Sexta aula de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

O objectivo da aula foi o estudo do sistema de ficheiros, através do exemplo do sistema Unix.

2 Conceitos básicos

Um processo, a nível do SO, é o conceito que representa um programa em execução.

Um ficheiro, num SO como o Unix, é o conceito que representa não só um arquivo ou repositório lógico de informação, como também todos os dispositivos que possam ser usados pelos processos para comunicarem com o seu ambiente exterior. O ambiente exterior de um processo abrange todos os periféricos aos quais ele possa aceder, e também todos os processos com os quais esse processo possa comunicar (isto é, trocar informação, seja sob a forma de cópia explícita de dados de uma zona do mapa de memória um processo para uma zona do mapa de memória de outro processo, seja sob a forma de acesso a zonas comuns de memória).

Um ficheiro, por um lado tem um determinado suporte físico, e por outro lado, tem um nome simbólico, pelo qual pode ser designado pelos programas.

Compete ao SO garantir que os programadores não precisam de se 'preocupar' com a manutenção da informação no suporte físico do ficheiro (seja um disco, seja outro qualquer periférico). Isto significa que, uma vez dado o nome simbólico do ficheiro, e especificada a operação pretendida sobre o ficheiro, o SO se encarrega de localizar as estruturas de dados que, internamente ao SO, representam o ficheiro e dão acesso ao seu suporte físico. Uma vez localizadas essas estruturas, o SO desencadeia as operações mais elementares, algumas das quais envolvem acesso às portas hardware das interfaceds de

i/o dos periféricos, e responsabiliza-se por transferir os dados de e para os mapas de memória dos processos. O SO também se responsabiliza por todo o controlo das operações de i/o, incluindo a gestão dos tempos de espera pelo completar das operações (por exemplo quando um programa invoca uma operação READ e tem de se esperar pela chegada dos dados, o retorno ao ponto de chamada de READ só se faz depois de os dados terem chegado, sendo o SO que se encarrega de todo este controlo).

Cada SO tem uma determinada forma de apresentar o conceito de ficheiro, ao nível da sua interface de chamadas ao SO. No caso do Unix, todos os ficheiros são apresentados de uma mesma forma: como uma sequência de *bytes* (dita *stream*), como se ilustra na figura 1

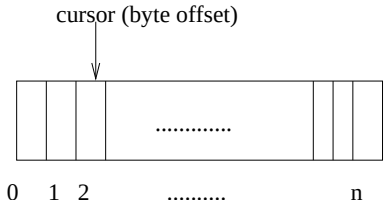

Figura 1: Nocão lógica de ficheiro Unix

3 Sistemas de ficheiros em geral

A parte do SO que se encarrega da gestão do acesso e da representação interna dos ficheiros, é habitualmente designada por Sistema de Ficheiros (SF). Note-se, contudo, que no Unix também é vulgar designar por sistema de ficheiros, um determinado subconjunto de ficheiros, existente no sistema, localizados por exemplo numa determinada unidade de disco.

Em geral, a parte do SO que trata dos ficheiros, tem as seguintes funções:

- manter estruturas de dados que representem a informação (os ficheiros) existentes num dado momento;
- decidir *onde* e *como* a informação é armazenada nos seus suportes físicos e *quem* tem acesso a ela;
- localizar e dar acesso à informação, quando os programas invocam as chamadas ao SO;

- gerir, isto é, reservar e libertar o espaço físico ocupado em disco, pelas representações dos ficheiros.

A figura 2 ilustra a relação entre o SF, as outras partes de um SO, o hardware do computador, e os processos utilizadores.

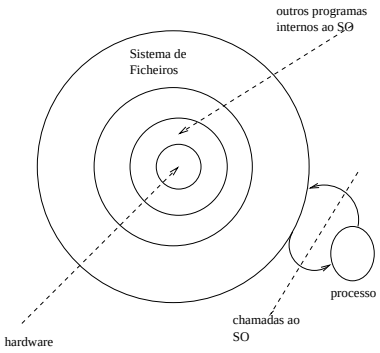

Figura 2: Sistema de ficheiros face às camadas de um SO

A figura, na sua multiplicidade de camadas concêntricas, ilustra o objectivo do SO, em ir 'escondendo' ao processo utilizador, os pormenores de realização das operações, desde o nível lógico (camadas mais exteriores da 'cebola') até ao nível físico, das operações de entrada e saída a nível hardware (camadas mais internas).

Esta ideia é reforçada pela figura 3, em que se ilustram os vários passos da execução de uma chamada ao SO, pedindo uma operação de leitura ou escrita de um ficheiro, atravessando as várias camadas do SO e do SF.

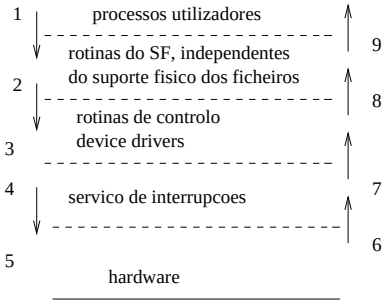

Figura 3: Hierarquia de níveis num sistema de ficheiros

1. Um programa invoca uma chamada ao SF;
2. as rotinas do nível superior do SF tratam de localizar o ficheiro, dado o seu nome simbólico (lógico), verificam direitos de acesso, e verificam se há dados disponíveis em *buffers* em memória; se a operação pode ser cumprida só com base nos dados em *buffers*, então, estas rotinas tratam logo de retornar ao programa utilizador (passo 9), senão descem ao nível inferior (passo 3);
3. as rotinas de controlo do periférico que suporta fisicamente o ficheiro (e.g. um disco) são chamadas e a operação física é desencadeada, acedendo às portas da interface hardware;
4. o mecanismo de interrupções é devidamente inicializado para se tratar a sequência de acções seguintes;
5. a operação está em curso;
6. o periférico, completada a operação, assinala-o, com um pedido de interrupção hardware, dirigido ao CPU;
7. a rotina de serviço correspondente, do *device driver* é invocada, para tratar o pedido de interrupção;
8. a rotina acede às portas da interface, obtém os dados, que são passados ao nível superior;
9. uma vez obtidos os dados no *buffer* respectivo, prepara-se o retorno ao programa;
10. retorno ao programa.

4 Sistema de Ficheiros: resumo dos objectivos

Um SF visa, idealmente, oferecer, aos utilizadores e aos programas em execução, uma abstracção de um espaço uniforme de ficheiros e esconder, o mais possível, os aspectos dependentes do dispositivos hardware.

Aspectos relevantes:

- Organização lógica de ficheiros: os ficheiros podem, conforme o SO, ser apresentados como sequências de bytes (e.g. como no Unix), ou surgirem com organizações mais elaboradas (e.g. estruturados em *records* lógicos);

- Sistema uniforme de entradas e saídas: o ideal seria todos os dispositivos de i/o serem tratados como se fossem ficheiros simbólicos, sobre os quais um mesmo conjunto de operações (chamadas ao SO) pudesse ser aplicado;

- Canais virtuais de i/o: o ideal seria dispor de um mecanismo que permitisse a um programa, mesmo durante a execução, decidir de e para que ficheiros pretende transferir dados, como se pudesse estabelecer canais de entrada e saída;

- Projecção completa de ficheiros em memória (*memory mapped files*): um ideal mais ambicioso seria encarar os ficheiros como uma 'extensão' do Espaço Virtual de Endereços de um processo, de tal modo que o SO, logo que o programa invocasse o primeiro acesso a um ficheiro, se encarregasse de o projectar completamente para uma região do mapa de memória virtual do processo e, a partir daí, o programa aceder-lhe-ia, da mesma forma que acede aos seus dados.

Exceptuando o último aspecto, iremos ver, de seguida, de que modo se podem realizr os outros objectivos de um SF, com ilustrações no caso do SO Unix.

5 Sistema de Ficheiros no Unix

No Unix e, em geral, num qualquer SO, há três níveis de interfaces de utilização:

- o nível do interpretador de comandos de linha, ao terminal, no Unix chamado *shell*;

- o nível das chamadas ao SO: executadas pelo núcleo do SO, com o processador em modo *supervisor* e invocadas como subrotinas, através de uma instrução máquina especial que origina uma interrupção quando executada (chamada uma interrupção originada por *software*, para a distinguir dos pedidos de interrupção originados pelo hardware das interfaces dos periféricos);

- um nível intermédio: corresponde às chamadas de funções das bibliotecas de suporte a linguagens de programação, e.g. C, que disponibilizam funções e tipos de dados possivelmente mais convenientes para o programador desenvolver aplicações; algumas destas funções são simplesmente de nível utilizador (e.g. funções sobre *strings* em C), mas outras funções

recorrem a chamadas ao SO, para efectuarem as operações pedidas, e.g. ler um caracter do teclado.

Neste capítulo, vamo-nos concentrar no nível das chamadas ao SO.

No Unix, um ficheiro é uma sequência de bytes, sem qualquer outra estrutura pré-definida, podendo estes bytes serem acedidos sequencialmente ou directamente, através de um cursor da posição corrente (*byte offset*). Qualquer outra estrutura ou interpretação do conteúdo do ficheiro compete aos programas de aplicação: e.g. ver o ficheiro organizado em parágrafos ou em linhas, ou interpretar caracteres aos quais se atribuem funções especiais, como a mudança de linha (*line feed*) ou o retorno(*CR- carriage return*); o mesmo se passa quanto à interpretação do conteúdo de ficheiros, para diferentes tipos, sejam executáveis, de imagem, etc. Isto compete aos programas de aplicação e não ao SO.

No Unix, os ficheiros são de três categorias principais:

- normais: correspondem aos ficheiros de dados em disco;
- directorias: contêm listas de nomes de ficheiros e seus identificadores internos (chamados *i-nodes*), que dão acesso à sua representação em tabelas internas ao SO, onde é guardada a informação sobre os seus atributos: qual o seu suporte físico, que permissões de acesso, se se trata de um ficheiro normal em disco ou de uma directoria, etc.
- especiais: correspondem a dispositivos periféricos, tal como o teclado, o écran, a impressora, o próprio disco, a memória (sim, a RAM), etc, os quais convém modelar como se fossem ficheiros lógicos, porque assim se torna muito mais fácil controlar as operações sobre eles.

Para se compreender o modo como um programa em execução (ou seja, um processo) acede aos ficheiros, temos de definir o conceito de *canal virtual* de entrada e saída.

6 Canais virtuais de entrada e saída

A figura 4 ilustra o conceito.

Na figura, um processo, com o seu mapa de memória encapsulado no rectângulo indicado, tem acesso a três canais pré-definidos, respectivamente designados pelo número 0 (canal standard de entrada), 1 (canal standard de saída) e 2 (canal standard de erro).

Estes canais identificam ficheiros aos quais o programa pode aceder. Habitualmente, no Unix, estes canais são automaticamente inicializados pelo

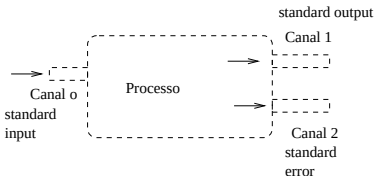

Figura 4: Canais de entrada e saída de um processo

SO (o modo exacto como isto é feito é estudado mais adiante), quando um processo é criado, dando tipicamente acesso ao teclado do computador (canal 0) ou ao écran (canais 1 e 2). O canal 0 pode ser nomeado em operações de leitura (*read*) e o canal 1 pode ser nomeado em operações de escrita (*write*). O que é interessante é que o programa não precisa, nessas operações, de indicar o nome do ficheiro que correponde ao teclado ou ao écran, apenas tendo de indicar os números de canais.

Isto torna-se mais interessante quando se permite que um processo 'veja', durante a sua própria execução, os seus canais 0, 1 ou 2, redirigidos para outros ficheiros, diferentes do teclado e do écran. Ao fazer isto, o programa pode continuar a tentar ler do canal 0 mas, agora, irá ler do ficheiro ao qual o canal foi ligado, em vez de ler do teclado, mas o programa não precisou de ser modificado, isto é continua a ler do canal 0.

A flexibilidade que isto traz é tão grande, que existem, no Unix e noutros SO, operadores especiais de redirecção de canais de i/o, que podem ser invocados a nível da linha de comandos do *shell*, como se ilustra na figura 5, na qual um processo, executando o comando p, tem o canal 0 ligado ao ficheiro 'f1' e o canal 1 ligado ao ficheiro 'f2'.

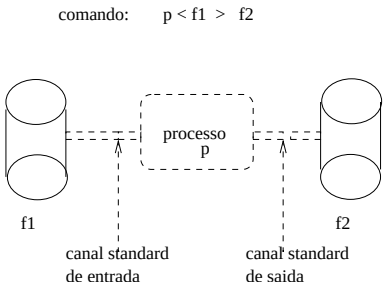

Figura 5: Redirecção de canais de um processo

A ligação dos canais, correspondente à configuração indicada na figura,

é efectuada sob controlo do programa do *shell*, através de chamadas ao SO.

Esquemas de comunicação mais gerais podem estabelecer-se, por exemplo, interligando múltiplos processos, cada um executando o seu comando, mas passando os dados e resultados uns aos outros, como se fosse um *pipeline* (figura 6)

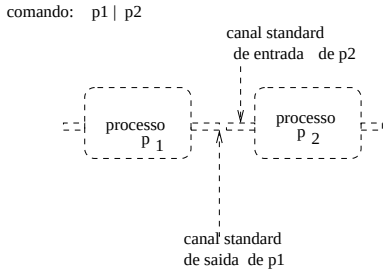

Figura 6: *Pipeline de processos*

Este esquema é possível se se dispuser de um dispositivo de comunicação (um *pipe*, como veremos mais adiante), que permita ligar o canal standard de saída do processo p1 ao canal standard de entrada do processo p2.

Resumo: no Unix, o acesso a um ficheiro faz-se estabelecendo canais de ligação entre um processo e o ficheiro. Estes canais são identificados por números inteiros não negativos e podem ser redirigidos de modo a dar acesso a outros ficheiros, mesmo durante a execução de um programa.

7 Chamadas ao SF Unix

Na figura 7 ilustra o ciclo típico de utilização dos ficheiros, num SO como o Unix.

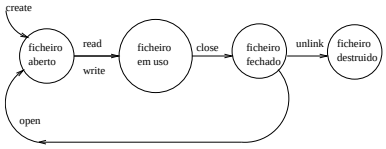

Figura 7: Ciclo de utilização de ficheiros

Neste ciclo indicam-se as operações permitidas sobre um ficheiro normal no Unix:

- create: para criar o ficheiro, atribuindo-lhe um nome simbólico;
- open: para abrir um novo canal para um dado ficheiro;
- read/write: para ler/escrever, através de um canal aberto;
- close: para fechar a ligação de um canal a um ficheiro;
- unlink: para destruir o ficheiro, removendo o seu nome simbólico.

Este ciclo pode também situar-se relativamente ao modo como cada processo acede aos ficheiros, através dos canais (figura 8).

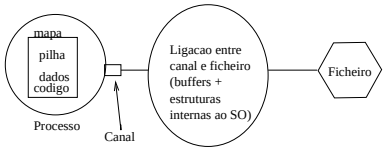

Figura 8: Ligação Processo-Canal-Ficheiro

As operações que abrem um canal para um ficheiro (open) estabelecem uma ligação entre as estruturas internas do SO que descrevem o acesso ao ficheiro e as estruturas intermédias, e.g. os *buffers* que são necessárias para se controlar a transferência de dados entre o processo e o ficheiro. São essas ligações e estruturas intermédias que concretizam o conceito de canal virtual de entrada e saída.

Do ponto de vista do processo, estes canais são os únicos pontos de acesso ao ficheiro, ou seja, uma vez fechados, o ficheiro deixa de ser acessível. Os números dos canais abertos são designados, no Unix, como *descritores de ficheiros*, na medida em que são eles que dão acesso às estruturas internas do SO que descrevem os ficheiros. Existe um número pré-definido (isto é um parâmetro do SO) de canais que podem ser abertos por cada processo. Durante a execução de um programa, os canais podem ser abertos e fechados, e redirigidos para diferentes ficheiros, conforme as necessidades.

Esta associação processo-canal-ficheiro pode ser feita e desfeita dinamicamente (isto é, durante a execução), através de operações open/close. Isto permite ao SO optimizar posteriores acesso ao ficheiro aberto, pois que certas estruturas que descrevem o ficheiro e que, no início estão em disco, podem ser copiadas para memória central, tornando o acesso mais rápido. O mesmo sucede com os buffers em memória, que vão guardando os dados de trabalho do ficheiro, conforme estes vão sendo transferidos entre o processo e o ficheiro.

Nas operações de abertura do ficheiro, o SO também pode controlar os direitos de acesso do processo utilizador, verificando se este tem as permissões necessárias para aceder ao ficheiro, no modo pretendido. Uma vez feita esta verificação, posteriores operações de read ou write são mais facilmente controladas.