Relatório - Sistemas Operativos

Grupo 99

Pedro Ferreira (A93282)
Ricardo Gama (A93237)
Rui Braga (A93228)

17 de junho de 2021

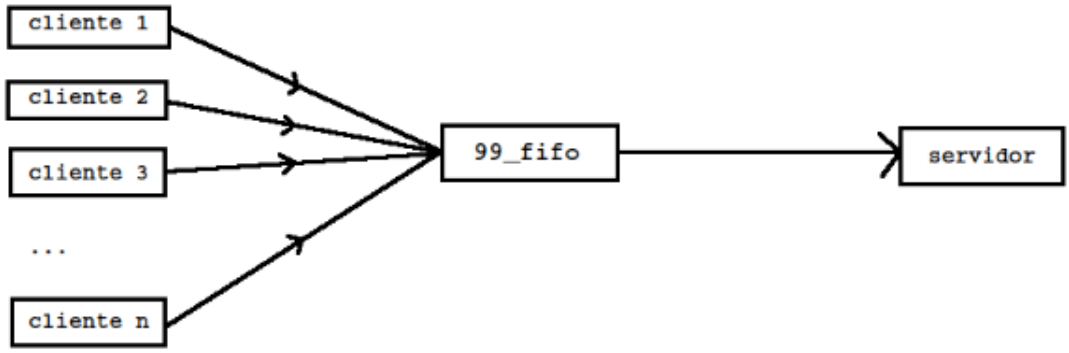
Conteúdo

1 Introdução

Neste projeto foi-nos proposto criar um serviço que transforme ficheiros de áudio através de uma sequência de filtros. Serviço esse que é composto por um programa servidor e por um programa cliente.

2 Comunicação entre servidor e cliente

O servidor começa por criar um pipe com nome, chamado `99_fifo` (somos o grupo 99), e é para aqui que todos os clientes enviam os seus pedidos.



2.1 Formato de um pedido

Para sincronizar devidamente os pedidos, optamos por formatar as strings desta maneira:

```
pid _ pedido(-_-)
```

Qual é o objetivo disto?

Pid:
Serve para o servidor poder reconhecer o processo cliente, e assim, cria um pipe, cujo nome é o seu pid. Desta feita, servidor e cliente podem comunicar sossegadamente sem "ruído"de outros processos.

Pedido:
É autoexplicativo. Se o cliente quiser saber o estado do servidor, o pedido será `status`, se quiser aplicar um filtro os argumentos são exatamente os mesmos que insere no programa cliente. Mas antes do pedido, existe o _ que divide o

pid e o pedido.

Barreira:
A barreira serve para separar pedidos diferentes. Muitas vezes acontece, que vários clientes sobrecarregam o servidor e para poder executar o desejo de cada um deles, o servidor tem que ser capaz de diferenciá-los. Para este efeito, precisávamos de algo que criasse um contraste gritante entre os pedidos, para não acontecerem efeitos indesejáveis ao sepára-los.

Exemplo do que um cliente envia ao `99_fifo`:

```
12345_transform input.mp3 output.mp3 eco rapido(-_-)
                54321_status(-_-)
```

3 Gestão de recursos no servidor

3.1 Estratégia de gestão de CPU

Para evitar que o servidor esteja em espera ativa, constantemente a perguntar se há novo conteúdo no fifo principal e a retirar recursos importantes à máquina, decidimos "interromper"o servidor recorrendo às funções `pause()` e a `alarm()`. Assim, mal o servidor é iniciado é agendado o envio de um `SIGALRM` e quando chega ao ciclo em que verifica se há algo para ler do pipe é interrompido com `pause()`, ficando então há espera de receber `SIGALRM`. Quando, enfim, o recebe, agenda o envio de outro `SIGALRM` para um segundo depois (inclusive, temos uma variável que conta estes segundos, mas não é importante aqui).

3.2 Estratégia de gestão dos processos filho

Escolhemos usar apenas 10 processos em concorrência para facilitar a gestão. A gestão dos processos filho varia com a exigência de recursos numa determinda altura.

Então, ao entrar um novo cliente, o servidor verifica se é possível executar o seu pedido, tendo em conta a capacidade máxima de filtros a utilizar, caso contrário, o cliente fica à espera até que o servidor receba um dos seus filhos que estão a executar outros pedidos. Caso seja possível, o servidor seguidamente, verifica se estão ativos os 10 processos, caso seja isto verdade, espera que um termine, se não, segue livremente.
Na eventualidade, do servidor não estar a receber nenhum pedido novo, ou os pedidos que está a receber não estejam a usar filtros necessários aos outros, a cada 10 segundos, o servidor espera que um processo termine, caso hajam processos ativos, claro está.

A nossa forma de monitorizar os processos que estão ativos consiste em ter um array de `int` em que cada um é o pid de um processo filho em execução, se o valor de um certo índice do array for 0, já sabemos que, no máximo, temos 9 processos ativos.

4 Execução dos pedidos na interface cliente

4.1 ./aurras

Nesta situação o cliente nem se chega a ligar ao servidor. Funciona como um botão de `HELP`.
É bastante simples, basta apenas, imprimir para o `STDOUT` de que forma é que se deve chamar o programa cliente.

4.2 ./aurras status

Para o utilizador obter o estado do servidor, o cliente envia o seu pedido ao fifo principal e cria um fifo, cujo nome é o seu pid, para o servidor o conseguir identificar.
Para poder reproduzir, fielmente, o seu estado, o servidor tem:
- uma matriz de strings, em que cada coluna refere a um filtro, ao executável do filtro e à sua capacidade máxima, respetivamente
- um array de inteiros que indicam os filtros que estão a ser utilizados
- um array de inteiros que indicam o número da tarefa que está a ser executada
- um array de strings em que cada uma tem o pedido que está a ser executado

Para gerar o output que o cliente quer, o servidor faz um `fork()` e redireciona o seu `STDOUT` para o pipe nomeado com o pid do cliente e escreve, então, para aí o seu `status`.
Depois, basta apenas o programa cliente ler a informação do pipe e mostrá-la ao utilizador até encontrar aquela que deve ser a última linha, que é a linha que indica o pid do servidor.
E termina aí a sua tarefa.

4.3 ./aurras transform input.m4a output.m4a filtro1 ...

Para transformar um ficheiro de áudio usando uma sequência de filtros, o programa cliente, inicalmente, faz exatamente o mesmo do que na situação de querer saber o estado do servidor. Envia o seu pedido ao fifo principal e cria um pipe, dando-lhe como nome o seu pid, tudo igual.
Seguidamente, o servidor recebe uma string com o pedido e divide-a nos seus vários argumentos, ignorando a parte que indica o que fazer ("transform"). Por exemplo:

[pedido]: transform input.m4a output.m4a eco
é transformado em
[argumento1]: input.m4a
[argumento2]: output.m4a
[argumento3]: eco

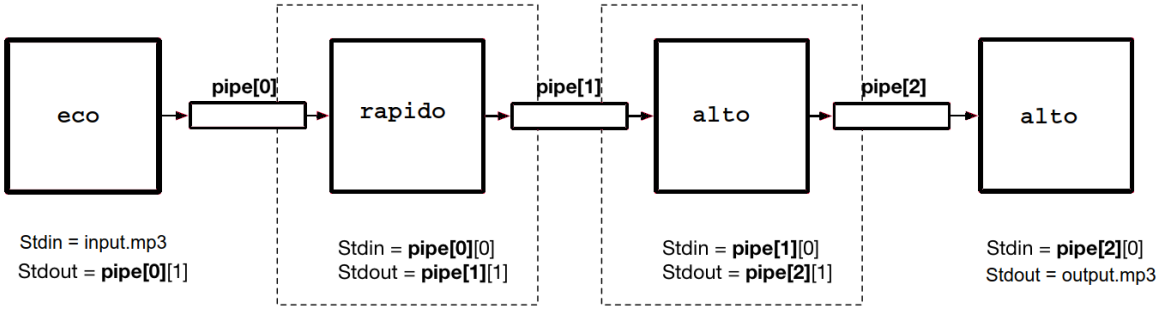
Assim que, já no servidor, o pedido receba a luz verde para iniciar a sua execução, o servidor liga-se ao pipe específico deste cliente e envia uma mensagem que indica o início do processamento. Aí o cliente, escreve no seu `STDOUT` `Processing...`.

De resto, a forma como tratámos o final do processamento do pedido ou caso tenha ocorrido um erro é idêntica, ao início do processamento. O servidor envia uma mensagem específica para cada caso e o cliente informa o utilizador do que quer que tenha acontecido.
Admitimos que esta talvez não seja a solução ideal, pois o programa cliente fica em espera ativa até haver algo novo escrito no pipe para poder processar.

Bem mais interessante do que o cliente faz enquanto a sua requisição está a ser efetuada, é o trabalho que o servidor faz.
Para transformar o ficheiro de input usando os filtros, decidimos recorrer à *system call* `fork()` e a pipes anónimos, exceto quando é solicitado o uso de um só filtro, aí basta fazer uma única vez `fork()` e redirecionar o input e o output para as ficheiros escolhidos pelo utilizador usando `dup2()`.
Caso o utilizador solicite o uso de mais que um filtro, sendo `n = nº de filtros`, criá-mos `n - 1` pipes anónimos (que nós erradamente, definimos com uma varíavel chamada `nr_forks`, em vez de `nr_pipes`).
Então, o que fazemos é criar `n` filhos sequenciais, em que o primeiro redireciona o seu `STDIN` para o ficheiro de input e o seu `STDOUT` para o descritor de escrita do primeiro pipe anónimo e chama a *system call* `execlp()` para usar o executável do filtro. O filho que vem a seguir (`i = 1`), redireciona o seu `STDIN` para o descritor de leitura do pipe anterior (`i - 1`) e o seu `STDOUT` passa a ser o descritor de escrita do segundo pipe anónimo e mais uma vez, chama `execlp()`. E assim, sucessivamente, até que chega ao último filtro em que, ao invés, de criar um novo pipe, redireciona o seu `STDOUT` para o ficheiro final.

Por exemplo (recorrendo a um diagrama, usado numa das aulas práticas, com algumas modificações):

[pedido]: transform input.mp3 output.mp3 eco rapido alto alto



5 SIGTERM

Tal como foi pedido no enunciado, caso o servidor recebe o sinal `SIGTERM`, verifica se existem processos a decorrer, então, o servidor, espera que todos os processos acabem naturalmente, e só depois disso é que termina de executar.

6 Conclusão

Para concluir, achámos que este trabalho foi bastante importante para testar as nossas capacidades e verificar quão bem absorvemos a matéria que foi lecionada durante o semestre, pois aplicámos o conhecimento que adquirimos em, literalmente, todos os guiões dados nas aulas práticas.