Relatório do Trabalho Prático de Sistemas Operativos

Aurras: Processamento de Ficheiros de Áudio

2020/2021

Grupo 1

José João Gonçalves - a93204
Maria Sofia Rocha Gomes - a93314
Miguel Rodrigues Santa Cruz - a93194

Índice

1. Introdução

No âmbito da Unidade Curricular de Sistemas Operativos lecionada no 2º semestre do 2º ano do Mestrado Integrado em Engenharia Informática, foi-nos proposta a realização de um projeto como forma de avaliação da componente prática da UC. Este projeto consiste na implementação de um serviço capaz de transformar ficheiros de áudio por aplicação de filtros, no qual o serviço a implementar é constituído por um servidor e um (ou mais) clientes que deverão comunicar via pipes com nome.

Existem diferentes tipos de filtros, e cada cliente solicita o processamento de um ficheiro de áudio através da aplicação no servidor de uma sequência de filtros. Para tal, o cliente passa como argumentos de linha de comando os nomes do ficheiro original e do ficheiro processado, assim como uma sequência de identificadores de filtros.

2. Descrição da Arquitetura

No desenvolvimento do projeto foi implementada uma arquitetura em que entre cliente(s) e servidor que se comunicam através de pipes com nomes. Para tal, temos o ficheiro **aurras.c** - que corresponde ao cliente - e o ficheiro **aurrasd.c** - que corresponde ao servidor.

O programa parte da inicialização do servidor, uma vez que este tem que estar disponível para o uso do cliente e à espera de tarefas. Posteriormente, o cliente pode começar a submeter tarefas, sendo estas enviadas via pipes com nome para o servidor, processadas pelo servidor, e finalmente devolvidas.

Para a conceção deste projeto foi utilizada a Makefile disponibilizada (com algumas alterações), assim como a estrutura de código fornecida.

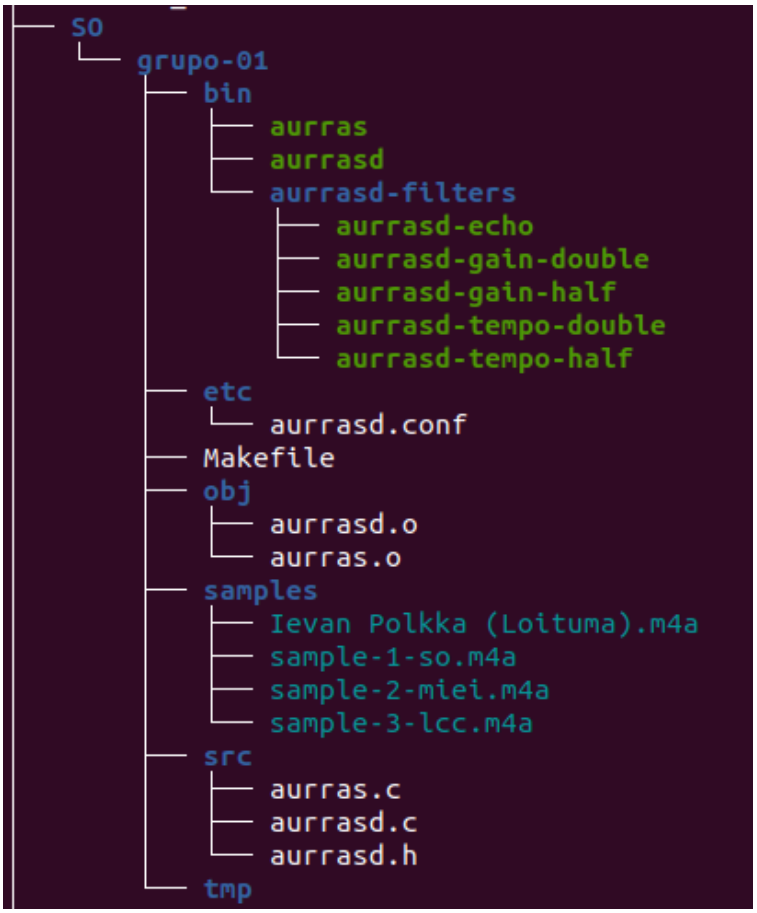

Figura 1. Estrutura da Aplicação

2.1 Cliente

O cliente pode interagir com o servidor pedindo para realizar uma tarefa ou para consultar o estado de informação. Para tal, temos comandos específicos, que são enviados pelo cliente, por exemplo:

- ***./aurras transform samples/sample-1.m4a output.m4a alto eco*** : para obter o ficheiro output.m4 através do ficheiro sample-1.m4a, mas submetendo este aos filtros alto e eco.
- ***./aurras status*** : para obter as informações do estado do processamento de cada um dos filtros.

Para podermos implementar estas funcionalidades recorremos, tal como descrito acima, a pipes com nome, sendo que para a primeira função é criado um pipe com nome e, deste modo, o pedido do cliente será escrito no pipe para chegar até ao servidor, onde será tratado.

Após submeter o pedido, a função **leResposta( )** abre o pipe que faz a ponte das informações entre o cliente e o servidor (**SENDTOCLIENT**) e lê para um buffer o pedido do cliente.

Para o tratamento de sinais, usamos a função ***handler*** *(int signum)* que controla o sinal SIGUSR1.

2.2 Servidor

Para o servidor, começamos por desenvolver a parte relativa aos sinais, sendo que o nosso handler trata e manipula os sinais de SIGPIPE e SIGINT.
Em seguida, com o objetivo de armazenar toda a informação e evitar o uso de variáveis globais, recorremos a uma **struct config,** em que armazenamos informações relativas a:

1. **char* filtersPath:** esta string corresponde ao path para a pasta onde se encontram os filtros (bin/aurras-filters).
2. **int * runningProcesses:** array de inteiros onde está armazenado o número de instâncias a correr para cada filtro, para poder comparar mais tarde com o máximo permitido.
3. **char** execFiltros:** cada índice contém o nome do ficheiro executável que implementa cada um dos filtros. **Ex:** aurras-gain-double
4. **int * maxInstancias:** array de inteiros com o número máximo de instâncias permitidas para cada filtro.
5. **char** identificadorFiltro:** cada índice deste array contém o identificador do filtro. **Ex:** alto, baixo, eco, ...

Para preencher esta struct, começamos por alocar o espaço necessário na struct config e, em seguida, recorremos à função **serverConfig** que começa por abrir o ficheiro aurrasd.conf e, através da função auxiliar **readline** (que lê linha a linha) preenche, para todas as instâncias da struct config, as informações deste ficheiro.
Assim, garantimos que se a primeira linha do ficheiro aurrasd.conf tiver as informações relativas ao filtro alto, o índice 0 de cada um dos componentes desta struct tem as informações relativas a este filtro, e assim sucessivamente para os restantes filtros.

Posteriormente, criamos dois *named pipes,* sendo que o FIFO tem o extremo de leitura no cliente e o de escrita no servidor, e de modo inverso, o SENDTOCLIENT tem o extremo de leitura no servidor e o de escrita no cliente.

Encoberto por um ciclo while(1) protegido pelo handler de ctrl+c, temos um ciclo que perdura enquanto houver bytes para ler do FIFO, sendo que aqui dentro são executados todos os pedidos do cliente. Neste ciclo, começamos por ler e fazer o parsing do request do cliente, através da função **splitWord** que recebe a string com o comando que o cliente enviou e o agrupa em uma lista de strings, alterando o identificador dos filtros para o nome do executável correspondente, com o path até lá.

Por exemplo, se o cliente enviar

./aurras transform samples/sample-1.m4a output.m4a alto

esta função retorna:

[[transform], [samples/sample-1.m4a], [output.m4a],
[../bin/aurrasd-filters/aurrasd-gain-double]]

Após esta função, conseguimos aceder diretamente ao primeiro elemento do array para verificar se o comando é válido. Caso este switch seja “transform”, passamos ao uso da função **executaTarefa**, que através dos argumentos obtidos a partir da função **splitWord** e a struct, aplica os filtros ao ficheiro de origem e devolve um novo ficheiro com com esses filtros e cujo nome é o que foi pedido pelo cliente.

O cliente tem a capacidade de saber o estado atual do servidor através do comando status, ou seja o resultado seja “status”. Esta operação é realizada pela função **sendStatus** que lê da estrutura de dados *struct config* o identificador do filtro, o número de filtros em execução e o número máximo de filtros que podem estar em execução. Também revela o identificador do processo através da função getpid().
Quando o cliente faz apenas “./aurras”, é mostrado na linha de comando os possíveis comandos que o cliente pode utilizar.

A função **sendProcessing( )** serve para enviar uma mensagem ao cliente de “processing”, pois este executou o comando “transform”.

Passamos então à explicação da função **executaTarefa**.

A função **executaTarefa** começa por fazer um **fork()**, sendo o processo filho resultante aquele que será responsável por criar tantos filhos quanto o número de filtros a aplicar através de um ciclo **for**.
No primeiro filho é redirecionado o ***standard input*** para o ficheiro de input.
Se apenas houver um filtro para aplicar é logo redirecionado o **standard output** para o ficheiro de saída, uma vez que não iremos precisar de um encadeamento de processos.

Quando é necessário aplicar mais que um filtro é criado um ficheiro temporário na pasta */tmp* que servirá de **standard output** para o filho atual e de **standard input** para o próximo filho. Em seguida é removido este ficheiro quando já não for necessário, fazendo com que não fiquem ficheiros desnecessários guardados quando já não são precisos. Cada processo filho faz um **execl** do caminho recebido como argumento.

Quando um processo inicializa, é enviado um sinal (*SIGUSR1*) ao processo pai que em seguida mostra essa informação ao utilizador. Quando acaba de se aplicar todos os filtros é enviado um sinal (*SIGUSR2*) ao processo principal, que avisa que terminou o processo.

3. Conclusão

Em suma, pensa-se que este projeto cumpre a maioria dos objetivos propostos, desde a comunicação entre cliente e servidor, passando pela aplicação dos filtros, até à manipulação de sinais.

As principais dificuldades que surgiram na realização deste trabalho centram-se na implementação de um controlo de pedidos, de maneira que pensamos que esta seja a maior limitação no projeto desenvolvido.

Para tentar implementar o comando ***status*** tentamos fazer com que os processos filhos se comuniquem com o processo principal através de pipes anônimos. No entanto isso obrigaria o processo pai a fazer uma espera ativa pelo fim dos processos, o que na prática limitava o servidor a executar uma tarefa de cada vez. Assim tentamos utilizar pipes com nomes mas voltamos a ter o mesmo problema. Por último, tentamos utilizar sinais mas com este método apenas sabemos quando começa e acaba a aplicação de cada filtro, não tendo informação acerca do filtro em concreto que está a ser aplicado.
Assim, não conseguimos implementar esta funcionalidade. (Uma ideia que tivemos mas não chegamos a implementar era a utilização de memória partilhada através da biblioteca ***sys/smh.c***).