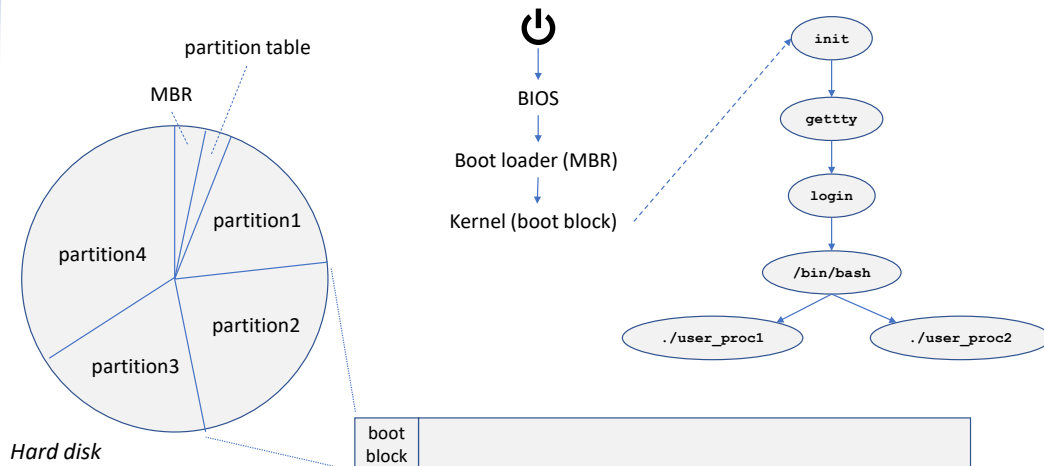


Gestione del file system

Ing. Giovanni Nardini - University of Pisa - All rights reserved

Boot del sistema

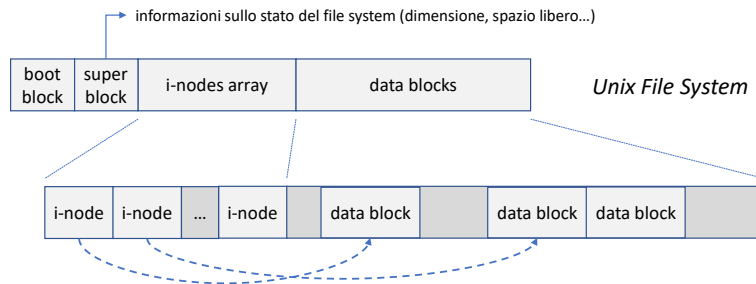


All'accensione del sistema, il **BIOS** (Basic Input/Output System) effettua dei controlli sull'integrità dei dischi e delle periferiche, carica in memoria principale il **boot loader** presente nel **Master Boot Record** e lo manda in esecuzione. Il MBR si trova nel primo settore dell'hard disk.

Il **boot loader** (che è tipicamente GRUB nei moderni sistemi Linux) accede alla tabella delle partizioni per individuare quali partizioni contengono un kernel. Ne sceglie uno (o lo fa scegliere all'utente tramite una schermata) e lo manda in esecuzione.

In particolare, manda in esecuzione il codice contenuto nel **boot block** della partizione scelta, la quale "monta" il file system e infine esegue il processo **init**, dal quale discendono tutti i processi del sistema (infatti ha il PID 1).

Struttura del file system



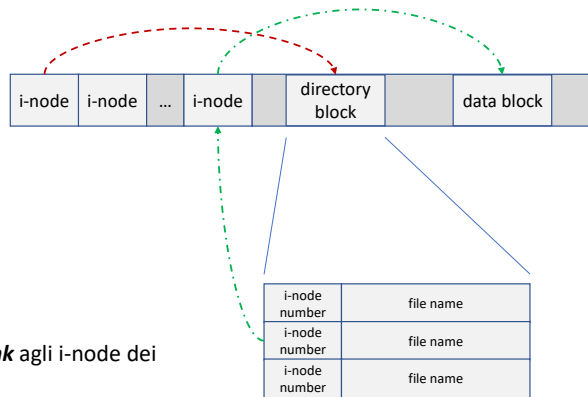
Un i-node contiene i metadati di un certo file e dei puntatori ai data block che contengono i dati effettivi del file.

Un data block ha una dimensione prefissata (tipicamente 4096 byte), ma un file può legittimamente essere più grande di tale dimensione. Dove viene immagazzinato?

L'i-node contiene un certo numero di puntatori a diversi data block, ciascuno dei quali può contenere a sua volta altri puntatori ad altri data block, realizzando fino a tre livelli di indirezione.

Struttura del file system

Anche le directory sono file!



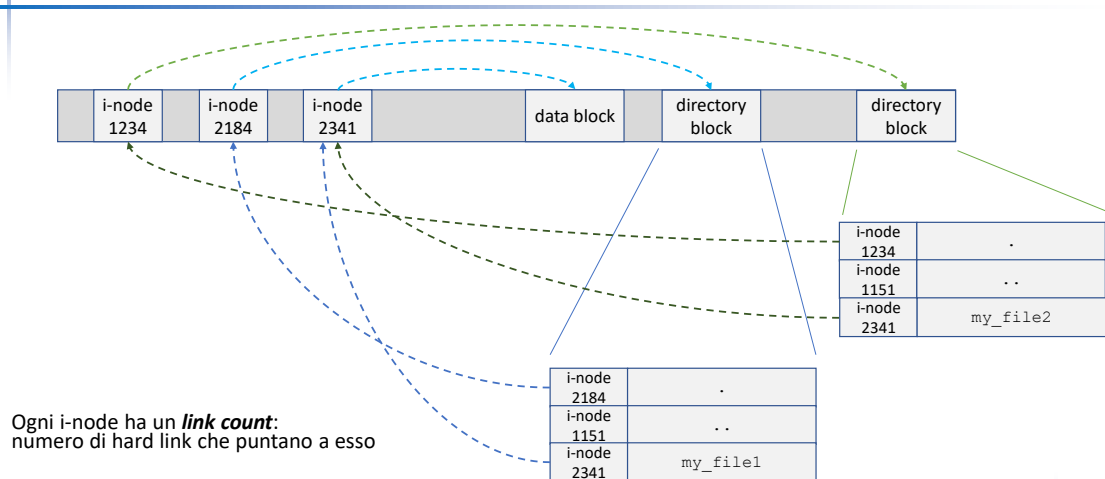
Il data block di una directory contiene **hard link** agli i-node dei file contenuti nella directory

- A sua volta, tali i-node possono puntare a delle directory

Una directory non è altro che un file contenente la lista dei file in essa contenuti. In particolare, ogni linea contiene il nome del file e il numero di i-node di quel file. Conoscendo il numero dell'i-node è possibile individuare la posizione dell'i-node stesso nell'array degli i-node, e da lì arrivare al data block contenente il file stesso.

In altre parole, il nome del file è solamente una specie di puntatore a un i-node (e quindi al file). E' possibile usare due nomi per indicare lo stesso data block.

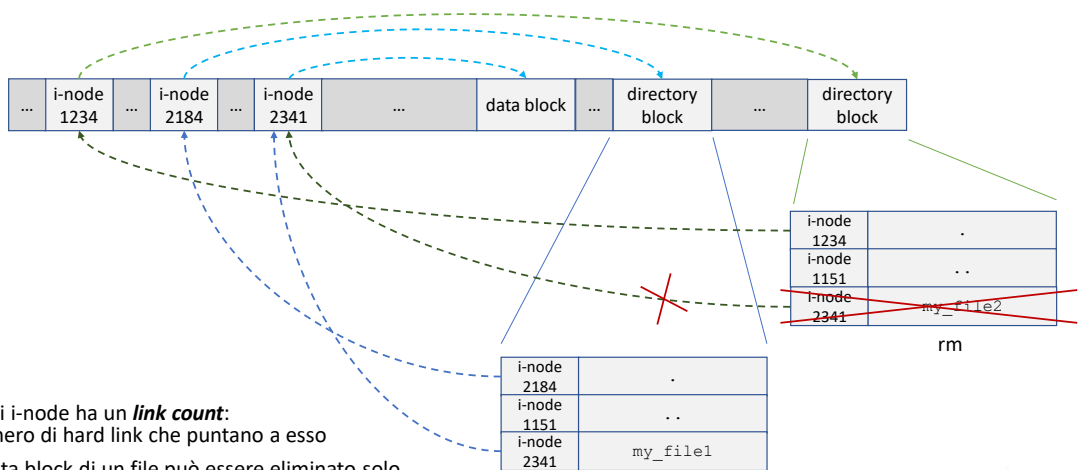
Hard links



In un directory block, troviamo sempre almeno due entry:

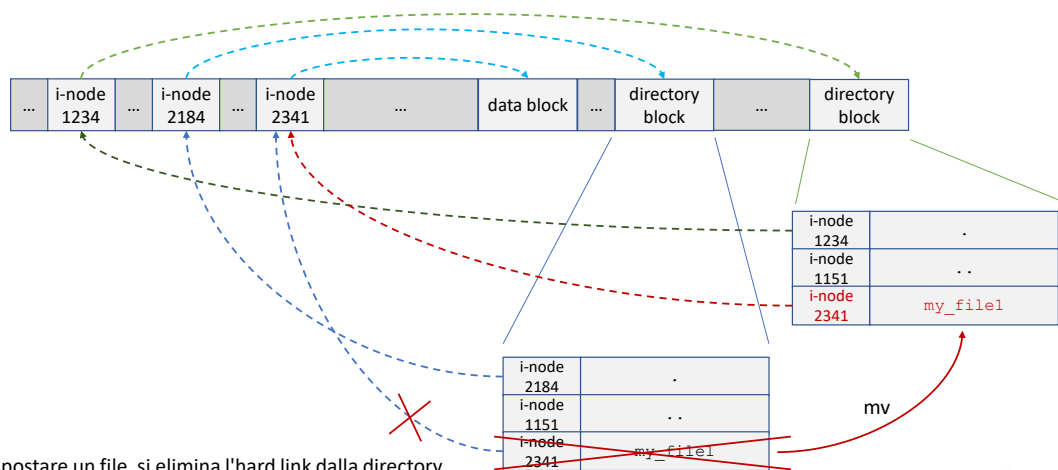
- . corrisponde all'i-node della directory corrente (cosa succede se digitiamo il comando "cd ." ?)
- .. corrisponde all'i-node della directory genitore (cosa succede se digitiamo il comando "cd .." ?)

Hard links



Digitando il comando "rm", in realtà, non stiamo eliminando veramente il file, che potrebbe essere raggiunto da un'altra directory e da un altro hard link. Pertanto, "rm" elimina solamente l'hard link indicato, ed elimina il file soltanto se quello era l'ultimo hard link che puntava all'i-node del file.

Hard links



Spostare un file da una directory a un'altra implica solamente spostare il suo nome da un directory block all'altro (il file vero e proprio non si muove).

Creare un hard link

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

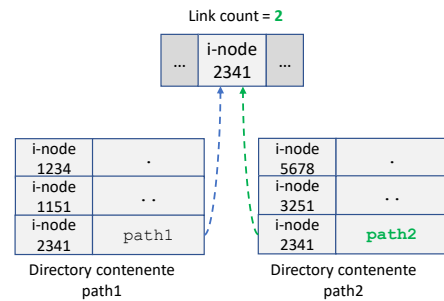
```
int link(const char* path1, const char* path2);
```

File a cui il link si riferisce

Nuova entry

La funzione restituisce:

- 0 se ok
- -1 se errore



- Crea una nuova entry nel percorso specificato da *path2*, con hard link verso l'i-node a cui *path1* si riferisce
- Incrementa il link count dell'i-node

Dopo l'esecuzione della system call `link()`, sia `path1` che `path2` puntano allo stesso i-node e quindi allo stesso file

Rimuovere un hard link

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

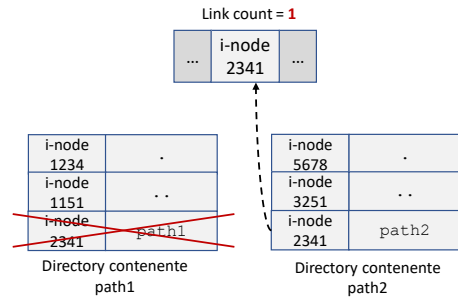
```
int unlink(const char* path1);
```

Entry da rimuovere

La funzione restituisce:

- 0 se ok
- -1 se errore

- Elimina la entry specificata e l'hard link corrispondente
- Decrementa il link count dell'i-node
 - Se il link count diventa 0 → il kernel libera il data block corrispondente



Metadati di un file

- Informazioni contenute dentro all'i-node:
 - Permessi
 - Utente e gruppo proprietario
 - Dimensione del file
 - Ora di ultima modifica
 - ...

```
studente@debian-SdE:~$ ls -l hello.c  
-rw-r--r-- 1 studente studente 102 Nov  3 11:31 hello.c
```

Il programma 'ls' deve dunque recuperare le informazioni contenute nell'i-node

Comando stat

- Il comando `stat` della shell restituisce tutte le informazioni contenute nell'i-node
- Il formato dell'output può cambiare a seconda delle versioni di Unix

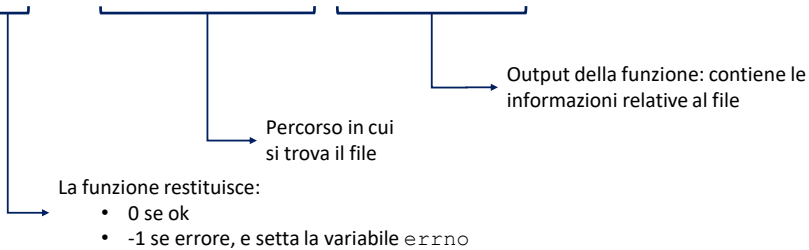
```
studente@debian-SdE:~$ stat hello.c
  File: hello.c
  Size: 102          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 474          Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/studente)   Gid: ( 1000/studente)
Access: 2022-11-03 11:40:03.746693870 +0100
Modify: 2022-11-03 11:40:10.510073838 +0100
Change: 2022-11-03 11:40:10.510073838 +0100
 Birth: 2022-11-03 11:38:49.601640178 +0100
```

Il programma 'stat' deve recuperare tutte le informazioni contenute nell'i-node

stat

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

```
int stat(const char* path, struct stat* sb);
```



- Poiché i metadati risiedono negli *i-node*, non ho bisogno di accedere al file



man 2 stat

Talvolta è utile ottenere i metadati di un file direttamente da un programma. La funzione `stat` (e le sue varianti) è una system call.

stat

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

```
int stat(const char* path, struct stat* sb);
```

Esempio:

```
...
struct stat sb;
const char* path = "path/to/file";

if (stat(path, &sb) < 0)
{
    fprintf(stderr, "Errore! %s", strerror(errno));
    exit(EXIT_FAILURE);
}
...
```

Per usare la system call è dunque necessario dichiarare prima una variabile di tipo **struct stat**, che verrà passata come argomento alla funzione stat (si passa il suo indirizzo, notare &)

fstat

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

```
int fstat(int fd, struct stat* sb);
```

Esempio:

```
...
struct stat sb;
int fd;

fd = open("path/to/file", O_RDONLY);

if (fstat(fd, &sb) < 0)
{
    fprintf(stderr, "Errore! %s", strerror(errno));
    exit(EXIT_FAILURE);
}
...
```

Differisce dalla stat per il fatto che prende come argomento un file descriptor già aperto in precedenza.

struct stat

```
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 (file system) */  
    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 */  
    struct timespec st_atim; /* time of last access */  
    struct timespec st_mtim; /* time of last modification */  
    struct timespec st_ctim; /* time of last file status change */  
    blksize_t    st_blksize; /* best I/O block size */  
    blkcnt_t     st_blocks; /* number of disk blocks allocated */  
};
```

```
studente@debian-SdE:~$ ls -l hello.c  
-rw-r--r-- 1 studente studente 102 Nov  3 11:31 hello.c
```

Questa slide mostra il contenuto del tipo struct stat: molti dei suoi campi vengono utilizzati dal comando ls -l


struct stat – tipo di file

```
mode_t      st_mode;    /* file type & mode (permissions) */
```

- Per ricavare il tipo di file, si usano delle macro definite in `<sys/stat.h>`

```
...
if (stat(path, &sb) < 0) {
    /* errore */
}

if (S_ISREG(sb.st_mode) > 0) {
    /* this is a regular file */
}
else if (S_ISDIR(sb.st_mode) > 0) {
    /* this is a directory */
}
...
```



Macro	Restituisce 1 se...
<code>S_ISREG()</code>	Regular file
<code>S_ISDIR()</code>	Directory
<code>S_ISLNK()</code>	Symbolic link
<code>S_ISCHR()</code>	File a caratteri
<code>S_ISBLK()</code>	File a blocchi
<code>S_ISFIFO()</code>	FIFO (named pipe)
<code>S_ISSOCK()</code>	Socket

struct stat – permessi


```
mode_t      st_mode;    /* file type & mode (permissions) */
```

- Per ricavare i permessi, si testa `st_mode` con i flag dei permessi
 - Gli stessi usati nella `open()` con l'opzione `O_CREAT` (creazione di un file)

```
...
if (stat(path, &sb) < 0) {
    /* errore */
}

if (sb.st_mode & S_IRUSR) > 0) {
    /* owner has read permission */
}

if (sb.st_mode & (S_IRUSR | S_IWUSR) > 0) {
    /* owner has read and write permission */
}
...
```



S_IRUSR	Lettura proprietario
S_IWUSR	Scrittura proprietario
S_IXUSR	Esecuzione proprietario
S_IRGRP	Lettura gruppo
S_IWGRP	Scrittura gruppo
S_IXGRP	Esecuzione gruppo
S_IROTH	Lettura altri utenti
S_IWOTH	Scrittura altri utenti
S_IXOTH	Esecuzione altri utenti

Cambiare i permessi di un file

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

```
int chmod(const char* path, mode_t mode);
```

Esempio:

```
...  
const char* path = "path/to/file";  
mode_t mode = S_IRUSR | S_IRGRP | S_IROTH;  
  
if (chmod(path, mode) < 0) {  
    fprintf(stderr, "Errore! %s", strerror(errno));  
    exit(EXIT_FAILURE);  
}  
...
```

Nota il riferimento

Creare/eliminare una directory

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

int mkdir(const char* path, mode_t mode);    int rmdir(const char* path);

...
const char* path = "newdir";
mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;

if (mkdir(path, mode) < 0) {
    ...
}
...

if (rmdir(path) < 0) {
    ...
}
```



man 2 mkdir



man 2 rmdir

mkdir crea una directory inizialmente vuota. Notare che assegniamo il permesso di esecuzione sulla nuova directory, necessario per poter navigare all'interno della directory stessa.

rmdir rimuove la directory solo se è vuota e se non ci sono altri processi che la stanno utilizzando (ovvero che hanno un file descriptor aperto)

Cambiare working directory

```
#include <unistd.h>                                     #include <unistd.h>
char* getcwd(char* buff, size_t size);                   int chdir(const char* path);

...
const char* path = "newdir";
mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
mkdir(path, mode);

char* buff;
buff = getcwd(NULL, 0);  // salva la directory corrente

chdir(path);             // entra nella nuova directory
...
chdir(buff);             // torna alla directory precedente
```



man 2 chdir

getcwd serve per ottenere il path assoluto della directory corrente del processo. Se gli viene passato NULL come primo argomento, alloca in memoria dinamica il numero di byte necessario a contenere la stringa risultante, il cui puntatore viene restituito come risultato.

chdir cambia la directory corrente del processo in esecuzione. Attenzione: la working directory è un attributo del processo in esecuzione, per cui chdir non cambia la directory della shell da cui si è lanciato il programma.

Leggere il contenuto di una directory

```
#include <dirent.h>

DIR* opendir(const char* path);

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

```
struct dirent {
    ino_t d_ino;    ← i-node number
    char d_name[]; ← Nome del file
                  (path relativo)
}
```



- Per leggere le informazioni sulla directory, questa deve essere "aperta" tramite `opendir()`
 - Concetto simile alla `open()` per I/O sui file
- La struttura `DIR` è una sequenza delle entry di una directory
 - Verrà usata dalle funzioni che manipolano la directory
- `readdir()` restituisce la prossima entry della directory
- Le funzioni restituiscono un puntatore `NULL` in caso di errore

E' possibile vedere le singole entry di una directory accedendole **sequenzialmente**.
L'ordine con cui le entry della directory vengono scansionate non è determinabile a priori

Leggere una directory

```
...
DIR *dp;
struct dirent *dirp;

// apertura della directory
if ( (dp = opendir("dir")) == NULL ) {
    // errore
}

// scorre il contenuto della directory
while ( (dirp = readdir(dp)) != NULL )
{
    printf("i-node number: %d, file name: %s \n", dirp->d_ino, dirp->d_name);
}
...
```

Leggere una directory

```
#include <dirent.h>
```

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

Resetta l'offset

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

Chiude la directory.

Analoga alla `close()` per le operazioni di I/O su file