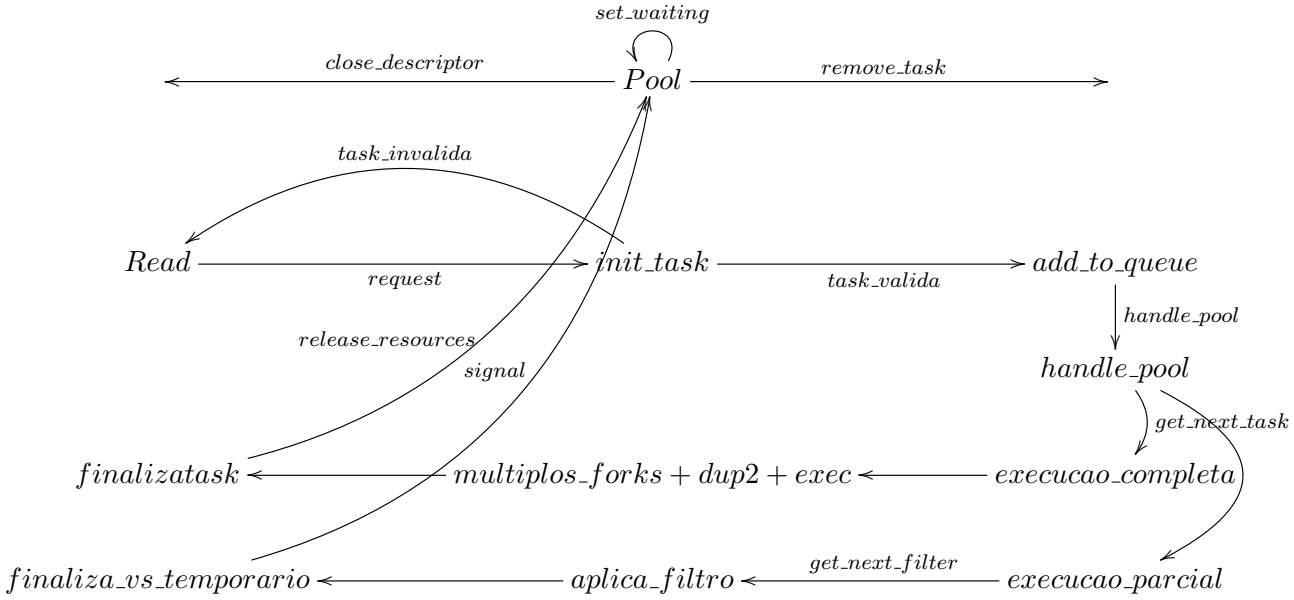
O módulo **Pool** serve como um controlador. Permite fazer a gestão dos processos que o cliente pede, desde a criação de *tasks*, verificação se são válidas e adição à *queue* até à execução de processos de forma concorrente.

```
static Pool POOL;

struct pool
{
    Queue queue;
    Config_Server config;
} PNode;
```

Como já abordado, a abordagem inicial seria uma estratégia do tipo FCFS. No entanto, devido ao entusiasmo e vontade de desenvolver uma solução mais completa, verificou-se que tal estratégia não seria compatível. Pelo menos de uma forma eficiente.

Assim, utiliza-se uma variação da estratégia FCFS, i.e. o servidor tenta executar uma tarefa assim que uma nova é recebida. Se for a sua vez, tenta executá-la. Caso não tenha recursos, procura a tarefa seguinte que seja possível de executar com os recursos disponíveis. Caso não seja possível executar mais nenhuma tarefa, volta para o *read* e aguarda por mais tarefas ou que sejam libertados recursos.



Através da estratégia utilizada, permite a execução, quer de pedidos que o servidor seja capaz de resolver de uma só vez (mesmo que esperando) pela sua disponibilidade, quer aqueles que nunca seria capaz e causaria *starving* caso entrasse na queue.

De forma sucinta:

- Recebe uma request
- Processa e cria uma *task*
- Verifica se a task é válida (caso não seja, responde ao cliente)
- Adiciona a task à queue
- Tenta executar a próxima task na queue
- Possível:

  - Verifica o tipo de excução (parcial ou total)
  - Executa e ocupa os recursos necessários
  - A execução é feita criando filhos para tal. ver 2.1.4
  - Quando estes terminam, enviam sinal ao pai a informar o seu término
  - O pai (pool) recebe a interrupção e verifica quem terminou
  - Vai à *queue* verificar de que task se trata
  - Verifica se terminou a query ou ainda há mais para processar
  - Coloca query como waiting no último caso
  - Remove-a da *queue* no primeiro
  - Liberta os recursos que a mesma estava a utilizar
  - Volta para o *handle_pool*

- Não possível
  - Volta para o modo bloqueado (*read*) a aguardar novas tasks OU interrupção de processos que terminaram e vão libertar recursos

2.2 Client - Aurras

A forma de efetuar pedidos ao servidor é através de um *Cliente* com as seguintes funcionalidades:

Ficheiro FIFO PID
Criado pipe com nome do PID do cliente
Pipe que será utilizado para receber estado de processamento do pedido
Abertura do extremo de leitura, ficando a aguardar em espera passiva

Comunicação Server SERVER_REQUEST
Abertura de extremo de escrita do pipe de comunicação com servidor
Utilizado para fazer o pedido

Gestão de sinais *SIGINT* e *SIGTERM*
É utilizado a função **handle_term** por forma a garantir que o ficheiro pipe gerado, é apagado aquando do término da execução do cliente. Mesmo que o término seja forçado.

Helper Caso a execução seja efetuada sem argumentos, imprime no ecrã o formato de pedidos aceite

Posto isto, o cliente recebe o pedido sob a forma de argumentos (na chamada da bash), podendo ser do tipo:

```
> aurras status
> aurras transform <input_file> <output_file> <filter_1> ... <filter_n>
```

O primeiro, irá apresentar o estado do servidor, nomeadamente quais as tarefas com estado WAITING, PENDING, PROCESSING, o número de instâncias em uso e número máximo das mesmas e o PID do servidor (ponto de entrada dos pedidos). Formato:

```
$ bin/aurras status
Task #1: status  PROCESSING
path filtros: bin/aurrasd-filters
alto aurrasd-gain-double: 0/2 (current/max)
baixo aurrasd-gain-half: 0/2 (current/max)
eco aurrasd-echo: 0/2 (current/max)
```

```
rapido aurrasd-tempo-double: 0/2 (current/max)
lento aurrasd-tempo-half: 0/2 (current/max)
pid: 1729
```

O segundo tipo, irá aplicar uma série de filtros pedidos ao input e armazenar o resultado no output:

```
$ bin/aurras transform samples/sample-1-so.m4a output_cenas.mp3 alto baixo
Processing 2 filters
Terminated
```

Conforme já abordado, o sistema desenvolvido permite a aplicação de filtros mesmo que o número de filtros exceda o limite de instâncias. Tal acontece através da criação de ficheiros temporários:

```
$ bin/aurras transform samples/sample-1-so.m4a output_cenas.mp3 alto baixo eco eco eco
Processing filter #1: alto
Processing filter #2: baixo
Processing filter #3: eco
Processing filter #4: eco
Processing filter #5: eco
```

Os ficheiros temporários gerados são eliminados assim que deixam de ser necessários. Notavelmente, é uma solução menos eficiente pois implica persistência em disco entre cada aplicação de filtro.

2.3 Server - aurrasd

Efetua a inicialização do servidor, criando todos os recursos necessários para a gestão de pedidos, a saber:

POOL_PIPE Cria um pipe com nome para comunicação com a instância pool
Efetuado fork()
Filho executar a pool (instância que faz a gestão dos pedidos)

comuniação cliente REQUEST_SERVER
Cria o ficheiro fifo que aceita os pedidos de execução dos clientes
Abre, ainda, o extremo de escrita do pipe, por forma a ficar aberto durante toda a execução do servidor, garantindo uma espera passiva

read Fica em espera passiva por escritas no pipe de comunicação com o cliente-servidor

A inicialização do processo *aurrasd* compreende ao seguinte comando:

```
> aurrasd <config_file> <filters_folder>
```

config_file $\langle nome\_filtro \rangle \langle ficheiro\_filtro \rangle \langle max\_instancias \rangle$
cada linha deve ter a estrutura indicada
É carregado na configuração do servidor. Ver 2.1.1

filters_folder path para pasta com os filtros
cada filtro tem um ficheiro associado que deve estar no caminho indicado

Caso se tente executar o servidor sem os argumentos necessários (seja a mais ou a menos), é retornada uma mensagem de erro e o mesmo é terminado.

```
$ bin/aurrasd op1
helper:
> aurrasd <config_file> <filters_folder>

$ bin/aurrasd op1 op2 op3
helper:
> aurrasd <config_file> <filters_folder>
```

3 Considerações Finais

Através da construção do **Serviço de Processamento de Áudio**, foi possível fortalecer conceitos desenvolvidos ao longo da unidade curricular (UC) **Sistemas Operativos**. Não obstante, foi também importante todo o conhecimento adquirido ao longo do curso de **Engenharia Informática**, nomeadamente estruturas de dados e modularidade.

A principal dificuldade durante a realização do **Serviço** foi a arquitetura da aplicação, definir a forma de comunicação entre cliente e servidor e gestão dos processos em execução, pendentes, em espera. Depois de se conseguir definir a estratégia, a sua implementação foi relativamente linear, sem grandes dificuldades.

No entanto, a aplicação desenvolvida apresenta algumas limitações, sejam elas:

sinais concorrentes
: limite da fila de espera de sinais concorrentes
pode levar a comportamento errático
não aconteceu durante os testes executados

máximo de tarefas
: o máximo de tarefas na queue, encontra-se limitada a 100
no entanto, esta limitação é facilmente ultrapassável
bastaria implementar uma queue com tamanho dinâmico, em vez de estático

término 'elegante'
: garantir que, quando o servidor recebe um sinal do tipo *SIGTERM* permite as tarefas pendentes terminarem, não aceitando mais
Foi implementada uma solução que não foi possível fazer debug, apesar de, conceptualmente, ser funcional

Apesar destas limitações, a solução desenvolvida encontra-se funcional e utilizou-se os conceitos alvo da UC em estudo, seja:

- Fila de espera
- Utilização de esperas passivas
- Criação de filhos
- Direcionamento de descritores
- Gestão de processos
- Processos concorrentes
- ...

Desta forma, acredita-se que se atingiu os objetivos da UC de forma satisfatória.

4 Anexos

4.1 Config API

```
#ifndef CONFIG_H
#define CONFIG_H

/**
 * @brief Estrutura a ser utilizada para armazenar a configuração do servidor, nomeadamen
 *        <nome filtro>
 *        <nome_ficheiro_execução>
 *        <instâncias_em_execução>
 *        <maximo_instâncias>
 *
```

```
 */
typedef struct config_server *Config_Server;

/**
 * @brief Inicializa uma configuração de servidor com espaço para 2 filtros
 *
 * @return Config_Server
 */
Config_Server init_config_server();

/**
 * @brief Devolve a pasta dos filtros
 *
 * @param cs Configuração do servidor
 * @return char* Path dos filtros
 */
char *get_filters_folder(Config_Server cs);

/**
 * @brief Define a path para os filtros
 *
 * @param cs Configuração do servidor
 * @param file Path da folder dos filtros
 */
void set_filters_folder(Config_Server cs, char *file);

/**
 * @brief Verifica se um filtro existe na configuração do server
 *
 * @param cs Config_Server
 * @param filter Filtro a procurar
 * @return int 1 se encontrar, 0 caso contrário
 */
int has_filter(Config_Server cs, char *filter);

/**
 * @brief Retorna o número de filtros disponíveis
 *
 * @param cs Configuração do servidor
 * @return int Número de filtros
 */
int get_total_filters(Config_Server cs);

/**
 * @brief Retorna o caminho para o filtro a aplicar
 *
 * @param cs Configuração do servidor
 * @param filter Filtro a ser aplicado
 * @return char* Caminho para o filtro (file)
 */
char *get_filter_path(Config_Server cs, char *filter);

/**
```

```
 * @brief Retorna o máximo de processos em execução para um determinado filtro
 *
 * @param cs Configuração do server
 * @param filter Filtro a procurar
 * @return int Número máximo de processos em execução (0 se não existir filtro)
 */
int get_max_filter(Config_Server cs, char *filter);

/**
 * @brief Adiciona uma nova instância em execução
 *
 * @param cs Configuração do servidor
 * @param filter Filtro
 * @return int 1 caso num processos em excução < maximo permitido
 */
int add_inuse_process(Config_Server cs, char *filter);

/**
 * @brief Adiciona uma nova instância em execução
 *
 * @param cs Configuração do servidor
 * @param filter Filtro
 * @param number Número de instâncias que vão ser utilizadas
 * @return int 1 caso num processos em excução < maximo permitido
 */
int update_inuse_process_size(Config_Server cs, char *filter, int number);

/**
 * @brief Retorna o número de processos que estão em execução para um determinado filtro
 *
 * @param cs Configração do servidor
 * @param filter Filtro
 * @return int >= 0 se processo existir; -1 caso contrário
 */
int get_inuse_filter(Config_Server cs, char *filter);

/**
 * @brief Adiciona um novo filtro, utilizando uma linha com formato:
 *        <nome_filtro> <ficheiro> <max_isntâncias>
 *
 * @param cs Configuração do servidor
 * @param line
 * @return int
 */
int add_filter(Config_Server cs, char *line);

/**
 * @brief Efetua parsing de um ficheiro de configuração, em que cada linha tem o formato:
 *        <nome_filtro> <ficheiro> <max_isntâncias>
 *
 * @param fd Descritor do ficheiro a ser efetuado parsing
 * @param cs Configuração do servidor
 * @return int Número de filtros adicionados
```

```
 */
int parseConfigLines(int fd, Config_Server cs);

/**
 * @brief Retorna o ficheiro de execução associado ao filtro
 *
 * @param cs Coniguração do servidor
 * @param filter Filtro
 * @return char* Nome do ficheiro a ser utilizado no exec
 */
char *get_filter_file(Config_Server cs, char *filter);

/**
 * @brief Verifica se um determinado filtro está disponível
 *
 * @param cs Configuração do servidor
 * @param filter Filtro a verificar
 * @param need Quantas unidades precisa
 * @return int 1 se true, 0 caso contrário
 */
int is_filter_available(Config_Server cs, char *filter, int need);

/**
 * @brief Mostra o estado da configuração do servidor
 *
 * @param c
 */
void show_config_status(Config_Server cs);

#endif
```

4.2 Task API

```
#ifndef TASK_H
#define TASK_H

typedef struct task *Task;

typedef enum taskstatus
{
    PENDING,
    WAITING,
    PROCESSING,
    FINISHED,
    ERROR
} Status;

typedef enum execute_type
{
    FULL,
    PARTIAL
} ExecuteType;

/**
```

```
 * @brief Inicializa uma tarefa
 *
 * @param request Comando com a tarefa completa
 * @return Task Tarefa inicializada
 */
Task init_task(char *request);

/**
 * @brief Retorna o set de filtros unitários
 *
 * @param t Tarefa
 * @return char** Set de filtros isolados
 */
char **get_task_filter_set(Task t);

/**
 * @brief Retorna o contador de filtros necessários
 *
 * @param t Tarefa
 * @return int* Lista de contador
 */
int *get_task_filter_counter(Task t);

/**
 * @brief Retorna o tamanho do set de filtros
 *
 * @param t Tarefa
 * @return int Tamanho do set de filtros
 */
int get_task_filter_size(Task t);

/**
 * @brief Retorna o pid do cliente que pediu a tarefa
 *
 * @param t Tarefa
 * @return char* Pid
 */
char *get_task_pid(Task t);

/**
 * @brief Retorna o tipo de execução da tarefa
 *
 * @param t Tarefa
 * @return ExecuteType Tipo de execução
 */
ExecuteType get_task_execution_type(Task t);

/**
 * @brief Define o tipo de execução da tarefa
 *
 * @param t Tarefa
 * @param type Tipo de execução
 */
```

```c
void set_task_execution_type(Task t, ExecuteType type);

/**
 * @brief Retorna o pid do filho que está a executar a tarefa
 *
 * @param t Task
 * @return int Pid do filho a executar
 */
int get_task_executer(Task t);

void set_task_executer(Task t, int pid);

/**
 * @brief Retorna o comando a ser executado: [status, transform]
 *
 * @param t Task
 * @return char* Comando, caso esteja definido!
 */
char *get_task_command(Task t);

/**
 * @brief Retorna o path para o ficheiro de input
 *
 * @param t Tarefa
 * @return char* path do ficheiro de input
 */
char *get_input_file(Task t);

/**
 * @brief Retorna o path para o ficheiro de destino
 *
 * @param t Tarefa
 * @return char* Path de ficheiro de output
 */
char *get_output_file(Task t);

/**
 * @brief Retorna o filtro que aguarda processamento
 *
 * @param t Tarefa
 * @param current Onde vai ser armazenado o próximo filtro a executar
 * @return índice do filtro em processamento / aguarda
 */
int get_current_filter(Task t, char **current);

/**
 * @brief Retorna o próximo filtro a processar (igual ao current mas incrementa)
 *
 * @param t Tarefa
 * @param next Onde vai ser armazenado o próximo filtro a executar
 * @return indíce do filtro que inicia processamento
 */
int get_next_filter(Task t, char **next);
```

```
/**
 * @brief Get the previous filter object
 *
 * @param t
 * @param next
 * @return int
 */
int get_previous_filter(Task t, char **next);

/**
 * @brief Incrementa o apontador para o próximo filtro
 *
 * @param t
 */
void increment_filter(Task t);

/**
 * @brief Retorna o estado de uma tarefa
 *
 * @param t Tarefa
 * @return Status Estado
 */
Status get_task_status(Task t);

/**
 * @brief Define o estado de uma tarefa
 *
 * @param t Tarefa
 * @param status Estado
 */
void set_task_status(Task t, Status status);

/**
 * @brief Retorna o total de filtros a serem aplicados
 *
 * @param t
 * @return int
 */
int get_task_total_filters(Task t);

/**
 * @brief Abre o descritor para o fifo do cliente
 *
 * @param t
 */
void open_task_client_fd(Task t);

/**
 * @brief Get the task client fd
 *
 * @param t
 * @return int
```

```
 */
int get_task_client_fd(Task t);


char **get_task_filters(Task t);

/**
 * @brief Print da tarefa
 *
 * @param t Tarefa a imprimir
 */
void show_task(Task t);

#endif
```

4.3 Queue API

```
#if !defined(QUEUE_H)
#define QUEUE_H
#include "task.h"

#define MAX_TASKS 100

typedef struct queue *Queue;

/**
 * @brief Inicializa uma queue com espaço para #MAX_TASKS tasks
 *
 * @return Queue queue inicializada
 */
Queue init_queue();

/**
 * @brief Adiciona uma task no próximo local disponível
 *        incrementa o #tarefas pendentes
 *        incrementa total tarefas
 *
 * @param q Queue
 * @param request <pid> <type> <input> <output> [filters]
 * @return int 0 se não foi possível inserir: queue cheia ou não inicializada
 *         1 caso contrário
 */
int add_task(Queue q, Task request);

/**
 * @brief Retorna a próxima tarefa a ser executada
 *        define o estado da task para PROCESSING
 *        decrementa os pendentes
 *        coloca apontador para próxima tarefa
 * @param q
 * @return Task
 */
Task get_next_task(Queue q);
```

```
/**
 * @brief Define o estado de uma task a partir do seu pid
 *        se a task já tinha o estado a alterar, não faz nada
 *        se alterar para WAITING ou PENDING, incrementa pending da queue
 *
 * @param q Queue
 * @param pid Pid da task a ser alterar
 * @param status Estado a definir
 * @return int 1 se fez alteração
 *             0 caso contrário
 */
int set_status_task(Queue q, int pid, Status status);

/**
 * @brief Coloca no apontador recebido por argumento, a task procurada por pid
 *
 * @param q Queue
 * @param pid Pid da task a procurar
 * @param t Apontador onde vai ser 'colocada' a task, caso exista (ou NULL)
 * @return int índice onde se encontra a tarefa
 */
int get_pid_task(Queue q, int pid, Task *t);

/**
 * @brief Retorna a tarefa através do pid de quem executou
 *
 * @param q Queue
 * @param pid Pid de quem executou
 * @return Task Tarefa encontrada ou NULL
 */
Task get_executer_task(Queue q, int pid);

/**
 * @brief Remove uma tarefa a partir do pid
 *        decrementa o total de tarefas
 *        APENAS REMOVE SE A TAREFA TIVER ESTADO FINISHED
 *
 * @param q Queue
 * @param pid pid da task a remover
 * @return int 1 se foi possível remover
 *             0 caso contrário
 */
int remove_pid_task(Queue q, int pid);

/**
 * @brief Retorna o número de tarefas pendentes
 *
 * @param q Queue
 * @return int Tarefas pendentes
 */
int get_pending_tasks(Queue q);

/**
```

```
 * @brief Retorna o total de tarefas ainda na queue (seja em que estado for)
 *
 * @param q Queue
 * @return int Total
 */
int get_total_tasks(Queue q);

/**
 * @brief Incrementa o número de tarefas pendentes
 *
 * @param q Queue
 */
void add_pending_tasks(Queue q);

/**
 * @brief Apresenta a queue
 *
 * @param q Queue a ser apresentada
 */
void show_queue(Queue q);

#endif // QUEUE_H
```

4.4 Exec Helper API

```
#if !defined(EXEC_HELPER_H)
#define EXEC_HELPER_H

#include "task.h"
#include "config.h"

/**
 * @brief Execução total de uma tarefa, utilizando filhos e redirecionamento
 *        de descritores através de dup2
 *
 * @param cs Configuração do servidor (para obter os ficheiros e path dos filtros)
 * @param t Tarefa a executar
 * @return int
 */
int exec_full(Config_Server cs, Task t);

/**
 * @brief Execução parcial de uma tarefa, utilizando ficheiros temporários em tmp/
 *        Obtém o próximo filtro a executar e, através de redirecionamento de descritores
 *        cria um filho que aplica o filtro atual
 *
 * @param cs Configuração do servidor
 * @param t Tarefa
 * @return int
 */
int exec_partial(Config_Server cs, Task t);

#endif // EXEC_HELPER_H
```

4.5 Pool API

```
#if !defined(POOL_H)
#define POOL_H

/**
 * @brief
 *
 * @param config_file
 * @param filter_folder
 * @return int
 */
int pool(char *config_file, char *filter_folder);
#endif // POOL_H
```

4.6 Client API

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
#include <string.h>
#include "aurras.h"
#include "dup_aux.h"

#define REQUEST_PIPE "SERVER_REQUEST"

#define BUFFER_SIZE 4096

char self_pid[10];

void handle_term(int status)
{
  unlink(self_pid);
  _exit(status);
}

int main(int argc, char *argv[])
{
  signal(SIGINT, handle_term);
  signal(SIGTERM, handle_term);
  if (argc > 1)
  {
    int fd, bytes_read;
    char buffer[BUFFER_SIZE];

    char self_pid[10];
    sprintf(self_pid, "%d", getpid());
    create_fifo(self_pid);

    if ((fd = open(REQUEST_PIPE, O_WRONLY)) < 0)
    {
```

```
      perror("error fifo write");
    }

    strcat(buffer, self_pid);
    strcat(buffer, " ");

    for (int i = 1; i < argc; i++)
    {
      strcat(buffer, argv[i]);
      strcat(buffer, " ");
    }

    write(fd, buffer, strlen(buffer)); // faz request ao server

    close(fd);

    open_dup(self_pid, O_RDONLY, 0666, 0); // abre o fifo para leitura
    while ((bytes_read = read(STDIN_FILENO, buffer, BUFFER_SIZE)) > 0)
    {
      write(STDOUT_FILENO, buffer, bytes_read); // faz request ao server
    }

    unlink(self_pid);
  }
  else
  {
    printf("helper:\n");
    printf("aurras status\n");
    printf("aurras transform <input_file> <output_file> <filter_1> ... <filter_n>\n");
    fflush(NULL);
  }

  return 0;
}
```