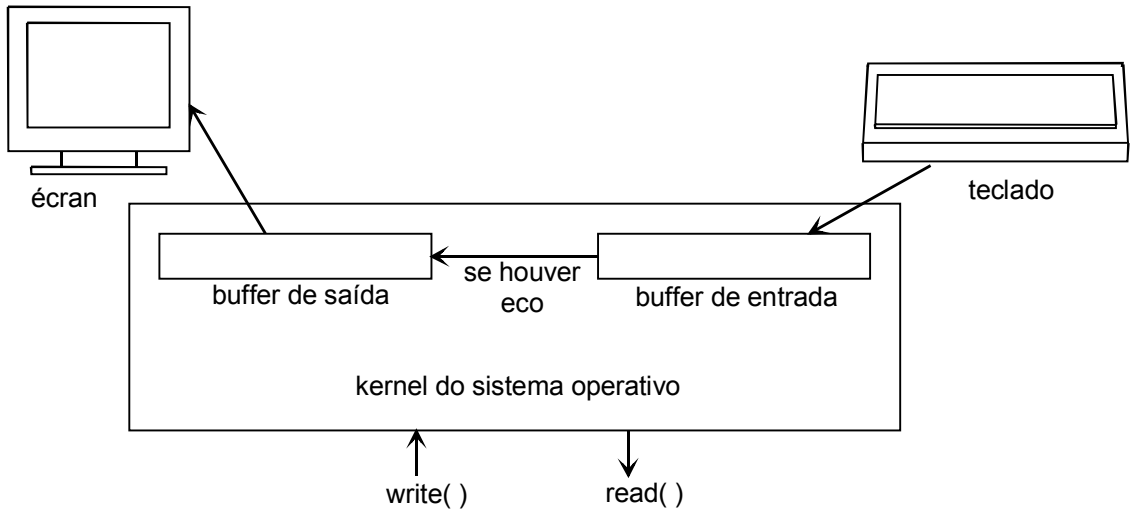
2. Consola, ficheiros e directórios

2.1. Entrada/Saída na consola

A consola (teclado + écran) é vista geralmente nos S.O.'s como um ou mais ficheiros onde se pode ler ou escrever texto. Esses ficheiros são normalmente abertos pela rotina de *C startup*. A biblioteca standard do C inclui inúmeras funções de leitura e escrita directa nesses ficheiros (`printf()`, `scanf()`, `getchar()`, `putchar()`, `gets()`, `puts()`, ...).
No entanto podemos aceder também a esses periféricos através dos serviços dos S.O.'s.



Implementação típica da consola num sistema operativo

Em Unix os ficheiros são acedidos através de descritores, que são números inteiros, geralmente pequenos. A consola tem 3 componentes, conhecidos por *standard input* (teclado), *standard output* (écran) e *standard error* (também écran) que são representados por ficheiros com os descritores 0, 1 e 2, a que correspondem as constantes `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`, definidas em `<unistd.h>`. No início da execução de todos os programas estes componentes encontram-se já abertos e prontos a utilizar.

Na API (*Application Programming Interface*) do Unix não existem funções específicas de leitura e escrita nos componentes da consola, sendo necessário usar as funções genéricas de leitura e escrita em ficheiros (descritas mais à frente), ou as já referidas funções da biblioteca standard do C.

No entanto os componentes da consola podem ser redireccionados para ficheiros em disco utilizando os símbolos da *shell* (`>`, `<`, `>>` e `<<`) na linha de comandos desta (por exemplo: **`ls -la > dir.txt`**), ou então programaticamente, utilizando o seguinte serviço:

```
#include <unistd.h>

int dup2(int filedes, int filedes2);
```

Identifica o ficheiro **`filedes`** com o descritor **`filedes2`**, ou seja copia a referência ao ficheiro identificado por **`filedes`** para a posição **`filedes2`**, de modo que quando se usa **`filedes2`** passa-se a referenciar o ficheiro associado a **`filedes`**. Se **`filedes2`** estiver aberto, é fechado antes da cópia. Retorna **`filedes2`** ou -1 se ocorrer um erro.

Assim, por exemplo, para redireccionar a saída standard (écran) para um outro ficheiro com descritor `fd` basta chamar:

```
dup2(fd, STDOUT_FILENO);
```

A partir deste momento qualquer escrita em `STDOUT_FILENO`, neste programa, passa a ser feita no ficheiro representado pelo descritor `fd`.

2.2. Características de funcionamento da consola

As consolas em Unix podem funcionar em dois modos distintos: canónico e primário (*raw*). No modo canónico existe uma série de caracteres especiais de entrada que são processados pela consola e não são transmitidos ao programa que a está a ler. São exemplos desses caracteres: <control-U> para apagar a linha actual, <control-H> para apagar o último carácter escrito, <control-S> para suspender a saída no écran e <control-Q> para a retomar. Muitos destes caracteres especiais podem ser alterados programaticamente. Além disso, em modo canónico, a entrada só é passada ao programa linha a linha (quando se tecla <return>) e não carácter a carácter. No modo primário não há qualquer processamento prévio dos caracteres teclados podendo estes, quaisquer que sejam, ser passados um a um ao programa que está a ler a consola.

Entre o funcionamento dos dois modos é possível programar, através de numerosas opções, algumas não padronizadas, muitos comportamentos diferentes em que alguns caracteres de entrada são processados pela própria consola e outros não.

As características das consolas em Unix podem ser lidas e alteradas programaticamente utilizando os serviços `tcgetattr()` e `tcsetattr()`:

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);
int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

O serviço `tcgetattr()` preenche uma estrutura `termios`, cujo endereço é passado em **`termptr`**, com as características do componente da consola cujo descritor é **`filedes`**. O serviço `tcsetattr()` modifica as características da consola **`filedes`** com os valores previamente colocados numa estrutura `termios`, cujo endereço é passado em **`termptr`**. O parâmetro **`opt`** indica quando a modificação irá ocorrer e pode ser uma das seguintes constantes definidas em `termios.h`:

- `TCSANOW` - A modificação é feita imediatamente
- `TCSADRAIN` - A modificação é feita depois de se esgotar o buffer de saída
- `TCSAFLUSH` - A modificação é feita depois de se esgotar o buffer de saída; quando isso acontece o buffer de entrada é esvaziado (*flushed*).

Retornam -1 no caso de erro.

A estrutura `termios` inclui a especificação de numerosas *flags* e opções. A sua definição (que se encontra em `termios.h`) é a seguinte:

```
struct termios {
    tcflag_t  c_iflag;          /* input flags */
    tcflag_t  c_oflag;          /* output flags */
    tcflag_t  c_cflag;          /* control flags */
    tcflag_t  c_lflag;          /* local flags */
```

```
    cc_t      c_cc[NCCS];          /* control characters */
}
```

Os quatros primeiros campos da estrutura `termios` são compostos por *flags* de 1 ou mais bits, em que cada uma delas é representada por um nome simbólico definido em `termios.h`. Quando pretendemos activar mais do que uma *flag* de um determinado campo simplesmente devemos efectuar o *or* bit a bit (`|`) entre os seus nomes simbólicos e atribuir o resultado ao campo a que pertencem, como se pode ver no seguinte exemplo:

```
struct termios tms;

tms.c_iflag = (IGNBRK | IGNCR | IGNPAR);
```

No entanto, e mais frequentemente, pretendemos apenas activar ou desactivar uma determinada *flag* de um dos campos de `termios` sem alterar as outras. Isso pode fazer-se com a operação *or* ou com a operação *and* (`&`) e a negação (`~`). Exemplos:

```
tcgetattr(STDIN_FILENO, &tms);  /* lê características actuais da consola */
tms.c_oflag |= (IXON | IXOFF);       /* activa Xon/Xoff */
tms.c_cflag &= ~(PARENB | CSTOPB);   /* desactiva paridade e 2 stop bits */
tcsetattr(STDIN_FILENO, TCSANOW, &tms); /* escreve novas características */
```

O último campo de `termios` é um array (`c_cc`) onde se definem os caracteres especiais que são pré-processados pela consola se esta estiver a funcionar no modo canónico. O tipo `cc_t` é habitualmente igual a `unsigned char`. Cada uma das posições do array é acedida por uma constante simbólica, também definida em `termios.h`, e corresponde a um determinado carácter especial. Por exemplo para definir o carácter que suspende o processo de *foreground* poderá usar-se:

```
tms.c_cc[VSUSP] = 26;     /* 26 - código ASCII de <control-Z> */
```

Na tabela seguinte podem ver-se os caracteres especiais e respectivas constantes simbólicas, definidos na norma POSIX.1.

Caracteres especiais	Constante de `c_cc[]`	Flag activadora		Valor típico
		Campo	Flag	
Fim de ficheiro	`VEOF`	`c_lflag`	`ICANON`	ctrl-D
Fim de linha	`VEOL`	`c_lflag`	`ICANON`	
Backspace	`VERASE`	`c_lflag`	`ICANON`	ctrl-H
Envia sinal de interrupção (SIGINT)	`VINTR`	`c_lflag`	`ISIG`	ctrl-C
Apaga a linha actual	`VKILL`	`c_lflag`	`ICANON`	ctrl-U
Envia sinal de quit (SIGQUIT)	`VQUIT`	`c_lflag`	`ISIG`	ctrl-\
Retoma a escrita no écran	`VSTART`	`c_iflag`	`IXON/IXOFF`	ctrl-Q
Para a escrita no écran	`VSTOP`	`c_iflag`	`IXON/IXOFF`	ctrl-S
Envia o sinal de suspenção (SIGTSTP)	`VSUSP`	`c_lflag`	`ISIG`	ctrl-Z

Listam-se agora as principais *flags* definidas na norma POSIX.1:

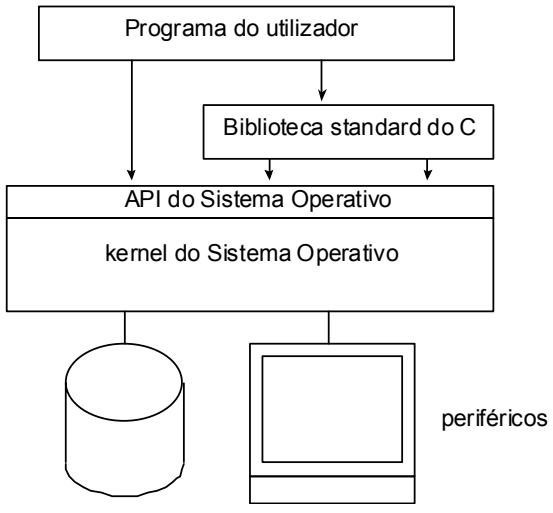
Campo	**Constante**	**Descrição**
`c_iflag`	BRKINT	Gera o sinal SIGINT na situação de BREAK
	ICRNL	Mapeia CR (return) em NL (newline) no buffer de entrada
	IGNBRK	Ignora situação de BREAK
	IGNCR	Ignora CR
	IGNPAR	Ignora caracteres com erro de paridade (terminais série)

	INLCR	Mapeia NL em CR no buffer de entrada
	INPCK	Verifica a paridade dos caracteres de entrada
	ISTRIP	Retira o 8º bit dos caracteres de entrada
	IXOFF	Permite parar/retomar o fluxo de entrada
	IXON	Permite parar/retomar o fluxo de saída
	PARMRK	Marca os caracteres com erro de paridade
`c_oflag`	OPOST	Executa pré-processamento de saída especificado noutras flags deste campo mas que são dependentes da implementação
`c_cflag`	CLOCAL	Ignora as linhas série de estado (RS-232)
	CREAD	Permite a leitura de caracteres (linha RD RS-232)
	CS5/6/7/8	5,6,7 ou 8 bits por carácter
	CSTOPB	2 stop bits (1 se desactivada)
	HUPCL	Desliga ligação quando do fecho do descriptor (RS-232, modem)
	PARENB	Permite o bit de paridade
	PARODD	Paridade ímpar (par se desactivada)
`c_lflag`	ECHO	Executa o eco dos caracteres recebidos para o buffer de saída
	ECHOE	Apaga os caracteres visualmente (carácter de backspace - VERASE)
	ECHOK	Apaga as linhas visualmente (carácter VKILL)
	ECHONL	Ecoa o carácter de NL mesmo se ECHO estiver desactivado
	ICANON	Modo canónico (processa caracteres especiais)
	IEXTEN	Processa caracteres especiais extra (dependentes da implementação)
	ISIG	Permite o envio de sinais a partir da entrada
	NOFLSH	Desactiva a operação de flush depois de interrupção ou quit
	TOSTOP	Envia o sinal SIGTTOU a um processo em background que tente escrever na consola.

Para mais pormenores sobre outras *flags* e características dependentes da implementação consultar o manual através do comando `man` do Unix.

2.3. Serviços de acesso a ficheiros

A biblioteca standard da linguagem C (*ANSI C library*) contém um conjunto rico de funções de acesso, criação, manipulação, leitura e escrita de ficheiros em disco e dos 3 componentes da consola. A utilização dessas funções torna os programas independentes do Sistema Operativo, uma vez que a biblioteca standard é a mesma para todos os S.O.'s. No entanto introduz uma camada extra de código entre o programa e o acesso aos objectos do S.O. (ficheiros, consola, etc) como se ilustra na figura seguinte.



A posição da biblioteca standard do C

Assim, para se conseguir um pouco mais de performance, é possível chamar directamente os serviços dos S.O.'s, pagando-se o preço dos programas assim desenvolvidos não poderem ser compilados, sem alterações, noutros Sistemas Operativos.

Apresentam-se de seguida alguns dos serviços referentes à criação, leitura e escrita de ficheiros no Sistema Operativo Unix, conforme especificados pela norma POSIX.1.

Utiliza-se o serviço `open()` para a abertura ou criação de novos ficheiros no disco. Este serviço retorna um descritor de ficheiro, que é um número inteiro pequeno. Esses descritores são depois usados como parâmetros nos outros serviços de acesso aos ficheiros. Os ficheiros em Unix não são estruturados, sendo, para o Sistema Operativo, considerados apenas como meras sequências de *bytes*. São os programas que têm de interpretar correctamente o seu conteúdo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char * pathname, int oflag, …);  /* mode_t mode */
```

`pathname` – nome do ficheiro;
`oflag` – combinação de várias *flags* de abertura:
uma e só uma das três constantes seguintes deve estar presente:

- `O_RDONLY` – abertura para leitura
- `O_WRONLY` – abertura para escrita
- `O_RDWR` – abertura para leitura e escrita

Outras *flags*:

- `O_APPEND` – acrescenta ao fim do ficheiro
- `O_CREAT` – Cria o ficheiro se não existe; requer o parâmetro **`mode`**
- `O_EXCL` – Origina um erro se o ficheiro existir e se `O_CREAT` estiver presente
- `O_TRUNC` – Coloca o comprimento do ficheiro em 0, mesmo se já existir
- `O_SYNC` – as operações de escrita só retornam depois dos dados terem sido fisicamente transferidos para o disco

`mode` – permissões associadas ao ficheiro; parâmetro opcional apenas obrigatório quando da criação de novos ficheiros; pode ser um *or* bit a bit (|) entre as seguintes constantes:
`S_IRUSR` - user read
`S_IWUSR` - user write
`S_IXUSR` - user execute
`S_IRGRP` - group read
`S_IWGRP` - group write
`S_IXGRP` - group execute
`S_IROTH` - others read
`S_IWOTH` - others write
`S_IXOTH` - others execute
A função retorna um descritor de ficheiros ou –1 no caso de erro.

Como se pode ver na descrição acima, quando se cria um novo ficheiro é necessário especificar as suas permissões no parâmetro **`mode`** de `open()`. Além disso, para se criar um novo ficheiro, é necessário possuir as permissões de *write* e *execute* no directório onde se pretende criá-lo. Se o ficheiro for efectivamente criado, as permissões com que fica podem não ser exactamente as que foram especificadas. Antes da criação, as permissões

especificadas são sujeitas à operação de *and* lógico com a negação da chamada "máscara do utilizador" (*user mask* ou simplemente *umask*). Essa máscara contém os bits de permissão que não poderão ser nunca colocados nos novos ficheiros ou directórios criados pelo utilizador. Essa máscara é geralmente definida no *script* de inicialização da *shell* de cada utilizador e um valor habitual para ela é o valor 0002 (octal), que proíbe a criação de ficheiros e directórios com a permissão de *write* para outros utilizadores (*others write*).

A máscara do utilizador pode ser modificada na *shell* com o comando `umask` ou programaticamente com o serviço do mesmo nome.

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Modifica a máscara de criação de ficheiros e directórios para o valor especificado em **`cmask`**; este valor pode ser construído pelo *or* lógico das constantes cujos nomes aparecem na descrição do parâmetro `mode` do serviço `open()`.
Retorna o valor anterior da máscara.

A leitura ou escrita em ficheiros faz-se directamente com os serviços `read()` e `write()`. Os *bytes* a ler ou escrever são tranferidos para, ou de, *buffers* do próprio programa, cujos endereços se passam a estes serviços. O kernel do S.O. pode também manter os seus próprios *buffers* para a transferência de informação para o disco. Os componentes da consola também podem ser lidos ou escritos através destes serviços.

A leitura ou escrita de dados faz-se sempre da ou para a posição actual do ficheiro. Após cada uma destas operações essa posição é automaticamente actualizada de modo a que a próxima leitura ou escrita se faça exactamente para a posição a seguir ao último byte lido ou escrito. Quando o ficheiro é aberto (com `open()`) esta posição é o início (posição 0) do ficheiro, excepto se a *flag* `O_APPEND` tiver sido especificada.

```
#include <unistd.h>

ssize_t read(int filedes, void *buffer, size_t nbytes);
ssize_t write(int filedes, const void *buffer, size_t nbytes);
```

`nbytes` indica o número de *bytes* a tranferir enquanto que **`buffer`** é o endereço do local que vai receber ou que já contém esses *bytes*.
As funções retornam o número de *bytes* efectivamente transferidos, que pode ser menor que o especificado. O valor –1 indica um erro, enquanto que o valor 0, numa leitura, indica que se atingiu o fim do ficheiro.

Quando se pretende ler da consola (`STDIN_FILENO`) e se utiliza o serviço `read()`, geralmente este só retorna quando se tiver entrado uma linha completa (a menos que a consola se encontre no modo primário). Se o tamanho do *buffer* de leitura for maior do que o número de *bytes* entrados na linha, o serviço `read()` retornará com apenas esses *bytes*, indicando no retorno o número lido. No caso contrário, ficarão alguns *bytes* pendentes no *buffer* interno da consola à espera de um próximo `read()`.

A posição do ficheiro de onde se vai fazer a próxima transferência de dados (leitura ou escrita) pode ser ajustada através do serviço `lseek()`. Assim é possível ler ou escrever partes descontínuas de ficheiros, ou iniciar a leitura ou escrita em qualquer posição do

ficheiro.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Desloca o apontador do ficheiro de **offset** bytes (pode ser negativo ou positivo) com uma origem especificada em **whence**:
SEEK_SET - início do ficheiro
SEEK_CUR - posição corrente do ficheiro
SEEK_END - final do ficheiro
Retorna -1 no caso de erro ou o valor da nova posição do ficheiro.

Se se quiser conhecer a posição actual do ficheiro, isso pode ser feito com a chamada:

```
pos = lseek(fd, 0, SEEK_CUR);
```

Quando não houver mais necessidade de manipular um ficheiro que se abriu anteriormente, este deve ser fechado quanto antes. Os ficheiros abertos fecham-se com o serviço `close()`.

```
#include <unistd.h>

int close(int filedes);
```

Retorna 0 no caso de sucesso e –1 no caso de erro.

Finalmente se quisermos apagar um ficheiro para o qual tenhamos essa permissão podemos utilizar o serviço `unlink()`.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Apaga a referência no directório correspondente ao ficheiro indicado em **pathname** e decrementa a contagem de *links*.
Retorna 0 no caso de sucesso e –1 no caso de erro.

Para se poder apagar um ficheiro é necessário ter as permissões de escrita e execução no directório em que se encontra o ficheiro. Embora a referência ao ficheiro no directório especificado desapareça, o ficheiro não será fisicamente apagado se existirem outros *links* para ele. Os dados do ficheiro poderão ainda ser acedidos através desses *links*. Só quando a contagem de *links* se tornar 0 é que o ficheiro será efectivamente apagado.

Se se apagar (com `unlink()`) um ficheiro aberto, este só desaparecerá do disco quando for fechado.

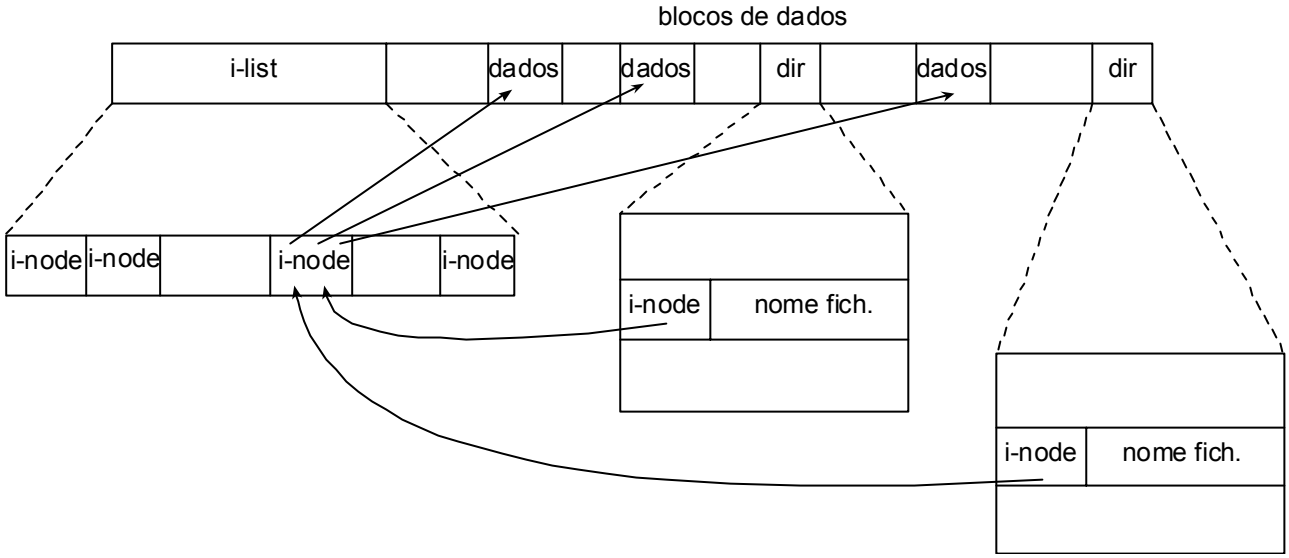
2.4. O sistema de ficheiros em Unix

Basicamente o sistema de ficheiros do Unix é constituído por duas partes distintas, armazenadas numa partição de um disco: uma lista de *i-nodes* e uma sequência de blocos de dados (ficheiros e directórios).

A lista de *i-nodes* (*i-list*) contém uma série de pequenas estruturas designadas por *i-nodes* que contêm toda a informação associada a cada ficheiro ou directório, nomeadamente a data de criação e modificação, tamanho, dono, grupo, permissões, número de links e principalmente quais os blocos, e por que ordem, pertencem ao ficheiro ou directório associado. Se o ficheiro for muito extenso poderá haver necessidade de associar mais informação (blocos de índices) a esse ficheiro (para listar todos os blocos que o compõem).

Os directórios são apenas sequências de entradas em que cada uma tem apenas o nome do ficheiro ou subdirectório nele contido e o número do *i-node* onde reside o resto da informação.

É possível que um mesmo ficheiro físico (ou directório) esteja presente em vários directórios, constituindo-se assim um conjunto de (*hard*) links para esse ficheiro. Há também ficheiros especiais (*symbolic links*) cujo conteúdo referencia outro ficheiro.



O sistema de ficheiros (com 2 *links* para o mesmo ficheiro físico)

2.5. Acesso ao sistema de ficheiros

Já vimos que o sistema de ficheiros pode armazenar vários tipos de informação nos seus blocos de dados. Na breve exposição anterior podemos para já identificar 3 tipos: ficheiros normais, directórios e *links* simbólicos. No entanto, como veremos mais tarde existem outros, como: os FIFOs, os ficheiros que representam directamente periféricos de entrada/saída (*character special file* e *block special file*) e por vezes *sockets*, semáforos, zonas de memória partilhada e filas de mensagens. Todos estes tipos aparecem pelo menos listados num directório e podem ser nomeados através de um *pathname*.
É possível extrair muita informação, programaticamente, acerca de um objecto que apareça listado num directório, ou que já tenha sido aberto e seja representado por um descritor, através dos seguintes três serviços.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

Todos estes serviços preenchem uma estrutura `stat`, cujo endereço é passado em **`buf`**, com informação acerca de um nome que apareça num directório (especificado em **`pathname`**), ou através do seu descritor (**`filedes`**) se já estiver aberto.
O serviço `lstat()` difere de `stat()` apenas no facto de retornar informação acerca de um *link* simbólico, em vez do ficheiro que ele referencia (que é o caso de `stat()`).
Retornam 0 no caso de sucesso e –1 no caso de erro.

A estrutura `stat` contém numerosos campos e é algo dependente da implementação. Consultar o `man` ou a sua definição em `<sys/stat.h>`.
Alguns campos mais comuns são:

```
struct stat {
  mode_t st_mode;    /* tipo de ficheiro e permissões */
  ino_t st_ino;      /* número do i-node correspondente */
  nlink_t st_nlink;  /* contagem de links */
  uid_t st_uid;      /* identificador do dono */
  gid_t st_gid;      /* identificador do grupo do dono */
  off_t st_size;     /* tamanho em bytes (ficheiros normais) */
  time_t st_atime;   /* último acesso */
  time_t st_mtime;   /* última modificação */
  time_t st_ctime;   /* último mudança de estado */
  long st_blocks;    /* número de blocos alocados */
}
```

O tipo de ficheiro está codificado em alguns dos bits do campo `st_mode`, que também contém os bits de permissão nas 9 posições menos significativas. Felizmente em `<sys/stat.h>` definem-se uma série de macros para fazer o teste do tipo de ficheiro. A essas macros é necessário passar o campo `st_mode` devidamente preenchido, sendo avaliadas como `TRUE` ou `FALSE`.

As macros que geralmente estão aí definidas são as seguintes:

- `S_ISREG()` Testa se se trata de ficheiros regulares
- `S_ISDIR()` Testa se se trata de directórios
- `S_ISCHR()` Testa se se trata de dispositivos orientados ao carácter
- `S_ISBLK()` Testa se se trata de dispositivos orientados ao bloco
- `S_ISFIFO()` Testa se se trata de fifo's
- `S_ISLNK()` Testa se se trata de links simbólicos
- `S_ISSOCK()` Testa se se trata de sockets

No exemplo seguinte podemos ver um pequeno programa que indica o tipo de um ou mais nomes de ficheiros passados na linha de comando.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int k;
  struct stat buf;
  char *str;

  for (k=1; k<argc; k++) {
```

```
    printf("%s: is ", argv[k]);
    lstat(argv[k], &buf);          /* preenche buf (struct stat) */

    if      (S_ISREG(buf.st_mode))  str = "regular";
    else if (S_ISDIR(buf.st_mode))  str = "directory";
    else if (S_ISCHR(buf.st_mode))  str = "character special";
    else if (S_ISBLK(buf.st_mode))  str = "block special";
    else if (S_ISFIFO(buf.st_mode)) str = "fifo";
    else if (S_ISLNK(buf.st_mode))  str = "symbolic link";
    else if (S_ISSOCK(buf.st_mode)) str = "socket";
    else str = "unknown";

    printf("%s\n", str);
  }
  return 0;
}
```

Como se viu, os directórios do sistema de ficheiros do Unix não passam de ficheiros especiais. Existe uma série de serviços para criar, remover e ler directórios, assim como tornar um determinado directório o directório corrente. O directório corrente (também designado por *cwd-current working directory* ou directório por defeito, ou ainda directório de trabalho) é aquele onde são criados os novos ficheiros, ou abertos ficheiros existentes, se apenas for indicado o nome do ficheiro e não um *pathname*.

Nunca é possível escrever directamente num directório (no ficheiro especial que representa o directório). Só o sistema operativo o pode fazer quando se solicita a criação ou a remoção de ficheiros nesse directório.

A criação de directórios e a sua remoção pode fazer-se com os serviços `mkdir()` e `rmdir()`. Os directórios deverão ser criados com pelo menos uma das permissões de *execute* para permitir o acesso aos ficheiros aí armazenados (não esquecer a modificação feita pela máscara do utilizador). Para remover um directório este deverá estar completamente vazio.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Cria um novo directório especificado em **`pathname`** e com as permissões especificadas em **`mode`** (alteradas pela máscara do utilizador).

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Remove o directório especificado em **`pathname`** se estiver vazio.
Ambos os serviços retornam 0 no caso de sucesso e –1 no caso de erro.

O acesso às entradas contidas num directórios faz-se através de um conjunto de 4 serviços, nomeadamente `opendir()`, `readdir()`, `rewinddir()` e `closedir()`. Para aceder ao conteúdo de um directório este tem de ser aberto com `opendir()`, retornando um apontador (`DIR *`) que é depois usado nos outros serviços. Sucessivas chamadas a `readdir()` vão retornando por ordem as várias entradas do directório, podendo-se voltar sempre à primeira entrada com `rewinddir()`. Quando já não for mais necessário deverá

fechar-se o directório com `closedir()`.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Abre o directório especificado em **pathname**. Retorna um apontador (`DIR *`) que representa o directório ou `NULL` no caso de erro.

```
struct dirent *readdir(DIR *dp);
```

Lê a próxima entrada do directório (começa na primeira) retornando um apontador para uma estrutura `dirent`. No caso de erro ou fim do directório retorna `NULL`.

```
void rewinddir(DIR *dp);
```

Faz com que a próxima leitura seja a da primeira entrada do directório.

```
int closedir(DIR *dp);
```

Fecha o directório. Retorna 0 no caso de sucesso e –1 no caso de erro.

A estrutura `dirent` descreve uma entrada de um directório contendo pelo menos campos com o nome do ficheiro e do *i-node* associado ao ficheiro. Esses dois campos obrigatórios são os que aparecem na definição seguinte:

```
struct dirent {
  ino_t d_ino;               /* número do i-node que descreve o ficheiro */
  char d_name[NAME_MAX+1];   /* string com o nome do ficheiro */
}
```

O directório corrente de um programa pode ser obtido com o serviço `getcwd()`.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Obtém o nome (*pathname*) do directório por defeito corrente. A esta função deve ser passado o endereço de um buffer (**buf**) que será preenchido com esse nome, e também o seu tamanho em *bytes* (**size**). O serviço retorna `buf` se este for preenchido, ou `NULL` se o tamanho for insuficiente.

Este directório corrente pode ser mudado programaticamente com o serviço `chdir()`.

```
#include <unistd.h>

int chdir(const char *pathname);
```

Muda o directório por defeito corrente para **pathname**.
Retorna 0 no caso de sucesso e -1 no caso de erro.