

Sistemas Operativos (2º ano de Curso de MIEI)

Trabalho Prático

Aurras: Processamento de Ficheiros de Audio

José Luís Abreu Mendes

(A75481)

Bernardo Emanuel Magalhães Saraiva

(A93189)

Rui Filipe Coelho Moreira

(A93232)

17 de junho de 2021

Resumo

Este trabalho prático consiste em implementar um programa que permite que múltiplos clientes possam efectuar em simultâneo pedidos de serviço a um servidor. Um dos objectivos principais do trabalho é ganhar experiência na concepção de soluções de programas concorrentes e na sua implementação em linguagem C. Conseguimos no final obter um programa que corria um servidor capaz de satisfazer pedidos de vários clientes em simultâneo.

Conteúdo

1	Introdução	2
1.1	Estrutura do Relatório	2
2	Análise do Problema	3
2.1	Descrição informal do problema	3
2.2	Especificação de requisitos	3
3	Concepção/desenho da Resolução	4
3.1	Arquitectura dos processos	4
3.2	Mecanismos de comunicação	5
3.3	Estruturas	6
3.3.1	Request	6
3.3.2	Task	6
3.3.3	ListTasks	6
3.3.4	Answer	7
3.3.5	FiltersConfig	7
3.4	Tratamento das tasks em espera	7
4	Testes e Resultados	8
4.1	Testes realizados e Resultados	8
5	Conclusão	11

Capítulo 1

Introdução

1.1 Estrutura do Relatório

Este documento está organizado em 3 partes importantes. A primeira, no capítulo 2 , corresponde à análise do problema, que consiste implementação de um programa servidor capaz de gerir os pedidos de vários clientes, de forma concorrente. O servidor encarrega-se de aplicar a diversos ficheiros de input ,em formato de audio , vários filtros disponibilizados pela equipa docente. Numa segunda parte, no capítulo 3 , vamos passar à formulação da solução. Esta passa por explicar como as diferentes tarefas foram resolvidas e implementadas. Na terceira parte, no capítulo 4, vamos demonstrar a utilidade do programa que desenvolvemos, realizando testes para verificar se as soluções correspondem às expectativas.No capítulo 5 termina-se o relatório com uma síntese do que foi dito,as conclusões e o trabalho futuro.

Capítulo 2

Análise do Problema

2.1 Descrição informal do problema

O enunciado apresentando pretende que se construa um sistema que implemente um servidor capaz de fornecer serviços a clientes. Existem disponíveis dois tipos de serviço:

1. Submissão de pedidos de processamento de ficheiro de áudio
2. Obter o estado de funcionamentos do servidor

2.2 Especificação de requisitos

A implementação do sistema foi realizada tendo por base o sistema operativo Linux como ambiente de desenvolvimento e de execução. O sistema também foi testado no sistema operativo macOS, sendo o comportamento do programa semelhante ao verificado no ambiente Linux. No desenvolvimento do projecto foram usadas chamadas ao sistema, para a criação de processos, criação de pipes anónimos e com nome, leitura e escrita de ficheiros, etc. A linguagem utilizada foi o C.

Capítulo 3

Concepção/desenho da Resolução

O sistema está organizado em dois componentes principais: o servidor (aurrasd.c) e o cliente (aurras.c). De modo a permitir o funcionamento do sistema o cliente e o servidor tem de comunicar entre si. Além disso o servidor faz tarefas em simultâneo. Temos assim duas questões importantes que devemos abordar: como definir a arquitectura de processos e que tipo de mecanismos de comunicação foram usados.

3.1 Arquitectura dos processos

No lado do cliente apenas temos um processo e este é suficiente para a realização de todas as tarefas. Este cliente faz um pedido ao servidor e depois fica à espera das respostas do servidor.

Já no caso do servidor podemos ter vários processos a correr em simultâneo. O servidor fica bloqueado à escuta de pedidos do cliente, e sempre que recebe um, cria um processo filho para o tratamento desse mesmo pedido. Isto permite libertar o processo pai para continuar a ler pedidos de outros clientes. O servidor tem os seguintes tipos de processos filhos:

1. **Tratamento do Serviço do tipo Processamento de Áudio**
2. **Tratamento do Serviço do tipo Status**
3. **Receptor de término do Tratamento de Serviço do tipo Processamento de Audio** Na inicialização do servidor é criado um processo filho que será responsável por saber quando os processos de tratamento de Serviço terminam. Assim, um processo de tratamento de serviço antes de terminar envia ao receptor o seu pid a indicar que já fez a sua tarefa. O Receptor vai ser responsável por verificar se o servidor está disponível para analisar um tratamento de serviço que acabou. Caso esteja disponível então o receptor indica ao servidor um tratamento de serviço que acabou. Caso contrário o receptor espera que o servidor esteja disponível. O servidor, recebendo a indicação do término dum tratamento do serviço, faz as devidas atualizações relacionadas com esse tratamento.

O processo filho que designei anteriormente por Tratamento de Serviço do tipo de Processamento de Audio é responsável por reproduzir o efeito do comando que cria um ficheiro de audio com uma série de filtros aplicados ao ficheiro de audio dado como input. Os filtros são aplicados da forma que se mostra na imagem 3.2. Cada filtro é um executável que recebe como input um ficheiro de audio e produz um ficheiro de audio output, e cada tratamento do serviço aplica vários filtros, pelo que criamos um processo filho para cada um dos filtros executáveis que precisavamos.

Os processos indicados são todos os processos que tivemos que criar para além dos processos pais: do servidor e do cliente.

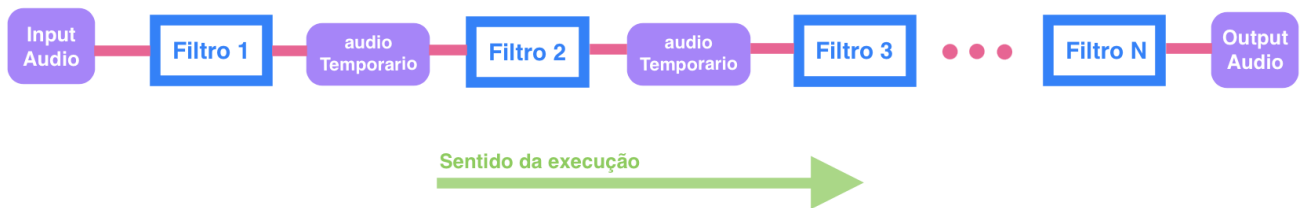


Figura 3.1: Processamento de um ficheiro áudio

3.2 Mecanismos de comunicação

Para poder comunicar entre processos usamos dois mecanismos de comunicação: pipes anónimos e pipes com nome. Segue-se os sítios onde usamos o mecanismo de pipes com nome:

1. **Comunicação entre cliente e servidor, sentido cliente para servidor** usamos um pipe com nome para que o cliente enviasse pedidos ao servidor. O nome deste pipe é conhecido por ambos os processos e é o servidor que o cria. O servidor fica bloqueado à espera que o cliente lhe envie um pedido. Na parte do cliente, este envia o pedido ao servidor. De notar que do lado do cliente tivemos de abrir o pipe para escrita e no lado do servidor abrimos para leitura e também para escrita. Caso não tivessemos aberto para escrita, e nenhum cliente estivesse aberto o read do pedido no lado do servidor daria EOF.
2. **Comunicação entre cliente e servidor, sentido servidor para cliente** O servidor tem de eventualmente dar uma resposta ao cliente referente ao pedido efectuado. Para isso o cliente inicialmente cria um pipe com nome específico para esse cliente. Nos decidimos identificar unicamente um cliente pelo pid do processo que corre o cliente. Assim estabelecemos uma ligação entre o servidor e um cliente específico. Isto foi feito desta forma porque uma resposta de um pedido tem de ser recebida pelo mesmo cliente que fez o pedido. Para o servidor abrir correctamente a extremidade de leitura do pipe de um cliente, este tem de saber o pid do processo cliente, e esta informação é enviada no pedido que vai no sentido cliente para servidor explicada no item anterior.
3. **Comunicação entre o receptor de tratamento de serviço e os processos que fazem o tratamento do serviço** Utilizamos um pipe com nome para que sempre que um processamento de audio vá acabar, este envie ao receptor algo que identifique que esse mesmo processamento de audio já acabou. O sentido do pipe é do processamento do tratamento de serviço para o receptor. Sempre que um processamento de audio tenha sido efectuado este receptor vai receber esta informação.
4. **Comunicação entre o receptor e o servidor** Utilizamos dois pipes com nomes nesta parte, uma para cada sentido. O servidor comunica ao receptor quando está pronto para receber informação de que um processamento acabou e o receptor quando recebe esta informação do servidor envia-lhe qual o processamento que acabou e um sinal a dizer que pode ler qual o processamento que acabou. Se o servidor não estiver disponível o receptor espera. O sinal usado foi o SIGUSR1

A criação do receptor foi necessário devido ao seguinte. Caso o processo que faz o processamento de audio comunicasse directamente com o servidor, se tivessemos dois processos a enviar o mesmo sinal ao mesmo tempo, e como a fila de sinais apenas pode ter um sinal do mesmo tipo então poderia dar o caso de dois processos terem acabado mas o servidor ter dado conta apenas de um.

Também fizemos uso dos pipes anónimos. Esta utilização está presente no processamento de audio quando executamos vários filtros num ficheiro de audio. Dados n filtros, tenho de criar $n-1$ pipes para permitir que o resultado da execução de um filtro seja usada como input no executável do segundo filtro. Tivemos de fazer os redirecionamentos de forma a permitir esta última ideia. No processo filho que executa o primeiro filtro tive de redirecionar o input para o ficheiro de input de audio e no processo que executa o último filtro tive de redirecionar o output para o ficheiro de áudio de output que eu pretendia. Feitos todos os redirecionamentos indicados e após a execução dos filtros com a chamada da system call exec nos vários processos filhos foi possível aplicar uma série de filtros a um ficheiro de áudio.

3.3 Estruturas

Para poder comunicar informação entre processos e para manter o estado do servidor foram criadas as seguintes estruturas.

3.3.1 Request

Estrutura que incorpora todas informações necessárias para criar o pedido no cliente. O cliente cria um request que é enviado para o servidor, e este último fica à espera até que uma estrutura deste tipo chegue momento em que pode ler o pedido. De notar que a estrutura Request tem um tamanho fixo. Caso o tamanho fosse variável quem recebe a mensagem não saberia quando é que a mensagem terminaria. As informações do Request apresentam-se a seguir:

```
struct request{
    int service;           /* Tipo de Serviço: Status ou processar audio */
    int pidProcess;        /* Pid do processo do cliente que fez o pedido */
    char arguments[MAXARGS][MAXSIZEARG]; /* Argumentos do comando inserido pelo cliente */
    int numberArguments;   /* Número de Argumentos do comando inseridos pelo cliente */
};
```

3.3.2 Task

Quando um pedido ou Request chega ao servidor, este pedido é transformado no servidor numa tarefa ou Task. A Task já tem mais informação do que o request informação essa que é necessária para o servidor gerir as tarefas em execução.

```
struct task{
    int numberTask;        /* Número de ordem da task */
    int pidProcess;        /* Pid do processo do cliente que fez o pedido */
    int numberTotalFilters; /* Numero total de filtros do config */
    char* comando;         /* Comando introduzido pelo cliente */
    char** execsFilters;
    int numberFiltersTask;
    char* inputFile;
    char* outputFile;
    int filtersRequired[];
};
```

3.3.3 ListTasks

A ListTasks permite agrupar um conjunto de tasks. O servidor faz uso desta lista de tasks por um lado para saber as tasks que estão em execução nesse momento e por outro lado para manter guardadas as tarefas que ficaram pendentes (tarefas que não puderam ser imediatamente processadas por falta de filtros). Definimos também neste caso uma estrutura que se apresenta a seguir.

```
struct listTaskSv{
    GPttrArray* listTasks;
    int numberTasks; // Numero de ordem da task
};
```

Estas duas listas usadas pelo servidor são dinâmicas, no sentido em que podemos estar a inserir tasks, remover uma task com determinado id, a lista pode crescer e diminuir de tamanho. Dado este factor usamos *Pointers Arrays* da biblioteca glib para definir estas listas.

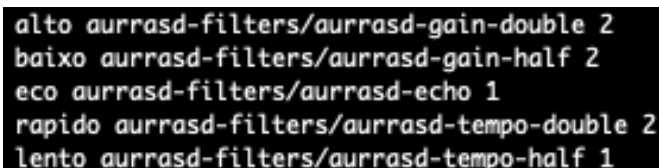
3.3.4 Answer

A Answer é a estrutura utilizada pelo servidor para enviar uma mensagem de resposta ao cliente. No caso de pedidos de processamento de ficheiro de áudio, o servidor pode responder com mensagens que tenham o conteúdo "processing" ou "pending". No caso do cliente fazer um request do tipo de serviço status, o servidor responde com a informação das Tasks que estão em execução juntamente com a ocupação dos filtros.

```
struct answer{
    char message[MAXSIZEMESSAGE]; /* mensagem que é enviada pelo servidor para o cliente */
};
```

3.3.5 FiltersConfig

Esta estrutura guarda a informação que está presente no ficheiro /etc/aurrasd.config. Este ficheiro tem informação sobre os filtros que existem, contendo os seus identificadores, os executáveis dos filtros correspondentes e número máximo de instâncias de executáveis de filtros que podem estar a correr ao mesmo tempo. O ficheiro config tem o seguinte aspecto:



```
alto aurrasd-filters/aurrasd-gain-double 2
baixo aurrasd-filters/aurrasd-gain-half 2
eco aurrasd-filters/aurrasd-echo 1
rapido aurrasd-filters/aurrasd-tempo-double 2
lento aurrasd-filters/aurrasd-tempo-half 1
```

Figura 3.2: Exemplo de um ficheiro de configuração dos filtros

A estrutura FiltersConfig contém ainda informação sobre os filtros que estão em execução permitindo facilmente verificar quantos filtros estão disponíveis de cada tipo.

```
struct filter{
    char *identificador; /* identificador do filter */
    char * executavel; /* nome do executável do respectivo filtro */
    int numeroMaxExecucao; /* número máximo de instâncias que podem estar a correr concorrentemente */
    int emExecucao; /* número de instâncias que estão a correr concorrentemente */
};

struct filtersConfig{
    GPtArray* filters; /* array com todos os filtros */
    int numberFilters; /* Numero de filtros */
};
```

3.4 Tratamento das tasks em espera

Quando as tasks ficam pendentes o servidor em algum momento trata de as colocar em execução. O servidor trata disto sempre que um processamento em execução acaba. Isto porque quando o processamento acaba então quer dizer que há filtros que passaram a ficar disponíveis. Assim quando um processamento acaba o servidor eventualmente recebe um sinal relacionado com este processamento e na rotina de tratamento tenho de atualizar os filtros disponíveis, remover a task da lista de tasks em execução e percorrer todas as tasks em espera para ver se elas podem ser colocadas em execução.

Capítulo 4

Testes e Resultados

4.1 Testes realizados e Resultados

Para verificar o correcto funcionamento do programa fizemos alguns testes. Para testar o programa temos que correr em primeiro lugar o servidor. Assim que o servidor estiver ativo podemos tomar a posição de vários clientes e enviar multiplos pedidos para o servidor. Existem dois comandos cuja estrutura é a seguinte:

```
./aurras transform transform [ficheiro input] [ficheiro output] [filter1] [filter2] ... [filterN]
```

Em que:

ficheiro input Nome do ficheiro de áudio que se pretende processar. Só são permitidos nomes de ficheiros que estejam na pasta samples

ficheiro output Nome do ficheiro que irá conter o resultado do processamento. Os ficheiros resultantes são colocados na pasta outputs.

Também existe o seguinte comando:

```
./aurras status
```

Pelo facto de não conseguirmos executar manualmente vários clientes em simultâneo, construímos uma script que executa vários comandos do cliente ao mesmo tempo. O ficheiro testes/testScript contém os comandos que serão executados e o ficheiro executeCommands.c contém o código que trata de executar os comandos. Na figura 4.1 apresenta-se um exemplo de um ficheiro para testes.

```
transform sample-1-so.m4a out1.mp3 alto baixo lento
transform sample-1-so.m4a out2.mp3 eco lento
status
transform sample-1-so.m4a out3.mp3 rapido baixo
transform sample-1-so.m4a out4.mp3 alto baixo lento
transform sample-1-so.m4a out5.mp3 alto baixo lento
status
transform sample-1-so.m4a out6.mp3 alto baixo lento
transform sample-1-so.m4a out7.mp3 eco lento rapido alto baixo
transform sample-1-so.m4a out8.mp3 eco
```

Figura 4.1: Exemplo de ficheiro com um conjunto de comandos

Explicação dos testes da figura 4.3 figura 4.2: Nas imagens que se seguem, encontra-se em demonstração o nosso script de testes que corre vários comandos num curto intervalo de tempo (quase instantaneamente) de forma a testar o controlo da concorrência do servidor. Para executar os comandos, é necessário correr o servidor, que fica à espera de comandos para serem executados. Quando o script é corrido são criados 8 processos que executam variados pedidos ao servidor de forma a testar todas as funcionalidades existentes. Como pode ser verificado nas screenshots tudo corre conforme previsto e sem erros e os ficheiros são gerados corretamente. De notar que alguns processos ficam pendentes pois não há filtros disponíveis naquele momento e mais tarde essas tarefas à espera são colocadas em execução.

```
saraiva@saraiva-OMEN:~/Desktop/trabalhoS0/grupo-xxx (Linux Intel)/grupo-45/bin$ ./execs
Já foram lidos todos os comandos da script de teste
processing
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)
pid: 89193
processing
pending
processing
pending
pending
pending
pending
Task #1: transform sample-1-so.m4a out1.mp3 alto baixo lento
Task #2: transform sample-1-so.m4a out3.mp3 rapido baixo
Task #4: transform sample-1-so.m4a out8.mp3 eco
filter alto: 1/2 (running/max)
filter baixo: 2/2 (running/max)
filter eco: 1/1 (running/max)
filter rapido: 1/2 (running/max)
filter lento: 1/1 (running/max)
pid: 89193
processing
processing
processing
processing
processing
saraiva@saraiva-OMEN:~/Desktop/trabalhoS0/grupo-xxx (Linux Intel)/grupo-45/bin$ ./aurras stop
saraiva@saraiva-OMEN:~/Desktop/trabalhoS0/grupo-xxx (Linux Intel)/grupo-45/bin$
```

Figura 4.2: Teste

```
saraiva@saraiva-OMEN:~/Desktop/trabalhoS0/grupo-xxx (Linux Intel)/grupo-45/bin$ ./aurras
sd
PID do servidor 89193
A task numero 4 terminou
A task numero 2 terminou
A task numero 1 terminou
A task numero 3 terminou
A task numero 5 terminou
A task numero 6 terminou
A task numero 7 terminou
A task numero 8 terminou
saraiva@saraiva-OMEN:~/Desktop/trabalhoS0/grupo-xxx (Linux Intel)/grupo-45/bin$
```

Figura 4.3: Teste

Finalmente conseguimos verificar pela figura 4.4 que os resultados do processamento foram colocados na pasta outputs.

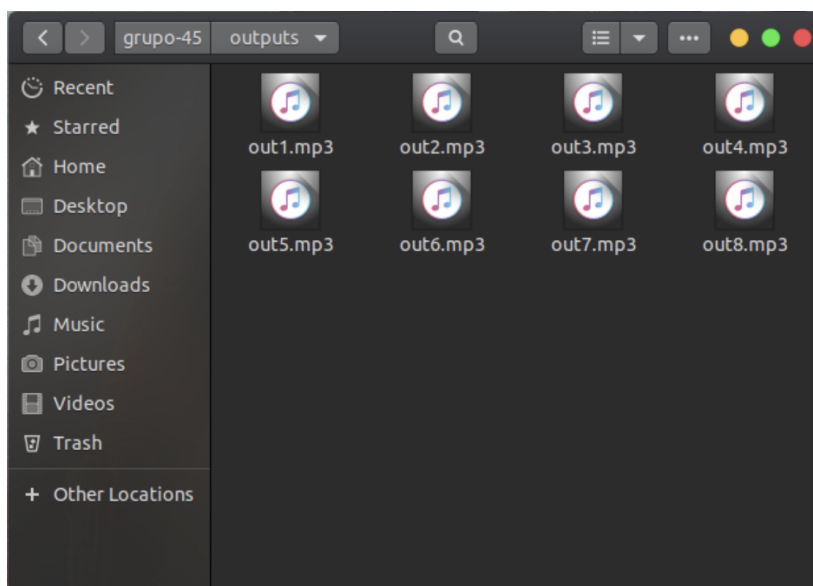


Figura 4.4: Resultados do processamento dos ficheiros de áudio

Capítulo 5

Conclusão

Dado os resultados dos testes pode-se concluir que o programa consegue ter um servidor a correr e multiplos clientes a pedir serviços. Tentamos colocar o nosso programa o mais eficiente possível, contudo houve alguns aspectos que poderíamos ter melhorado, como o caso do tratamentos das tasks em espera por parte do servidor. Caso o servidor fosse bombardeado com muitos pedidos dos clientes, o servidor quando tivesse de decidir quais as tasks em espera para para colocar em execução, poderia gastar muito tempo a analisar todas as tasks em espera e eventualmente poderia provocar deficiência no desempenho do servidor. Estas observações permitiram-nos desenvolver capacidades, que facilitarão a abordagem a problemas deste género no futuro.