10
Ficheiros e Directorias

Sumário:

- **Introdução**
- **Nome de ficheiro**
- **Nome de caminho**
- **Descritor de ficheiro**
- **Estado dum ficheiro**
- **Tipos de ficheiros**
- **Permissões de acesso a ficheiros**
- **Set-User-ID e Set-Group-ID**
- **Função umask**
- **Funções de modificação de permissões de acesso: chmod e fchmod**
- **Funções de modificação de proprietário e de grupo: chown, fchown e lchown**
- **Elos para ficheiros**
- **Funções de manipulação de elos: link, unlink, remove e rename**
- **Funções de manipulação de elos simbólicos: symlink e readlink**
- **Função de temporização de ficheiros: utime**
- **Funções de manipulação de directorias: mkdir e rmdir**
- **Funções de leitura de directorias**
- **Funções chdir, fchdir e getcwd**

Introdução

Há vários utilitários do Unix para **aceder** e **manipular** ficheiros e directorias, tais como, por exemplo, ls, mkdir, cp, rm, etc. Nesta subsecção vamos ver como fazer o mesmo a partir dum programa escrito em linguagem C. Esta programação requer a inclusão dos ficheiros de protótipos <**unistd.h**>, <**sys/types.h**> e <**sys/stat.h**>.

Nome de ficheiro

Qualquer ficheiro, directoria ou ficheiro especial tem um nome (*filename*).

Contudo, há dois caracteres ASCII que não são permitidos no nome dum ficheiro: o carácter ´\0´ (ou NULL) e o carácter ´/´. O carácter ´/´ é o separador de nomes de ficheiros no nome dum caminho (ou *pathname*) do sistema de ficheiros, ao passo que o carácter ´\0´ é o carácter terminador duma string, e, por consequência, do nome dum caminho.

Nome de caminho

Um caminho é uma string construída por um ou mais nomes de ficheiros separados pelo carácter ´/´. Um caminho é absoluto se começa com o carácter ´/´; caso contrário, o caminho é relativo.

Há uma diferença entre o nome dum ficheiro e o nome dum caminho. O nome dum ficheiro é um componente dum nome dum caminho. O nome dum caminho é uma string que é passada dum processo para o kernel quando um ficheiro específico é referenciado. No entanto, muitos utilizadores e autores de livros referem-se a strings como, por exemplo, /usr/lib/crontab como se fosse um nome dum ficheiro, quando na realidade é o nome dum caminho.

Descritor de ficheiro

Um descritor de ficheiro é um número inteiro que é usado para identificar um ficheiro aberto para I/O. Os actuais sistemas operativos já permitem que haja mais de 20 ficheiros abertos por processo. Esta informação fica guardada no PCB (process control block) de cada processo. Normalmente, os descritores de ficheiros 0, 1 e 2 estão associados à entrada estandardizada, à saída estandardizada e ao erro estandardizado, respectivamente, dum processo.

O kernel atribui descritores de ficheiros aquando das seguintes chamadas ao sistema bem-sucedidas: **open**, **creat**, **dup**, **pipe**, e **fcntl**.

Note-se que os sockets também tem descritores inteiros. O kernel atribui descritores a sockets quando as seguintes chamadas ao sistema são bem-sucedidas: **accept**, **pipe**, **socket**, e **socketpair**.

Estado dum ficheiro

Um ficheiro tem ATRIBUTOS!
Estes atributos constituem o ESTADO do dito ficheiro.

Há duas funções que permitem saber o estado dum ficheiro.

- Obtém informação sobre o ficheiro identificado por `path`.
  `int stat(char *path, struct stat *buf)`

- Obtém informação sobre o ficheiro identificado pelo descritor `fd`.
  `int fstat(int fd, struct stat *buf)`

O argumento `buf` é um ponteiro para uma estrutura `stat` na qual está colocada a informação respeitante ao ficheiro. Ambas as funções devolvem ou o valor 0 ou o valor –1, consoante há sucesso ou erro na chamada da função. A estrutura **stat** está definida em <`sys/types.h`> como se segue:

```
struct stat {
        mode_t      st_mode;/* file mode (type, permissions) */
        ino_t       st_ino;/* i-node number */
        dev_t       st_dev;/* ID of device containing a directory entry for this file */
        dev_t       st_rdev;/* ID of device defined only for char special or block special files */
        short       st_nlink;/* number of links */
        uid_t       st_uid;/* user ID of the file' s owner */
        gid_t       st_gid;/* group ID of the file's group */
        off_t       st_size;/* file size in bytes */
        time_t      st_atime;/* time of the last access */
        time_t      st_mtime;/* time of the last data modification */
        time_t      st_ctime;/* time of the last file status change */
        long        st_blksize;/* optimal I/O block size for filesystem operations*/
        long        st_blocks;/* actual number of 512 byte blocks allocated */
}
```

A componente **st_mode** da estrutura **stat** contém o tipo de ficheiro, as permissões de acesso (9 bits), o set-user-ID bit, o set-group-ID bit e o sticky bit.

Tipos de ficheiros

O tipo de ficheiro codificado em **st_mode** pode ser um dos seguintes:

- **Ficheiro regular** ou **ordinário**. É o tipo mais comum de ficheiro, o qual contém dados sob alguma forma. O sistema UNIX não faz distinção entre dados textuais e dados binários. Qualquer interpretação do conteúdo dum ficheiro ordinário é deixado para as aplicações.
- **Directoria**. Contém os nomes e os i-nodes doutros ficheiros (Um i-node é uma estrutura de dados em disco que contém informação acerca dum ficheiro como, por exemplo, a sua localização em disco.) Qualquer processo que tem permissão de leitura duma directoria pode ler o seu conteúdo, mas só o kernel pode escrever nela.
- **Ficheiro especial de caracteres**. Este tipo de ficheiro é usado para certos tipos de dispositivos I/O.
- **Ficheiro especial de blocos**. É um tipo de ficheiro usado para dispositivos de disco rígido. Todos os dispositivos I/O são ficheiros especiais de caracteres ou de blocos, e que aparecem como ficheiros ordinários no sistema UNIX. A manipulação destes ficheiros depende do dispositivo real que está a ser referenciado.
- **Fifo**. É uma fila de espera first-in first-out, também conhecida por pipeta com nome (*named pipe*) que é usada para comunicação entre processos (interprocess communication ou IPC).
- **Elo simbólico** (*symbolic link*). É um ficheiro que contém o caminho doutro ficheiro.
- **Socket**. É um ficheiro especial usado em comunicação entre processos em rede.

É possível determinar o tipo dum ficheiro através da utilização das seguintes máscaras definidas em <sys/stat.h>:

```
#define   S_IFMT      0170000    /* type of file */
#define       S_IFDIR     0040000    /* directory */
#define       S_IFCHR     0020000    /* character special */
#define       S_IFBLK     0060000    /* block special */
#define       S_IFREG     0100000    /* regular */
#define       S_IFLNK     0120000    /* symbolic link */
#define       S_IFSOCK    0140000    /* socket */
#define       S_IFIFO     0010000    /* fifo */
```

O tipo dum ficheiro pode ser determinado directamente através das seguintes macros definidas em <sys/stat.h>:

```
#define       S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)
#define       S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)
#define       S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)
#define       S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)
#define       S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)
#define       S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)
#define       S_ISSOCK(m)     (((m) & S_IFMT) == S_IFSOCK)
```

em que m é a componente **st_mode** da estrutura **stat**. Note-se que o valor de st_mode é combinado com a máscara **S_IFMT** referente ao tipo de ficheiro.

Permissões de acesso a ficheiros

O valor de **st_mode** também codifica os bits ou permissões de acesso a um ficheiro, qualquer que seja o seu tipo. Há 9 bits de permissão de acesso a qualquer ficheiro divididos em 3 subconjuntos de 3 bits. O primeiro subconjunto de 3 bits estabelece as permissões do propritário do ficheiro. O segundo subconjunto de 3 bits estabelece as permissões do grupo ao qual o proprietário pertence. Por fim, o terceiro subconjunto de 3 bits estabelece as permissões dos restantes utilizadores. Para cada subconjunto de 3 bits, o primeiro bit define a permissão de leitura (r), o segundo bit define a permissão de escrita (w) e o terceiro bit define a permissão de execução (x).

Naturalmente que também há 9 máscaras, cada uma das quais identifica uma das 9 permissões. Além disso, cada subconjunto de 3 bits tem reservada uma máscara.

```
#define S_IRWXU         0000700 /* rwx, owner */
#define     S_IRUSR     0000400 /* read permission, owner */
#define     S_IWUSR     0000200 /* write permission, owner */
#define     S_IXUSR     0000100 /* execute/search permission, owner */
#define S_IRWXG         0000070 /* rwx, group */
#define     S_IRGRP     0000040 /* read permission, group */
#define     S_IWGRP     0000020 /* write permission, grougroup */
#define     S_IXGRP     0000010 /* execute/search permission, group */
#define S_IRWXO         0000007 /* rwx, other */
#define     S_IROTH     0000004 /* read permission, other */
#define     S_IWOTH     0000002 /* write permission, other */
#define     S_IXOTH     0000001 /* execute/search permission, other */
```

Para determinar se um dado ficheiro tem uma dada permissão, há que fazer também a conjunção lógica dos bits da máscara do subconjunto em causa com o valor de **st_mode**. O resultado é naturalmente uma das nove permissões possíveis do ficheiro.

Set-User-ID e Set-Group-ID

Todo o processo tem 6 ou mais IDs associados:

- Real user ID
- Real group ID
- Effective user ID
- Effective group ID
- Saved set-user ID
- Saved set-group ID

Os IDs **reais** identificam a quem pertence o processo na realidade. Estes dois campos são extraídos do registo do utilizador no ficheiro de passwords (/etc/passwd) quando o utilizador em causa dá entrada na máquina. Normalmente, estes valores não mudam durante uma sessão, a não ser que um processo do super-utilizador os altere.
Os IDs **efectivos** referem-se ao ficheiro executável associado ao processo, i.e. determinam as permissões de acesso ao dito ficheiro.
Os IDs **salvaguardados** contêm cópias dos IDs efectivos quando um programa é executado.

Recorde-se que todo o ficheiro tem um proprietário (owner) e um grupo (group), os quais são especificados pelas componentes **st_uid** e **st_gid** da respectiva estrutura **stat**.

Normalmente, quando um ficheiro entra em execução fica associado a um processo, o real user ID e effective user ID são iguais, o mesmo se passando com o real group ID e o effective group ID. Esta associação faz-se de modo que o effective user ID do processo passa a ser o proprietário do ficheiro codificado na componente **st_uid** da estrutura **stat** do ficheiro. Esta associação só é concretizada após a activação do bit **set-user-ID** codificado na componente **st_mode** da estrutura **stat**. Do mesmo modo, o bit **set-group-ID** codificado na componente **st_mode** da estrutura **stat** serve para associar o o effective group ID do processo ao grupo do respectivo ficheiro codificado na componente **st_gid** da estrutura **stat** do ficheiro. Voltaremos a este assunto mais tarde.

Função **umask**

Agora que estão descritas os 9 bits de permissão associados a cada ficheiro, podemos agora descrever a máscara de criação de file mode que está associado a todo o processo.
A função **umask** activa esta máscara para um processo e retorna o valor anterior.

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

mode_t umask(mode_t   cmask);
```

Retorna: o valor anterior da máscara

O argumento *cmask* é formado por um OR bit-a-bit de quaisquer das 9 permissões de leitura, de escrita e execução acima indicadas. Uma máscara de criação de ficheiro é usada sempre que um ficheiro ou directoria é criado. Recorde-se que as funções creat e open aceitam o argumento **mode** que especifica os bits das permissões de acesso a um novo ficheiro. Quaisquer bits que estão *on* na máscara de criação de modo do ficheiro são colocados a *off* no modo do ficheiro. Este é um mecanismo de segurança permitindo assim a criação de ficheiro novos com a segurança que, por exemplo, a permissão de execução ou permissão de leitura para outros não serão ligados.

Exemplo 10.1
O programa seguinte cria dois ficheiros, um com umask a 0 e o outro com umask que inibe todas as permissões de grupo e de outros utilizadores.

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

int main(void)
{
   umask(0);
   if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH) < 0)
         printf("creat error for foo\n");

   umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
   if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH) < 0)
         printf("creat error for bar\n");

   return(0);
}
```

Output:

```
$ umask
02
$ a.out
$ ls –l foo bar
-rw-------    1          agomes   0 Apr  17   16:42                    bar
-rw-rw-rw-   1          agomes   0 Apr  17   16:42                    foo
$ umask
02
$
```

Como se pode constatar, o comando umask (não confundir com a função umask) foi usado antes e depois de executar o programa acima. Note-se que a máscara do processo progenitor (neste caso a sh shell) não é afectada pelas máscaras do processo progénito usadas no programa anterior.

Funções de modificação de permissões de acesso: **chmod** e **fchmod**

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

int chmod(const char *pathname, mode_t  mode);
int fchmod(int filedes, mode_t  mode);
```

Ambas retornam: 0 se OK, -1 em caso de erro

Estas duas funções permitem mudar as permissões de acesso dum ficheiro:

- `int chmod(char *pathname, mode_t mode)`
  Altera as permissões de acesso a um ficheiro. O ficheiro é identificado pelo seu caminho (string) armazenado em `path`. O valor 0 é devolvido em caso de sucesso, ao passo que o valor –1 é devolvido em caso de erro.

- `int fchmod(int filedes, mode_t mode)`
  Altera as permissões de acesso a um ficheiro aberto. O ficheiro é identificado pelo seu descritor `filedes`. O valor 0 é devolvido em caso de sucesso, ao passo que o valor –1 é devolvido em caso de erro.

O parâmetro `mode` estabelece as permissões de acesso através de macros acima indicadas e existentes em <sys/stat.h>, ou através da atribuição de números octais de 3 dígitos. O dígito à direita especifica os privilégios do proprietário (ou owner) do ficheiro, o dígito do meio especifica os privilégios do grupo a que pertence o proprietário do ficheiro, e o dígito à esquerda especifica os privilégios dos outros utilizadores. O equivalente binário de cada dígito octal é um número binário de 3 bits. O bit à direita especifica o privilégio de execução, o bit intermédio especifica o privilégio de escrita e o bit à esquerda especifica o privilégio de leitura.

Por exemplo:

4 (octal 100) = read only;
2 (octal 010) = write only;
6 (octal 110) = read and write;
1 (octal 001) = execute only.

Assim, o modo de acesso 600 fornece ao proprietário os privilégios de read e write, mas nenhuns privilégios a quaisquer outros utilizadores, ao passo que o modo de acesso 666 dá privilégios de read e write a toda a gente.

Exemplo 10.2
Pretende-se activar a permissão de escrita e desligar a permissão de execução de grupo do ficheiro foo. Pretende-se ainda activar o modo absoluto "rw-r—r--" no ficheiro bar.

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

int  main(void)
{
   struct  stat  statbuf;

  /* turn on set-group-ID and turn off group-execute */

   if (stat("foo", &statbuf) < 0)
         printf("stat error for foo\n");
   if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_IWGRP) < 0)
         printf("chmod error for foo\n");

   /* set absolute mode to "rw-r--r--" */

   if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
         printf("chmod error for bar\n");

   return(0);
}
```

Output:

```
$ touch foo bar              //O Comando touch pode ser usado para criara ficheiro vazios
$ chmod 650 bar foo
$ ls –l foo bar
$ a.out
$ ls –l foo bar
-rw-r--r--    1                  crocker   0 May  2   16:00                              bar
-rw-rw-rw-  1                    crocker   0 May  2   16:00                              foo
$
```

Exercício 10.1
Escreva um programa que identifique o tipo dum dado ficheiro. O nome do ficheiro é passado por argumento
Requisito: o ficheiro deve ser previamente criado em disco.

Exercício 10.2
(i) Escreva um programa que adicione as permissões de execução dum dado ficheiro.
(ii) Escreva um programa que retire as permissões de execução dum dado ficheiro.
Requisito: o ficheiro deve ser previamente criado em disco.

Exercicio 10.3
O que que dizer e para que servem os flags de "Sticky bit" e o "Group ID" (S_ISGID)

Funções de modificação de proprietário e de grupo: **chown**, **fchown** e **lchown**

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

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int filedes, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Todas retornam: 0 se OK, -1 em caso de erro

Estas funções permitem alterar o user-ID e o group-ID dum ficheiro. Mas se o ficheiro é referenciado por um elo simbólico (symbolic link), a função lchown modifica o proprietário do elo simbólico, não o do próprio ficheiro. Note-se que, em geral e à excepção do superutilizador, um utilizador só poderá alterar a propriedade e o grupo dos seus próprios ficheiros.

Elos para ficheiros

Para perceber a utilidade de elos para ficheiros, há que compreender em primeiro lugar a organização dum sistema de ficheiros.

Um disco tem uma ou mais partições e cada partição contem um sistema de ficheiros (Figura 10.1).

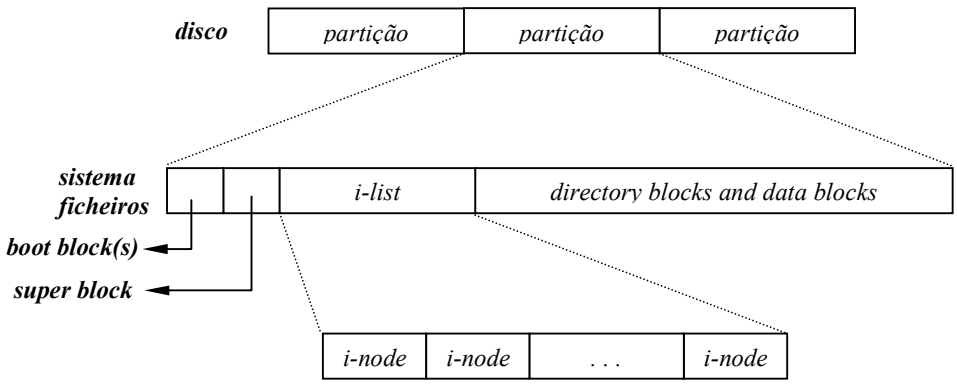

Figura 10.1: Disco, partições e um sistema de ficheiros.

Cada ficheiro tem um **i-node** e é identificado por um i-node number no sistema de ficheiros. Um **i-node** é um registo de tamanho fixo que contêm a maior parte de informação acerca dum ficheiro. Os i-nodes fornecem informações importantes sobre ficheiros nomeadamente: propriedade individual (owner) e de grupo (group), modo de acesso (permissões read, write e execute) e tipo de ficheiro.

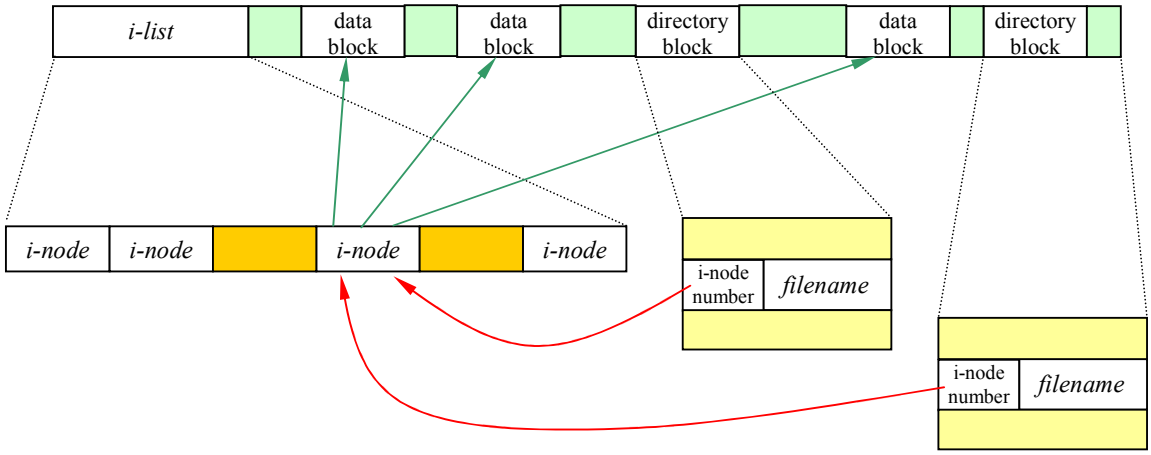

Figura 10.2: Sistema de ficheiros com mais detalhe.

A Figura 10.2 mostra o seguinte:

Dois itens em directorias podem apontar para o mesmo i-node. Cada i-node tem um contador de elos (link count) que contém o número de itens em directorias que apontam para ele. Quando este contador vai a zero, o respectvo ficheiro pode ser apagado, i.e. os blocos de dados (data blocks) associados ao ficheiro podem ser libertados. Isto explica porque a operação de "cortar o elo a ficheiro" não significa necessariamente "apagar os blocos associados ao dito ficheiro". Na estrutura stat, o contador de elos está contido na componente **st_link**, a qual é uma variável do tipo nlink_t. Estes elos são os chamados **elos fortes** (hard links).

Há outro tipo de elos, designado por **elo simbólico** (symbolic link). Neste caso, o conteúdo do ficheiro (os blocos de dados) contém o nome do ficheiro para o qual o elo aponta. No seguinte exemplo,

```
lrwxrwxrwx                1                              root    7 Sep 25
                         07:14                   lib -> usr/lib
```

o ficheiro `lib` é um elo simbólico (tipo de ficheiro `l`) que contém 7 bytes referentes ao nome `usr/lib` do ficheiro apontado.

Portanto, um elo simbólico não é mais do que um ponteiro indirecto para um ficheiro. Por outro lado, um elo (forte) aponta directamente para o i-node dum ficheiro.

O i-node contém toda a informação acerca dum ficheiro: tipo de ficheiro, bits de permissão de acesso, tamanho do ficheiro, ponteiros para os blocos de dados do ficheiro, etc. A maior parte da informação na estrutura stat é obtida a partir do i-node do ficheiro. Só dois itens são armazenados numa directoria: o nome do ficheiro e o seu i-node number.

Funções de manipulação de elos: **link**, **unlink**, **remove** e **rename**

Já vimos que qualquer ficheiro pode ser apontado por vários verbetes (entries) existentes em directorias. A forma de criar um elo para um ficheiro é através da função **link**. Do mesmo modo, o corte dum elo para um ficheiro é feito através da chamada da função **unlink**.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
int unlink(const char *pathname);
```

Ambas retornam: 0 se OK, -1 em caso de erro

Exemplo 10.3

O seguinte programa abre o ficheiro `teste.txt` e, de seguida, corta o elo (forte) que ele tem na directoria corrente. Isto é, o ficheiro `teste.txt` deixará de existir na directoria corrente. Antes de terminar, o processo que executa o programa é colocado a dormir durante 15 segundos.

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

int
main(void)
{
    if (open("teste.txt ", O_RDWR) < 0)
                                        printf("open error");

    if (unlink("teste.txt ") < 0)
                                        printf("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

A função **unlink** é muitas usada para assegurar que um eventual ficheiro temporário num programa não seja deixado ao acaso numa situação de *crash* do programa. O processo cria um ficheiro através da função **open** ou da função **creat**, e depois chama de imediato a função **unlink**. No entanto, o ficheiro não é logo eliminado porque ainda se encontra aberto. Só quando o processo fecha (**close**) o ficheiro ou termina (o que faz com que o kernel feche todos os seus ficheiros abertos) é que o ficheiro é eliminado.

Note-se que se o *pathname* é um elo simbólico, a função unlink corta o elo ao ficheiro que é o elo simbólico, não ao ficheiro apontado pelo elo simbólico.

Também se pode cortar o elo (unlink) a um ficheiro ou directoria com a função remove. No caso dum ficheiro, a função **remove** é idêntica à função **unlink**. No caso duma directoria, a função **remove** é idêntica à função **rmdir**.

```
#include <stdio.h>

int remove(const char *pathname);
int rename(const char *oldname, const char *newname);
```

Ambas retornam: 0 se OK, -1 em caso de erro

Funções de manipulação de elos simbólicos: **symlink** e **readlink**

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath;
int readlink(const char *pathname, char *buf, int bufsize);
```

`symlink` retorna: 0 se OK, -1 em caso de erro
`readlink` retorna: número de bytes se OK, -1 em caso de erro

Um elo simbólico é criado pela função **symlink**. Um novo item `sympath` é criado na directoria corrente que passa a apontar para `actualpath`. Não é necessário que `actualpath` exista quando o elo simbólico é criado. Além disso, `actualpath` e `sympath` não precisam de residir no mesmo sistema de ficheiros.

A função readlink serve para ler o conteúdo do elo simbólico, i.e. ler o nome do ficheiro apontado pelo elo simbólico. Note-se que este conteúdo não é null-terminated. Esta função combina as funções open, read e close de manipulação de ficheiros.

Função de temporização de ficheiros: **utime**

Há três itens temporais associados a cada ficheiro: **st_atime** (last-access time of file data), **st_mtime** (last-modification time of file data) e **st_ctime** (last-change time of i-node status). O primeiro fornece o tempo do último acesso ao ficheiro, o segundo fornece o tempo da última alteração ao conteúdo do ficheiro e, finalmente, o terceiro refere-se a qualquer alteração do estado (permissões, user-ID, etc) do ficheiro, mas não do seu conteúdo.

O tempo de acesso e o tempo de modificação dum ficheiro podem ser alterados através da função **utime**.

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

int utime(const char *pathname, const struct utimbuf *times);
```

Retorna: 0 se OK, -1 em caso de erro

A estrutura usada por esta função é a seguinte:

```
struct utimbuf  {
    time_t  actime; /* access time  */
    time_t  modtime;    /* modification time  */
}
```

Funções de manipulação de directorias: **mkdir** e **rmdir**

Uma directoria (vazia) pode ser criada através da invocação da função mkdir:

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

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

Retorna: 0 se OK, -1 em caso de erro

Os verbetes (entries) relativos às directorias . e .. são automaticamente criadas. As permissões de acesso ao ficheiro, ou seja mode, são modificadas pela máscara de criação de modo de ficheiro do processo em execução.

Um erro comum na criação duma directoria é especificar um *mode* idêntico ao da criação dum ficheiro (só permissões de leitura e escrita), mas pelo menos um bit de execução deve ser activado para que haja acesso aos ficheiros existentes numa directoria.

Para remover uma directoria usa-se a seguinte função:

```
#include <unistd.h>

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

Retorna: 0 se OK, -1 em caso de erro

Funções de leitura de directorias

Uma directoria pode ser lida por qualquer utilizador que tenha permissão para lê-la.
Mas, só o kernel pode escrever na directoria por forma a preservar a sanidade do sistema de ficheiros.

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

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

Retorna: ponteiro se OK, NULL em caso de erro

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

Retorna: ponteiro se OK, NULL em caso de erro

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

Retorna: 0 se OK, -1 em caso de erro

A estrutura `dirent` está definida no ficheiro <`dirent.h`> e tem, entre outros, os seguintes campos:

```
struct dirent
{
    ino_t                          d_ino;
    /* i node number */
    char                       d_name[NAME_MAX+1];      /* null-terminated filename */
}
```

No entanto, esta estrutura é dependente da implementação do sistema operativo.
A estrutura DIR é uma estrutura interna que é usada pelas quatro funções anteriores. Esta estrutura guarda a informação acerca da directoria que está a ser lida. Tem um propósito semelhante à estrutura FILE para ficheiros.

O ponteiro para uma estrutura DIR que é devolvido pela função **opendir** é depois usado pelas outras três funções.

Exemplo 10.4:
Uma versão do comando **ls** do Unix.

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

int main(int argc, char **argv)
{
    DIR *dp;
    struct dirent *dirp;

    if (argc != 2)
            fprintf(stderr, "Usage: %s Um argumento é necessário\n", argv[0]);

    if ((dp = opendir(argv[1])) == NULL)
            fprintf(stderr, "Can't open %s\n", argv[1]);

    while ((dirp = readdir(dp)) != NULL)
                        printf("%s\n", dirp->d_name);

    closedir(dp);
}
```

Exemplo 10.5:
Outra versão do comando **ls** do Unix.
As funções a usar são as seguintes:

- *Scans* o conteúdo duma directoria:
  `scandir(char *dirname,struct direct *namelist,int(*select)(),int(*compar)())`
  A função **`scandir`**() , onde os argumentos são:
  - `dirname` identifica a directoria a ser lida;
  - `namelist` é um ponteiro para um array de ponteiros para estruturas (ou structs);

- `(*select)()` é um ponteiro para uma função que é chamada com um ponteiro para uma directory entry (definida em <sys/types>) e que devolve um valor não-nulo se a directory entry está incluída no array. Se este ponteiro é NULL, então todas as directory entries serão incluídas.
- `(*compar)()` é um ponteiro para uma rotina que é passada para o qsort (o quicksort que ordena os elementos dum array). Se o ponteiro é NULL, o array não é ordenado.

- Ordena alfabeticamente um array :
  `alphasort``(struct direct **d1, **d2)`

Exemplo 10.6:

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

int file_select(struct  dirent  *entry);                    /* tirar os ficheiros . e .. da listagem  */
char pathname[MAXPATHLEN];

main()
{
        int count, i;
        struct dirent **files;

        if (getwd(pathname)==NULL)
        {
                                printf("Error getting path\n");
                                        exit(0);
        }
        printf("Current working directory = %s\n", pathname);
        count=scandir(pathname,&files,file_select,alphasort);

        /* If no files found, make a non-selectable menu item */
        if (count <= 0)
        {
                                printf("No files in this directory\n");
                                        exit(0);
        }
        printf("Number of files = %d\n",count);
        for (i=1;i<count+1;++i)
                                printf("%s",files[i-1]->d_name);
        putchar("\n");
}

int file_select(struct dirent *entry)
{
        if ((strcmp(entry->d_name,".") == 0) || (strcmp(entry->d_name, "..")==0))
                                        return(FALSE);
        return(TRUE);
}
```

A função `scandir`() também devolve '.' (da directoria corrente) e '..' (da directoria imediata e hierarquicamente superior), mas são excluídas pela função `file_select`().
As funções `scandir`() e `alphasort`() têm os seus protótipos em <`sys/types.h`> e <`sys/dir.h`>, respectivamente.
As definições de `MAXPATHLEN` e `getwd`() encontram-se em <`sys/param.h`>.

Exercício 10.3:
Altere o programa anterior de modo a listar somente os ficheiros com extensão .c, .o ou .h.

Exercício 10.4:
Escreva um programa **sv** com uma funcionalidade semelhante ao comando **cp** do Unix. A sua sintaxe é a seguinte: `sv <file1> <file2> … <filen> directory`
O comando sv copia os ficheiros `<file1> <file2> … <filen>` para a directoria `directory` mas só se forem mais recentes que as versões já existentes em `directory`.

```
/* sv.h file */
#include <stdio.h>                /*  headers  */
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

#ifdef UNIX                        /*  Windows or Unix  */
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#else
#include <io.h>
#include <sys/stat.h>
#endif

void sair(char *s1, char *s2);
void sv(char *file, char *dir);
char *progname;
```

```
int main(int argc, char **argv)
{
        int i;
        struct stat stbuf;
        char *dir=argv[argc-1];
        progname=argv[0];

        if (argc<3)
            sair("Utilizacao file1 file2 … filen directory\n", " ");
        if (stat(dir,&stbuf)==-1)
            sair("Nao consegue aceder a directoria %s\n", dir);
        if ((stbuf.st_mode & S_IFMT) != S_IFDIR)
            sair("%s nao e uma directoria\n",dir);
        for (i=1;i<argv-1;i++)
            sv(argv[i],dir);
        return 0;
}
```

```
void sair(char *s1, char *s2)              //Função para sair quando surge um erro
{                                          //Esta função devia ser melhorada e explicada – Um Voluntário?
        extern int errno, sys_nerr;
        extern char *progname;

        #ifdef UNIX
        extern char *sys_errlist[];
        #else
        extern char *_sys_errlist[];
        #endif

        fprintf(stderr, "%s", progname);
        fprintf(stderr,s1,s2);

        if (errno>0 && errno<sys_nerr)
        #ifdef UNIX
        fprintf(stderr, "(%s)", sys_errlist[errno]);
        #else
        fprintf(stderr, "(%s)", _sys_errlist[errno]);
        #endif
        fprintf(stderr, "\n");
        exit(1);
}
```

```
void sv(char *file, char *dir)
{
        struct stat stin, stout;
        int fin, fout, n;
        char target[BUFSIZ], buf[BUFSIZ];

        sprintf(target, "%s/%s", dir, file);

        if ((strchr(file, '/'))!=NULL)  sair("Nao ha tratamento do / no ficheiro %s", file);
        if (stat(file,&stin)==-1)      sair("Nao consegue stat no ficheiro: %s", file);

        if (stat(target,&stout)==-1)   /* target nao existe */
        stout.st_mtime=0;

        if (stin.st_mtime <= stout.st_mtime)
                        fprintf(stderr, "%s nao foi copiado\n", file);
        else{
             fin = open(file,0);
             fout = creat(target,stin.st_mode);
             while ((n=read(fin,buf,sizeof(buf)))>0)
                        if (write(fout,buf,n)!=n)
                                sair("Erro a escrever %s", target);
             fprintf(stderr, "%s copiado\n", file);
             close(fin);
             close(fout);
        }
}
```

Funções **chdir**, **fchdir** e **getcwd**

Todo o processo tem uma directoria corrente. Esta directoria é onde a pesquisa de qualquer caminho (pathname) começa. Quando um utilizador entra no sistema, a directoria corrente de trabalho começa normalmente por ser a directoria especificada no sexto campo do verbete do utilizador no ficheiro **/etc/passwd**, também conhecida por directoria base do utilizador.
A directoria corrente é um atributo dum processo; a directoria de base é um atributo dum nome de utilizador autorizado (login name).

A directoria corrente pode ser alterada por um processo através das seguintes funções:

```
#include <unistd.h>

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

Ambas retornam: 0 se OK, -1 em caso de erro

Note-se que a directoria corrente é um atributo dum processo, o que significa que a directoria corrente não pode ser afectada por qualquer sub-processo que execute a função **chdir**.
Por exemplo, no Exemplo 10.7, a mudança de directoria não provoca alteração de directoria no processo progenitor que é a Bourne ou a C-shell. Isto é, quando o programa termina a execução, o controlo volta à shell e a directoria corrente antes da execução do programa do Exemplo 10.7.

Exemplo 10.7:

```
#include <stdio.h>
#include <unistd.h>

char cur[100];

main(int argc, char **argv)
{
    if (argc < 2)
    {
       printf("Usage: %s <pathname>\n", argv[0]);
         exit(1);
    }

    getwd(cur)

    if (chdir(argv[1]) != 0)
    {
        printf("Error in chdir\n");
           exit(1);
    }

    printf("Mudamos da directoria %s\n", cur);
    printf("para a directoria %s\n", getwd(cur));

}
```

A função que obtém a directoria corrente é a seguinte:

```
#include <unistd.h>

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

Retorna: *buf* se OK, NULL em caso de erro

Exercício 10.5:
Escreva um programa semelhante ao comando **ls –l** do Unix que mostre todos os ficheiros duma directoria, incluindo as suas permissões.
(Nota: Não deve usar a função `exec`.)

Exercício 10.6:
Escreva um programa semelhante ao comando **grep** do Unix que mostre todas as linhas dum ficheiro que contêm uma dada palavra.

Exercício 10.7:
Escreva um programa que mostre todos os ficheiros duma directoria e das suas subdirectorias.

Exercício 10.8:
Escreva um programa que iniba a permissão de leitura dum ficheiro ao grupo a que pertence o seu proprietário.