Universidade do Minho

Sistema Operativos

Aurras: Processamento de Ficheiros de Áudio 2020/2021

[1]

Autores:

Vasco Oliveira, Nº93208

Diogo Matos, Nº93234

Miguel Fernandes, Nº94269

[1] https://tech-blogs.com/4-methods-to-combine-audio-files-into-windows-10/

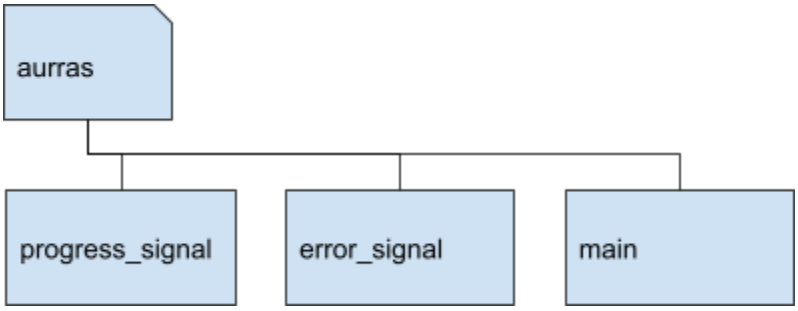
Introdução

Este programa tem como objetivo transformar ficheiros de áudio por aplicação de uma variedade de filtros, para além disso, permitir a submissão de ficheiros de áudio como também a consulta de tarefas em execução e o número de filtros em uso.

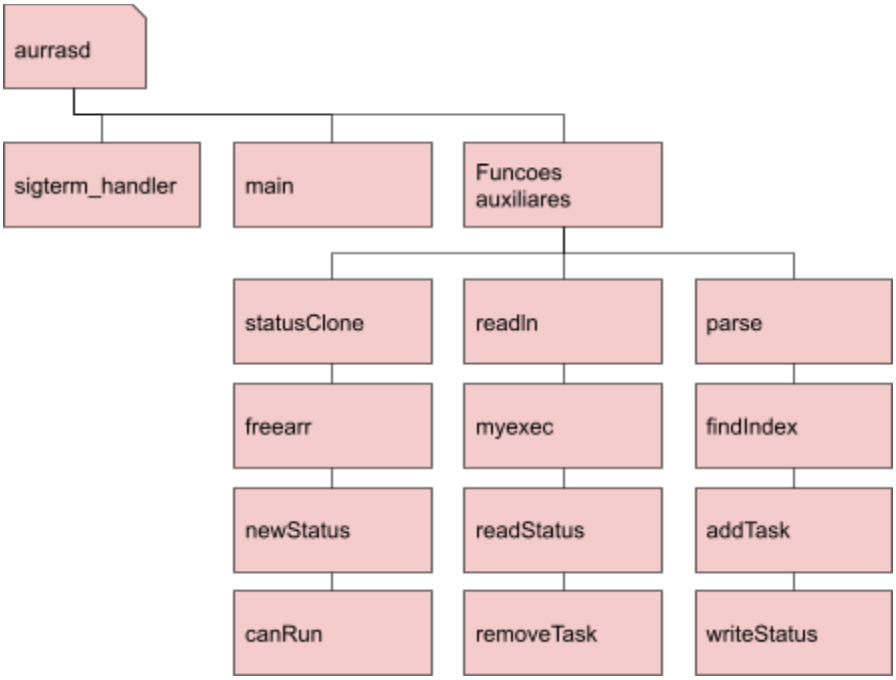
Para tal foram usados os conhecimentos adquiridos ao longo do tempo sobre a linguagem de programação C, bem como o conteúdo lecionado na cadeira de Sistemas Operativos.

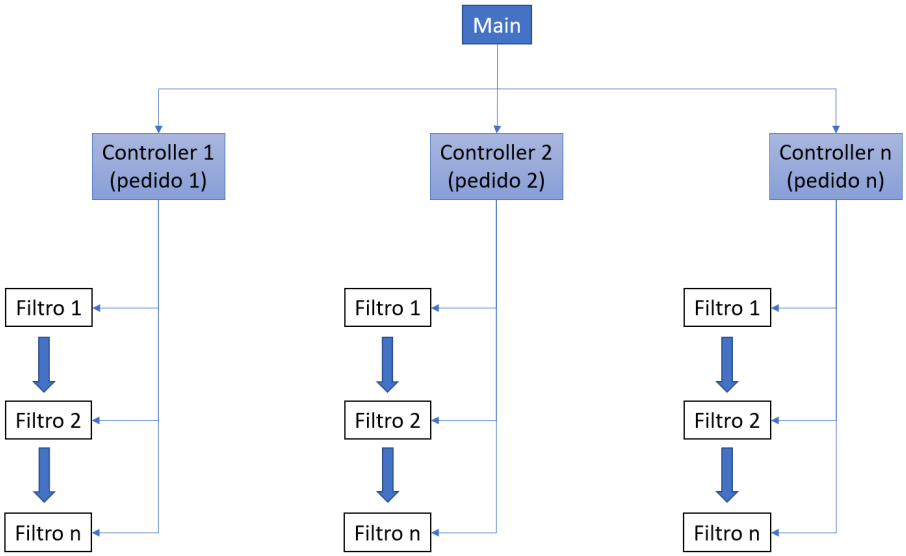
Arquitetura do programa

aurras



aurrasd






```c
typedef struct status
{
    int pid_server, num_filters;
    int *max, *running;
    char **filters, **filtersT, **tasks;
} * STATUS;
```

Descrição das diferentes funções e seus porquês

aurras

progress_signal

Função de processamento de sinal(SIGUSR1) que consoante a variável status(1,2 ou 3), imprime diferentes mensagens (“pending”, ”processing” e “finished”).

error_signal

Função de processamento do sinal(SIGUSR2) que imprime uma mensagem com o sinal recebido e está responsável pelo reenvio do pedido ao servidor.

main

A main é responsável por ler os pedidos do utilizador. Esta está preparada para receber um “status” e então ler o ficheiro status e imprimir o seu conteúdo, ou então receber um “transform” onde escreverá num **pipe com nome** o pedido registado. Esse pedido vai ser recebido e interpretado no aurrasd.

aurrasd

sigterm_handler

Função que lida com com o sinal SIGTERM, de modo a fechar o servidor de forma ‘amigável’. Apenas põe o valor 0 na variável global run, o que vai impedir a main de aceitar novos pedidos.

main

Esta main está à espera de receber pedidos via pipe com nome, quando o recebe, verifica se é possível correr, se não a main ‘mata-o’, se sim, este é adicionado ao STATUS, que atualiza o ficheiro correspondente, e de seguida o pedido é executado.

readln

Função que lê uma linha de um ficheiro, ou seja, até aparecer o caráter ‘\n’.

parse

Função que faz ‘parse’ de uma string a partir de um delimitador recebido e guarde-a num array de arrays.

freearr

Função que liberta todos os elementos de um array.

myexec

Função encarregue de aplicar os filtros ao áudio original passado como argumento. Esta função tem o caso especial de um só filtro e também está preparada para receber mais que um. Para resolver o problema de executar mais que um filtro, usamos a questão 5 do

guião 5 para nos guiar fazendo algumas adaptações, isto é, fazer os redirecionamentos do ficheiro de entrada para stdin e de stdout para o ficheiro de saída.

findIndex

Função que procura uma string num array de strings e devolve o índice da ocorrência deste, ou -1 no caso de não encontrar.

newStatus

Função que cria um novo STATUS, lê o ficheiro de configuração e inicializa-o conforme este último.

addTask

Função que adiciona uma 'task' a um STATUS, aumentando a cláusula "running" dos filtros pedidos.

canRun

Função que testa se é possível correr a 'task' pedida pelo utilizador, isto é, se não ultrapassa o máximo permitido.

removeTask

Função que remove uma 'task' de um STATUS.

writeStatus

Função que imprime um STATUS no ficheiro status.

statusClone

Função que clona um STATUS e devolve-o.

Escolha e uso da arquitetura utilizada

Um dos requerimentos do projeto era que o servidor suportasse o processamento concorrente de pedidos. Para poder notificar o cliente do estado do seu pedido, foi

necessário criar um *controller* com essa responsabilidade, uma vez que a *main* não pode estar a aguardar pela finalização dos filtros. Essa decisão levou a um sério problema relativo a outro requerimento do projeto: limitar o número de instâncias do mesmo filtro. Este problema é discutido em mais detalhe na secção "Objetivos concluídos e Aspetos a melhorar". Outra decisão que tomamos foi processar os pedidos em *batches* devido ao mesmo problema.

Para guardar os identificadores dos filtros, comandos externos correspondentes, número máximo de instâncias e número de instâncias que estão a correr concorrentemente, assim como o *process id* do servidor e as *tasks* que está a executar, usamos a *struct status*.

O loop da *main* consiste principalmente num "while(run == 1)", que é executado até o servidor receber um SIGTERM. A partir desse momento, o servidor não aceita novos pedidos e elimina ficheiros temporários que teve que criar.

Como referido na descrição da função *myexec*, esta está encarregue da aplicação dos filtros ao ficheiro original, fazendo-o de uma maneira semelhante, e fortemente baseada em, à questão 5 do guião 5.

O papel do cliente, de uma maneira simplificada, é apenas concatenar os filtros do pedido e escrevê-lo no pipe com nome criado pelo servidor para estabelecer essa comunicação de pedidos. Caso o processo ocorra sem erros, o cliente recebe vários sinais indicativos do estado dele, ou, caso contrário, recebe um sinal diferente e procede a reenviar o pedido.

Escolha e uso dos mecanismos de comunicação

Ao longo do trabalho utilizamos as várias técnicas aprendidas em SO para responder às necessidades criadas durante a realização do mesmo.

Para a comunicação entre servidor e utilizador utilizamos pipes com nomes para pedidos de transformação, ficheiros para os pedidos de status e ainda usamos sinais para indicar os vários estados do pedido.

O pipe com nome foi usado como uma *queue*, uma vez que não seria preciso revisitar os pedidos recebidos anteriormente. O status está guardado num ficheiro para ser acessível a qualquer cliente que o queira aceder, a qualquer altura. Para os vários estados do pedido usamos sinais, já que não é preciso transferir dados entre servidor e o cliente - indicar uma mudança no estado é suficiente.

Para a aplicação encadeada de filtros utilizamos pipes anônimos, em semelhança ao exercício resolvido na aula. Estes foram usados como um mecanismo intermediário entre a aplicação dos vários filtros, por exemplo na aplicação de 2 filtros, primeiramente o stdin é redirecionado para o ficheiro de áudio original, e o áudio com o primeiro filtro é passado pela extremidade de escrita, onde de seguida o stdout é redirecionado para o ficheiro de saída passado por argumento e utiliza o áudio recebido na extremidade de leitura para aplicar o segundo filtro.

Objetivos concluídos e Aspetos a melhorar

Ao longo do processo de realização do trabalho, deparamo-nos com várias dificuldades, algumas destas conseguimos superar e outras, infelizmente, não conseguimos arranjar maneira de “dar a volta por cima”.

Quanto aos pedidos de transformação de filtros, é possível aplicar vários filtros a um aúdio, e o servidor, como pedido no enunciado, também consegue executar vários pedidos em execução simultaneamente. Outro objetivo mencionado no enunciado e que conseguimos concluir com sucesso foi o servidor tratar o SIGTERM como uma forma graciosa de terminar os processos em execução e pendentes e rejeitar novos pedidos.

Porém, algo que não está operacional é a questão de limitar o número de instâncias do mesmo filtro:

- Para notificar o cliente do progresso do seu pedido, usamos um *controller*. Só este é que sabe quando a aplicação dos filtros termina, portanto, está responsável por retirar a *task* do STATUS.

- No entanto, o STATUS não é partilhado pela *main* e pelo *controller* uma vez que estes têm uma relação pai-filho e, consequentemente, regiões de memória diferentes. Isto significa que 1. o *controller* tem que escrever o STATUS atualizado no ficheiro correspondente, ou 2. informar, de alguma maneira, a *main* de que o pedido foi finalizado.
- Em primeiro lugar, dando ao *controller* o privilégio de escrever o STATUS no ficheiro, significa que ao existirem vários pedidos a serem tratados ao mesmo tempo, existem vários *controllers* a escrever no mesmo ficheiro ao mesmo tempo, o que dá asneira.
- Portanto, só a segunda opção seria viável, usando, simultaneamente, pipes e sinais. Isso introduziria uma complexidade que não tivemos tempo de resolver.
- O que fizemos foi usar processamento por *batches*, por exemplo: 3 pedidos simultâneos chegaram ao servidor; o servidor cria os *controllers* para os 3 pedidos, rejeitando aqueles que não podem ser processados no momento (nós criamos a função *canRun* que faz isso mesmo, porém esta não está 100% operacional o que nos leva a aceitar todos os pedidos feito pelos utilizadores); imediatamente faz *reset* do STATUS; e espera por uma nova *batch*.

Em relação aos ficheiros temporários criados, apenas são criados dois, um deles é um pipe com nome que serve, como dito anteriormente, para fazer a comunicação de pedidos entre utilizador e servidor, e o outro é o ficheiro que contém o estado do servidor.

Além da função *canRun* não estar como desejaríamos, encontramos outro problema: como o *reset* do STATUS pode acontecer enquanto existem *tasks* ainda em execução, decidimos manter todas as *tasks* processadas, ao invés de apagar as *tasks* do *batch*, levando a que quando executamos *tasks* e estas acabam, output de "./aurras status" seja este:

```
Task #0  transform samples/sample-1-so.m4a teste9.mp3 ec
o baixo lento

Task #1  transform samples/sample-1-so.m4a teste7.mp3 al
to eco lento lento

filter alto: 0/2 (running/max)
filter baixo: 0/2 (running/max)
filter eco: 0/1 (running/max)
filter rapido: 0/2 (running/max)
filter lento: 0/1 (running/max)
```

Por fim, ainda implementamos uma pequena interface para ajudar o utilizador a saber que comandos tem disponível.

Conclusão

Em suma, achamos que foi um trabalho um pouco difícil e que requereu aprofundar alguns conhecimentos que não foram tão bem consolidados quer em aulas teóricas quer em práticas, algo que achamos positivo e principalmente enriquecedor.

Acrescentamos ainda que o facto de ser um trabalho que conseguimos “brincar” com os sons e fazer algumas coisas engraçadas, tornou o trabalho muito mais interativo e divertido de realizar.