2.0 STRUTTURA STAT

In un file system di tipo Linux abbiamo a disposizione diversi tipi di file:

- **Regular file**: dal punto di vista del kernel un file regolare contenente testo oppure binario.
- **Directory file**: contiene nomi e puntatori ad altri file; solo il kernel può scriverci per motivi di sicurezza.
- **Character special file**: usato per individuare alcuni dispositivi del sistema per esempio il display, la tastiera (/dev/tty), etc…
- **Block special file**: usato per individuare i dischi, ad esempio /dev/sda1 che indica la prima partizione del disco sda.
- **Pipe e FIFO**: usati per la comunicazione tra processi.
- **Symbolic link**: un tipo di file che punta ad un altro file.
- **Socket**: usato per la comunicazione in rete tra processi.

Per ogni file che abbiamo all'interno del file system dobbiamo avere delle informazioni che ci permettono di usare al meglio i file. In particolare:

- **Tipo di file**: (normale, directory, link, speciale) utile alle system call per permettere di capire con che file bisogna lavorare.
- **Permessi**: (lettura, scrittura, esecuzione) ci permettono cosa possiamo fare su un certo file.
- **Tempo ultimo accesso, modifica e cambiamento dello stato (permessi)**.
- **User ID e Group ID del proprietario del file**.
- **Numero di link che puntano al file**

La struttura contiene al suo interno tutte le informazioni descritte in precedenza. Ad esempio, vediamo il primo campo di questa struttura che è ***st_mode***: contiene al suo interno informazioni quali i permessi e il tipo di file. Il secondo campo, ***st_ino***, il numero di i-node che contiene queste informazioni. ***st_dev*** il numero di device, cioè su quale partizione del nostro file system si trova. ***st_nlink*** il numero di link, ***st_uid*** e ***st_gid*** lo user ID e il group ID del proprietario. ***st_size*** ci fornisce la size in bytes del nostro file. ***st_blocks*** il numero di blocchi allocato per il file,. ***st_blksize*** ci dice la dimensione del blocco ottimale per le operazioni di I/O.

```
struct stat {
  mode_t   st_mode;     /* file type & mode (permissions) */
  ino_t    st_ino;      /* i-node number (serial number) */
  dev_t    st_dev;      /* device number (filesystem) */
  dev_t    st_rdev;     /* device number for special files */
  nlink_t  st_nlink;    /* number of links */
  uid_t    st_uid;      /* user ID of owner */
  gid_t    st_gid;      /* group ID of owner */
  off_t    st_size;     /* size in bytes, for regular files */
  time_t   st_atime;    /* time of last access */
  time_t   st_mtime;    /* time of last modification */
  time_t   st_ctime;    /* time of last file status change */
  long     st_blksize;  /* best I/O block size */
  long     st_blocks;   /* number of 512-byte blocks allocated */
};
```

Il campo ***st_mode*** è l'unico campo di questa struttura che contiene 2 informazioni al suo interno. In questo campo sono inclusi i 9 bit che regolano i permessi di accesso al file cui esso si riferisce →

st_mode mask	Meaning
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	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

2.0.1 stat, fstat e lstat

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

Questa system call ci permette di leggere le informazioni relative ad un file che sono memorizzate nell'i-node. Questa system call ha diverse varianti in questo caso rispetto alla **stat** ci sono anche **fstat** e **lstat**.

La **stat** prende come primo parametro un nome di un file e poi passa come secondo parametro un puntatore ad una struttura stat che è stata precedentemente dichiarata. L'effetto è che le informazioni contenute nell'i-node del file, vengono ricopiate all'interno di *buf.* Questa invocazione restituisce 0 se è andato tutto bene, -1 in caso di errore.

La **fstat** si usa quando non si ha a che fare con un nome di un file, ma con un file descriptor che magari è stato dichiarato in precedenza. La funzionalità è la stessa della **stat**.

La **lstat**, che per il momento citiamo solamente, ha a che fare con i link simbolici. Se il primo parametro che passiamo è un link simbolico, ci darà le informazioni sul link simbolico e non sul file puntato.

Nell'uso reale della **stat,** bisogna dichiarare prima la struttura e poi passare il puntatore. Se non viene dichiarata prima, si corre il rischio di non avere a disposizione dello spazio allocato.

L'utilizzatore tipico di queste informazioni che vengono caricate è il comando **ls -l** in quanto le informazioni che ricaviamo da questa chiamata sono contenute nell'i-node.

Se volessimo scoprire di quale tipo è un file che viene passato come parametro, possiamo usare delle **Macro**, che sono funzioni booleane che aiutano ad identificare il tipo di un file verificando ciò che è contenuto nel campo **st_mode** della struttura stat del file. Ne sono 7, una per ogni tipo di file, per verificare se un determinato file è un regular file, una directory e così via. La funzionalità pratica banalmente è la seguente: dalla struttura **stat** recuperiamo il campo *st_mode* e viene passato a queste Macro. Queste macro daranno vero (valore di ritorno 1) se il tipo di file che abbiamo passato è uguale alla Macro che abbiamo invocato, falso altrimenti (S_ISREG(buf.st_mode)).

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link (not in POSIX.1 or SVR4)
S_ISSOCK()	socket (not in POSIX.1 or SVR4)

Questa porzione di codice prende da terminale un nome di un file, e all'interno di un if verifica che la lstat che prende in input il nome del file passato da terminale e il puntatore buf (la struttura è stata creata precedentemente) sia minore di 0, nel caso errore.
Successivamente c'è una sequenza di if else if per verificare il file di che tipo è. Questo lo si fa perché sicuramente uno e un solo else if sarà vero e verrà stampato il tipo del file.

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


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

 for (i=1;i<argc;i++){
  printf("%s:", argv[i]);
  if (lstat(argv[i],&buf) <0) {
              printf("lstat error");
              continue;
      }
  if (S_ISREG(buf.st_mode))  printf("regular");
  else if (S_ISDIR(buf.st_mode)) printf("directory");
  else if (S_ISCHR(buf.st_mode)) printf("character special");
  else if (S_ISBLK(buf.st_mode)) printf("block special");
       ..............
    }
   exit(0);
}
```

Vediamo ora quelli che sono gli ID associati ai processi. Così come ci sono degli ID associati ad ogni file (accessibili attraverso *st_uid, st_gid*) per dire chi sia il proprietario, ovviamente ci sono anche degli ID associati ai processi (ne sono 6) che sono: **real u/g ID, effective u/g ID, saved set-u/g-ID**. Quello che capita nella maggioranza dei casi è che il real u ID coincide con l'effective u ID, cioè chi sta eseguendo il nostro processo (ID dell'utente) sarà uguale sia a real u ID, sia a effective u ID.

Il **real**, come suggerisce il nome, ci dice chi siamo realmente nel nostro sistema.

L'**effective** determina i permessi di accesso ai file. Avere accesso ad un file, capire quindi chi è l'utente e chi è il suo gruppo, tutto ciò si fa effettuando un confronto tra l'ID del proprietario del file (presente nell'i-node, *st_uid*, *st_gid*), con l'ID del processo che tenta di accedere al file. Per fare questo test non si usa il real user ID, ma l'effective user ID.

Per i programmi standard eseguiti normalmente, l'effective user ID coincide sempre con il real user ID. C'è però un flag speciale nel campo **st_mode**, che se viene settato, fa in modo che l'effective invece di essere uguale al real, diventa uguale al proprietario (o group) dell'eseguibile. Questi bit possono essere testati usando le costanti S_ISUID e S_ISGID.

Facciamo un esempio per capire meglio la situazione:

Supponiamo di avere a disposizione un file chiamato **pippo.doc** che è un file dell'utente pippo sul quale solo pippo può scrivere (0644). Prendiamo a disposizione un eseguibile detto **scrivi** che è un word-processor di pippo che può essere usato da tutti. La prima domanda che ci poniamo è: pippo può modificare pippo.doc usando scrivi? Chiaramente si, in quanto sia il file sia l'editor sono di sua proprietà.

La domanda più interessante è: l'utente pluto può modificare pippo.doc usando scrivi di pippo? Sembrerebbe di no, a meno che scrivi abbia il set-user-id flag settato, poiché in questo modo chiunque esegua scrivi, verrà visto come pippo. Questa caratteristica è utilizzata quando cambiamo la nostra password del sistema, in quanto la password è modificabile solo da root. Settando il set-user-id, il sistema ci vedrà come root e ci permetterà di cambiare la password.

Tutto ciò serve per capire come un processo accede ad un file. Gli ID del proprietario sono una proprietà del file, infatti sono contenuti nell'i-node e di conseguenza c'è un campo nella struct stat. Gli effective ID sono proprietà del processo che utilizza quel file. Per aprire un file (lettura o scrittura) bisogna avere permesso di esecuzione in tutte le directory contenute nel path assoluto del file. Per creare un file bisogna avere permessi di scrittura ed esecuzione nella directory che conterrà il file.

Fatte queste premesse, ogni volta che si cerca di accedere ad un file viene eseguito un vero e proprio algoritmo:

- eff. uid =0 --> accesso libero
- eff. uid = owner ID
  - accesso in accordo ai permessi "*User*"
- eff. gid = group ID
  - accesso in accordo ai permessi "*Group*"
- accesso in accordo ai permessi "*Other*"

Per prima cosa si controlla che il processo che vuole accedere al file abbia un effective uid=0, nel caso allora l'accesso è libero. Nel caso in cui fallisce la condizione precedente, si effettua un ulteriore controllo: se l'effective uid= owner ID allora si avrà l'accesso in accordo ai permessi "User". Se fallisce anche questo controllo, controllo se l'effective gid= group ID, cioè se chi vuole accedere al file faccia parte almeno dello stesso gruppo del proprietario del file. In caso positivo si ha l'accesso al fine in accordo ai permessi "Group". Se fallisce anche questo controllo allora significa che si rientra nella categoria Other e di conseguenza l'accesso al file si ha in accordo ai permessi "Other".

Supponiamo di avere un file test che ha i seguenti permessi:

_r_ _ r w _ r w _ (466)

Chi può modificare il file? Tutti tranne il proprietario.

Ogni volta che si crea un nuovo file, l'uid è settato come l'effective ID del processo che sta creando il file, cioè significa che un processo che sta creando un nuovo file dovrà crearlo come se fosse un proprio processo quindi uguale all'effective.

Il gid è il group ID della directory nel quale il file è creato oppure il gid del processo.

2.0.2 access

```
#include <unistd.h>
int access (const char *pathname, int mode);
```

Questa system call serve a verificare se il **real** ID ha accesso al file *pathname* nella modalità specificata da *mode*. Restituisce 0 se OK, -1 in caso di errore.

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Questa porzione di codice controlla se l'utente che esegue il processo per un certo file di cui passiamo il *pathname* da linea di comando abbia i permessi di lettura.
Successivamente prova ad aprire il file in lettura. Vediamo l'esempio di questo eseguibile:

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

  int main(int argc, char *argv[])
  {
      if (argc != 2)
            printf("usage: a.out <pathname>");

      if (access(argv[1], R_OK) < 0)
            printf("access error for %s", argv[1]);
      else
            printf("read access OK\n");

      if (open(argv[1], O_RDONLY) < 0)
            printf("open error for %s", argv[1]);
      else
            printf("open for reading OK\n");

      exit(0);
  }
```

Primo caso:
a.out che è di Rescigno come proprietà. Se lanciamo a.out passando come argomento a.out abbiamo che i permessi di lettura si hanno e quindi il file viene aperto normalmente. Questo succede perché rescigno lancia qualcosa di rescigno.

```
$ ls -l  a.out
$ -rwxrwxr-x 1 rescigno 1234 jan 18 08:48 a.out
$ a.out a.out
read access OK
open for reading OK
```

Secondo caso:
prendiamo il file di un altro utente, basile, che ha come permessi rw solo per il proprietario. Se facciamo a.out su prova avremo access error in quanto l'utente rescigno non può accedere al seguente file in lettura come user id.

```
$ ls -l  prova
$-rw------- 1 basile 1234 jan 18 15:48 prova
a.out  prova
access error for prova: Permission denied
open error for prova: Permission denied
```

Terzo caso:
con il comando *su* si diventa momentaneamente super user. Quello che si fa è modificare il proprietario di un file. Con *chown* abbiamo fatto diventare a.out di basile. Successivamente è stato aggiunto il set user id bit, il quale permette nel momento in cui si esegue un eseguibile di essere visti come effective uid come se fossi il proprietario del file. A questo punto, se si lancia a.out su prova la access fornisce un valore minore di 0, di conseguenza access error, perché ricordiamo che la access verifica il real ID. E' vero che noi precedentemente abbiamo settato il set user id, ma esso ci permette di essere visti come effective uid, no real. L'accesso al file avviene normalmente secondo l'algoritmo di apertura di un file.

2.0.3 chmod e fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Ci consentono di modificare i bit di permesso del primo argomento. Come negli altri casi è possibile attraverso il nome del file oppure il fd del file. Restituiscono 0 se OK, -1 in caso di errore.
Per cambiare i permessi, l'effective uid del processo deve essere uguale all'owner del file, cioè deve essere un mio file, o il processo deve avere i permessi di root.
Il *mode* è specificato come l'OR bit a bit di costanti che rappresentano i vari permessi. Questi sono elencati nella tabella qui di fianco.

mode	Description	
S_ISUID	set-user-ID on execution	4000
S_ISGID	set-group-ID on execution	2000
S_ISVTX	saved-text (sticky bit)	1000
S_IRWXU	read, write, and execute by user (owner)	700
S_IRUSR	read by user (owner)	400
S_IWUSR	write by user (owner)	200
S_IXUSR	execute by user (owner)	100
S_IRWXG	read, write, and execute by group	070
S_IRGRP	read by group	040
S_IWGRP	write by group	020
S_IXGRP	execute by group	010
S_IRWXO	read, write, and execute by other (world)	007
S_IROTH	read by other (world)	004
S_IWOTH	write by other (world)	002
S_IXOTH	execute by other (world)	001

Questa porzione di codice ci permette di capire che ci sono 2 modi diversi di modificare i permessi: il primo chmod che prende in input il nome di un file *foo*, mentre il secondo parametro, nella parentesi c'è un AND tra statbuf.st_mode che rappresenta i permessi attuali e ~S_IXGRP che rappresenta il negato di S_IXGRP (dunque tutti 1 e 0 nel campo dell'eseguibilità del gruppo). Questo AND mi consente di eliminare l'esecuzione per il gruppo. Dopo questo AND c'è un OR con S_ISGID, cioè set-group-ID on execution, poiché è un OR stiamo aggiungendo questo permesso.
L'altra modalità invece la vediamo applicata sul file "bar", in cui si passa una sequenza di permessi. I permessi di "bar" sono quelli indicati. Con questa modalità i permessi diventeranno il classico 0644.

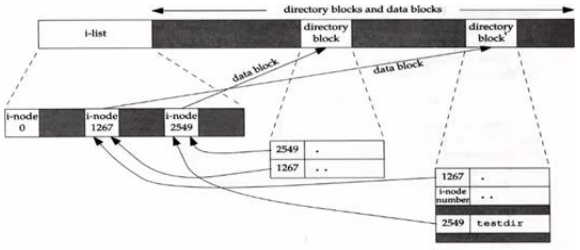
```
#include      <sys/types.h>
#include      <sys/stat.h>
int main(void)
{
   struct stat      statbuf;
   /* turn on set-group-ID and turn off group-execute */
   if (stat("foo", &statbuf) < 0)
       printf("stat error for foo");
   if (chmod("foo",(statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
       printf("chmod error for foo");

   /* set absolute mode to "rw-r--r--" */
   if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
       printf("chmod error for bar");
```

2.0.4 chown

```
#include <sys/types.h>
#include <sys/stat.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Questa system call è l'equivalente di chmode solo che serve a cambiare il proprietario del primo argomento e li setta uguale a owner e group.
Quando si crea una directory, vengono sempre creati 2 file speciali: **dot** (.) e **dot dot** (..) che servono al file system per poter gestire quelle che chiamiamo directory padre e directory figlio.

Consideriamo la seconda directory block nell'immagine. Supponiamo di creare una nuova directory che si chiama *testdir*. Quando viene creata le viene assegnata un i-node number (2549). Questo vuol dire che questa directory punterà all'i-node 2549. Visto che abbiamo creato una directory, l'i-node punterà a quello che chiamiamo directory block. Il contenuto della directory block sono 2 record: dot (puntatore a se stesso) e dot dot (descrive la directory padre). Il padre della directory che ha i-node 2549 è l'i-node 1267. Tutto questo ha portato che appena creiamo una nuova directory, creo un **hard link** che permette dalla directory figlio di risalire alla directory padre e viceversa.



Ogni i-node ha un contatore di link che contiene il numero di directory entry che lo puntano. **Solo** quando scende a zero, allora il file può essere cancellato (i blocchi sono rilasciati mediante funzione **ulink**). Il contatore è nella struct stat nel campo *st_nlink*.

Un file può avere più di una directory entry che punta al suo i-node, cioè più hard link il cui numero è contenuto in *st_nlink*. Gli hard link possono essere creati usando la funzione **link**.

2.0.5 link

```
#include <unistd.h>
int link(const char *path, const char *newpath);
```

Questa system call consente di creare una nuova directory entry *newpath* che si riferisce a *path.* Restituisce 0 se OK, -1 in caso di errore (anche se *newpath* già esiste).

Con la link crea automaticamente il nuovo link ed incrementa anche di uno il contatore dei *link* ***st_nlink*** (questa è un'operazione atomica). Il vecchio ed il nuovo link, riferendosi allo stesso i-node, condividono gli stessi diritto di accesso al "file" cui essi si riferiscono.

2.0.6 unlink

```
#include <unistd.h>
int unlink(const char *pathname);
```

Questa system call banalmente rimuove la directory entry specificata da *pathname* e decrementa il contatore dei link del file. Restituisce 0 se OK, -1 in caso di errore. È consentita solo se si hanno i permessi di scrittura ed esecuzione nella directory dove è presente la directory entry.

Attenzione a queste due condizioni:

- Se tutti i link ad un file sono stati rimossi e nessun processo ha ancora il file aperto, allora tutte le risorse allocate per il file vengono rimosse e non è più possibile accedere al file.
- Se però uno o più processi hanno il file aperto, quando l'ultimo link è stato rimosso, pur essendo il contatore dei link a 0 il file continua ad esistere e sarà rimosso solo quando tutti i riferimenti al file saranno chiusi.

Con questa porzione di codice, si apre un file *tempfile*. Subito dopo con l'unlink lo si cancella, con la printf("file unlinked") ci serve a notificare sullo standard output questa operazione. Subito dopo c'è una sleep di 15 secondi che serve ad addormentare, bloccare temporaneamente, il nostro processo. Dopo 15 secondi verrà stampato "done". Questa operazione serve a far si che nel momento in cui si fa un run di questo programmino, nei 15 secondi che abbiamo a disposizione dalla sleep, osserveremo che il file non è più presente nella directory, ma da un punto di vista fisico risiede ancora sul disco. Nel momento in cui termina l'esecuzione del programma, e di conseguenza viene chiuso il file, si libera lo spazio dalla memoria.

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

int main(void)
{
      if (open("tempfile", O_RDWR) < 0)
            err_sys("open error");

      if (unlink("tempfile") < 0)
            err_sys("unlink error");

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

      exit(0);
}
```

2.0.7 remove

```
#include <stdio.h>
int remove(const char *pathname);
```

Questa system call rimuove il file specificato da *pathname*. Restituisce 0 se OK, -1 in caso di errore. È equivalente alla **unlink**.

2.0.8 rename

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Questa system call assegna un nuovo nome *newname* ad un file od ad una directory data come primo argomento (*oldname*). Restituisce 0 se OK, -1 in caso di errore.

2.1 LINK SIMBOLICI

Un link simbolico (soft link) sono dei file che possono essere visti come un puntatore indiretto ad un file, oppure quelli che possiamo vedere come dei collegamenti. Quando si fa un collegamento sul desktop ad un programma, di fatto viene creato un nuovo file che ci consentirà in maniera veloce di puntare ad un file che si trova in una directory particolare.

Creare un soft link vuol dire creare un nuovo file, creare un nuovo file significa avere un nuovo i-node che punta ad un data block con un contenuto. Creare un hard link, vuol dire creare una nuova directory entry che punta ad un i-node già esistente.

Quando si usano funzioni che si riferiscono a file (open, read, stat, etc.), si deve sapere se seguono il link simbolico oppure no.

In questa tabella sono riportare varie funzioni, e per ognuna di essa, se segue il link simbolico oppure no. Tutte seguono il link simbolico ad esclusione di quelle che cominciano con la lettera **l** e di altre visibili nella tabella. Ad esempio la **lstat**. Se do in pasto alla stat un link simbolico, la stat mi darà le informazioni del file puntato. Ma se volessi le informazioni del link simbolico, cosa faccio? Con la stat non potrò mai ottenerlo e per questo uso la **lstat**.
Remove e rename lavorano sul link simbolico e non sul file puntato altrimenti non ci sarebbe modo di lavorare sui link simbolici.
La readlink legge il contenuto del link simbolico, è l'equivalente di una open, read e close in un colpo solo.

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

2.1.0 symlink

```
#include <unistd.h>
int symlink(const char *path, const char *sympath);
```

Questa system call crea un link simbolico *sympath* che punta a *path*. Restituisce 0 se OK, -1 altrimenti.
I link simbolici possono essere creati anche su file che non esistono.

2.1.1 readlink

```
#include <unistd.h>
int readlink(const char *pathname, char *buf, int bufsize);
```

Questa system call legge il contenuto del link simbolico del primo argomento e lo copia in *buf*, la cui taglia è *bufsize*. Restituisce il numero di byte letti se OK, -1 in caso di errore. Legge il contenuto del link e non del file cui esso si riferisce. Se la lunghezza del link simbolico è > *bufsize* viene restituito errore.

2.1.2 mkdir

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

Questa system call crea una directory i cui permessi di accesso vengono determinati da mode e dalla mode creation mask del processo. Restituisce 0 se OK, -1 in caso di errore. La directory creata avrà come:

- owner ID = l'effective ID del processo.
- group ID = group ID della directory padre.

La directory sarà vuota ad eccezione di . e .. .

2.1.3 rmdir

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int rmdir(const char *pathname);
```

Con questa system call viene decrementato il numero di link al suo i-node. Se esso diventa uguale a 0 allora si libera la memoria solo se nessun processo ha quella directory aperta. Restituisce 0 se OK, -1 in caso di errore.
Sappiamo che dal punto di vista del sistema operativo, le directory sono dei file speciali e per questo esiste la possibilità di leggerne il contenuto. Per leggere il contenuto di una directory, esistono delle system call ad hoc:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```

Con **opendir** si effettua l'operazione equivalente a quella della **open** su un file, però viene effettuata su una directory. Restituisce il puntatore se OK, NULL in caso di errore.
Con **readdir** verrà restituita una struct dirent, OK se tutto va bene, NULL in caso di errore.
Con **rewinddir** si riparte da 0, con **closedir** si chiude la directory.
Per scandire una directory si effettua un while.

STRUCT DIRENT:

```
struct dirent {
        ino_t   d_ino;      /* i-node number */
        ......
        char    d_name[256]; /* Null-terminated filename */
    };
```

La **struct dirent** è fatta almeno da questi due parametri: un campo *d_ino* che rappresenta l'i-node number, e un'array di caratteri che contiene il nome. Se volessi leggere tutti i file contenuti in una directory devo effettuare una opendir della directory, fare un ciclo while readdir diverso da NULL e visualizzare il campo *d_name*.

2.1.4 chdir, fchdir, getcwd

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
char *getcwd(char *buf, size_t size);
```

Le prime 2 system call cambiano la cwd del processo chiamante a quella specificata come argomento. Restituiscono 0 se OK, -1 in caso di errore.
La terza system call ottiene in *buf* il path assoluto della cwd. Restituisce *buf* se OK, NULL in caso di errore.