



---

# Struttura interna del sistema operativo Linux

## Il file system

.revisione del 30 marzo 2004

---



# File System

---

- Il **file system** è il componente del sistema operativo che realizza i **servizi di gestione dei file**
  - **file**: unità di archiviazione in memoria di massa.
- Il file system rappresenta quindi la **struttura logica della memoria di massa** (**organizzazione e memorizzazione dei file**) ed è costituito da un insieme di funzioni di sistema operativo e dalle relative strutture dati di supporto.
- Le astrazioni fornite dal file system sono:
  - File, link e operazioni sui file
  - Directory e file speciali
  - Attributi e politiche di accesso
  - Mapping del file system logico sui dispositivi fisici



## Bibliografia

---

- Informazioni accurate e aggiornate fino al kernel 2.4 sono disponibili nel seguente testo:

Daniel P. Bovet, Marco Cesati,  
**Understanding the Linux Kernel,**  
O'Reilly

- capitoli 12 (VFS) e 17 (ext2fs)
- si può leggere online al seguente indirizzo  
<http://www.oreilly.com/catalog/linuxkernel/>



# File

---

- Il concetto di *file* offre una visione logica omogenea delle informazioni memorizzate.
  - La visione non dipende dal tipo di dispositivo fisico su cui le informazioni vengono memorizzate
- Un *file* è costituito da:
  - una sequenza di byte, che rappresenta l'insieme di informazioni omogenee;
  - un nome simbolico;
  - un insieme di attributi.
- Nei sistemi operativi unix-like, un *file* può
  - contenere dati o programmi (file regolare)
  - contenere riferimenti ad altri file (directory)
  - rappresentare un dispositivo (file speciale)



# File

---

- Nome:
    - nome simbolico, univoco nella sua directory, con cui ci si riferisce ad esso
  
  - Attributi:
    - *Tipo*: definisce il tipo dei dati contenuti (estensione del nome, opzionale)
    - *Locazione (implicito)*: è un riferimento alla posizione fisica sul dispositivo
    - *Dimensione*: dimensione dei dati espressa in byte o blocchi
    - *Protezione*: definisce le politiche di gestione degli accessi
    - *Ora e Data*: indicano il momento della creazione, ultima modifica o ultimo accesso
    - *Proprietario*: indica l'utente e il gruppo che possiedono il file
-



## Funzioni per la gestione dei file

---

- Sui file possono essere compiute diverse **operazioni**, che vengono svolte attraverso richieste di **servizi al sistema operativo**, invocabili da programma:
  - creazione;
  - apertura;
  - scrittura;
  - lettura;
  - riposizionamento;
  - cancellazione;
  - ...



## Funzioni per la gestione dei file

---

- Per operare su un file è necessario:
  - **aprirlo**, se il file esiste già
  - **crearlo** (cioè inserirlo nel file system), se il file non esiste
  
- Le funzioni di apertura e creazione richiedono come parametro il **nome** del file e restituiscono un *intero non negativo* che identifica il **descrittore del file**
  
- Le funzioni di lettura e scrittura, e altre ancora, usano come riferimento il descrittore del file e operano su sequenze di byte a partire da un byte specifico, la cui posizione è contenuta dall'indicatore di **posizione corrente**



# Funzioni per la gestione dei file

---

- L'immagine del file system è memorizzata in memoria di massa (vedremo più avanti come)
- Per rendere più efficienti gli accessi a file, il sistema operativo mantiene in memoria centrale una parte delle **strutture dati** per gestire i file. Al momento interessa mettere in evidenza:
  - **Tabella dei file aperti per ogni processo**
    - è un vettore che contiene alcune informazioni sui file aperti da un certo processo. Il **descrittore del file** agisce da **indice del vettore**;
    - è contenuta nel process descriptor
  - **Tabella globale dei file aperti nel sistema**
    - in realtà in Linux non esiste così come la descriviamo, esiste la *dcache* (directory entry cache);
    - è un vettore che contiene un elemento per ogni file aperto nel sistema. Ogni elemento viene referenziato tramite gli elementi delle tabelle dei file aperti dei processi e contiene, tra le altre cose, il **contatore di uso** del file, la **posizione corrente** e un **riferimento** al file su disco



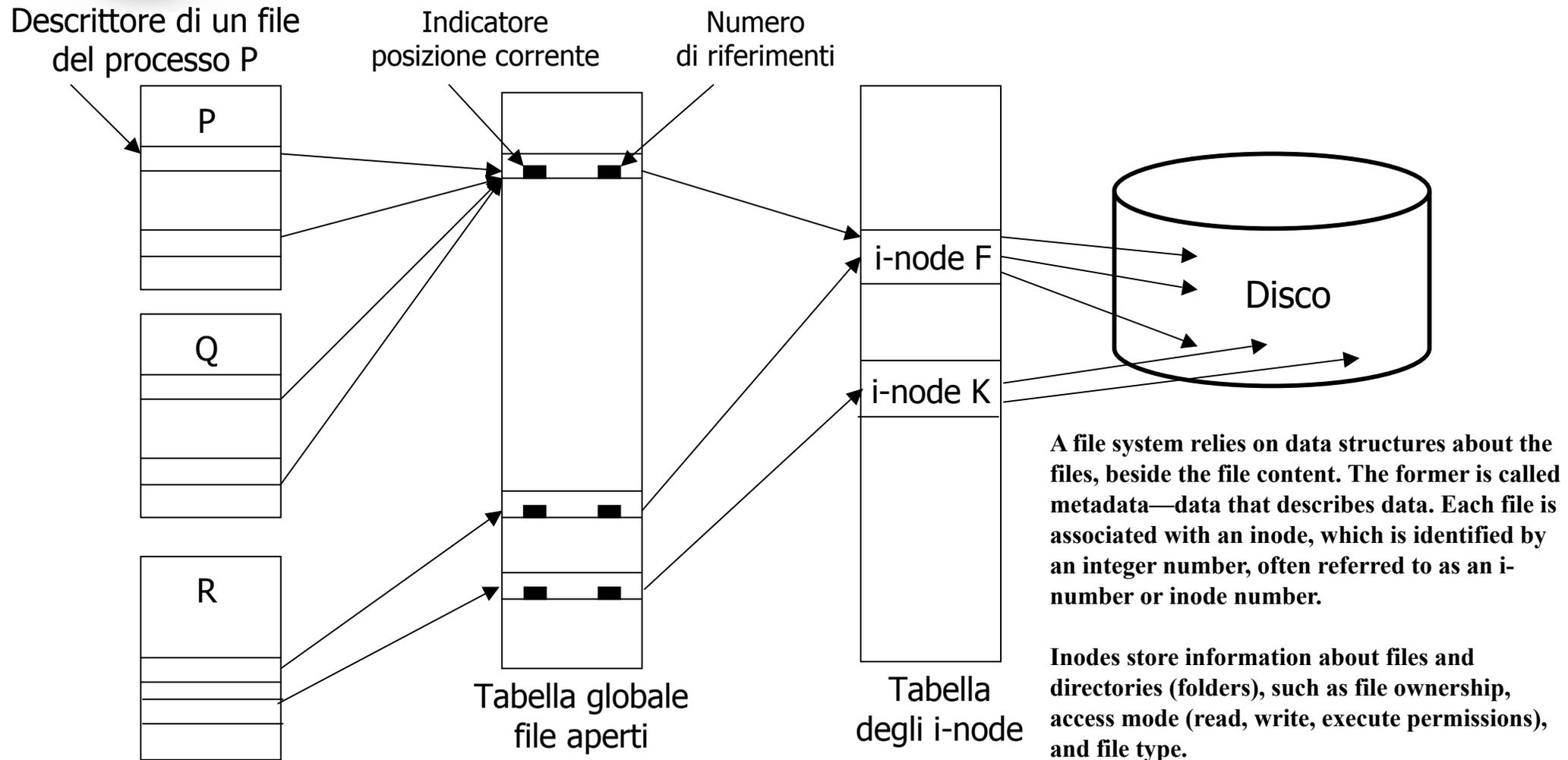
## Linguaggio C, file e sistema operativo

---

- Il linguaggio C dispone di una libreria standard di ingresso e uscita (<stdio.h>), che definisce le **funzioni C** per le operazioni sui file (fopen, fread, fwrite, fclose, fputs, fgets etc.);
- Il concetto di file fornito dal linguaggio C si chiama *stream*; il tipo di dato associato è un puntatore ad una struttura interna della C runtime library:  
**FILE \* file\_pointer**
- Le funzioni C operano su stream e invocano internamente i servizi messi a disposizione dal sistema operativo;
- Le implementazioni delle funzioni C su un certo sistema operativo definiscono le modalità di chiamata dei servizi di sistema e il **legame** tra **stream** e **descrittori di file**;
- Attenzione: fopen e simili potrebbero avere comportamenti differenti da un sistema operativo ad un altro (ad es. Unix System V e Linux).



# Il File System





## Creazione - `creat()`

---

Permette di aggiungere un nuovo file al **file system**

□ Le operazioni richieste sono:

- allocare lo spazio su disco;
- creare il nuovo descrittore del file;
- aggiungere il descrittore al file system
- modifiche conseguenti nella tabella dei file aperti dal processo e nella tabella globale dei file aperti

□ Uso:

```
int creat (char * name, int mode)
```

- mode: indica i permessi da impostare (maschera di costanti `S_lxxx`).



## Apertura - `open()`

---

- Sulla base del nome individua la posizione del file sul disco.
- Copia il descrittore del file nella tabella globale dei file aperti.
- Aggiorna la prima voce libera nella tabella dei file aperti dal processo e restituisce il suo indice (il descrittore)
- Uso:

`int open(char * name, int flags, int mode)`

- flags indica il tipo di accesso richiesto:
    - `O_RDONLY`: il file viene aperto in sola lettura
    - `O_WRONLY`: il file viene aperto in sola scrittura
    - `O_RDWR`: il file viene aperto in lettura e scrittura
    - `O_CREAT`: il file viene creato se non esiste
    - ...
  - mode indica i permessi da impostare nel caso in cui il file non esista e debba essere creato (maschera di costanti `S_lxxx`)
-



## Scrittura - `write()`

---

- ❑ Aggiunge dati ad un file già creato.
- ❑ Per scrivere dati su un file è necessario indicare:
  - il descrittore del file (ottenuto tramite `open` o `creat`);
  - i dati da scrivere.
- ❑ Uso:

```
int write(int fd, const void * buffer, int count)
```

  - restituisce il numero di caratteri effettivamente scritti.
- ❑ Il file system aggiorna il puntatore alla posizione in cui deve essere effettuata la scrittura successiva.



## Lettura - `read()`

---

- ❑ Preleva dati da un file già creato.
- ❑ Per leggere dati da un file è necessario fornire:
  - il descrittore del file (ottenuto tramite `open`);
  - un puntatore al buffer destinato a contenere i dati
- ❑ Uso:
  - `int read(int fd, void * buffer, int count)`
  - restituisce il numero di caratteri effettivamente letti.
- ❑ Il file system aggiorna il puntatore alla posizione in cui deve essere effettuata la lettura successiva.



## Riposizionamento - `lseek` ()

---

- Sposta il puntatore di lettura/scrittura
- Le operazioni consentite dipendono dall'accesso al file
- Uso:

`long lseek(int fd, long offset, int whence)`

- modifica la posizione corrente e la restituisce come long
- *offset* indica lo scostamento rispetto ad una posizione notevole, che si sceglie con il valori dell'argomento *whence*:
  - `SEEK_SET` = 0: inizio del file;
  - `SEEK_CUR` = 1: posizione corrente;
  - `SEEK_END` = 2: fine del file;

- Esempi:

- `lseek(fd, 0L, SEEK_CUR)` : restituisce la posizione attuale senza modificarla;
- `lseek(fd, 0L, SEEK_SET)` : posiziona all'inizio del file;
- `lseek(fd, 0L, SEEK_END)` : posiziona alla fine del file.



## Duplicazione di descrittore - `dup()`

---

- ❑ Duplica un descrittore, cioè crea un nuovo descrittore, associato ad un file già aperto;
- ❑ Aggiunge una riga nella tabella dei file aperti da quel processo;
- ❑ Usa il primo descrittore disponibile;
- ❑ Uso:
  - `fd1 = open(...);`
  - `fd2 = dup(fd1);`
  - la **posizione corrente** è **unica** per i due o più descrittori (**esiste una sola riga nella tabella globale dei file aperti**).



## Duplicazione di descrittore via `fork()`

---

- ❑ La chiamata `fork()` per la creazione di processi condividono automaticamente tutti i file aperti;
- ❑ il figlio eredita dal padre i descrittori aperti, che restano condivisi fra padre e figlio;
- ❑ i descrittori condivisi hanno ovviamente una sola **posizione corrente**.



## Chiusura - `close()`

---

- ❑ Libera il descrittore nella tabella dei file aperti dal processo; quel descrittore può essere riusato;
- ❑ Decrementa il contatore di uso del descrittore nella tabella globale dei file aperti corrispondente;
- ❑ Può eliminare il descrittore dalla tabella globale dei file aperti se il contatore vale zero e c'è penuria di spazio;
- ❑ **Uso:**  
`int close(int fd)`
  - restituisce l'esito dell'operazione di chiusura.



## Nuovo riferimento - `link()`

---

- Collega un nuovo nome ad un file già esistente:
  - `int link(char *nome, char *nuovo_nome)`
  - restituisce l'esito dell'operazione.
- Crea una nuova voce di directory per *nuovo\_nome*
- Incrementa di 1 il contatore dei riferimenti (`link`) all'i-node associato a quel file



## Cancellazione - `unlink()`

---

- ❑ Scollega un nome di file dal relativo i-node e cancella tale nome;
- ❑ Il contatore dei riferimenti all'i-node viene decrementato;
- ❑ Se tale contatore giunge a zero, anche l'i-node e il file vengono **eliminati dal filesystem**:
  - lo spazio occupato dal file sul dispositivo fisico viene disallocato;
  - esso viene aggiunto alla lista dello spazio disponibile sul dispositivo;
- ❑ Uso:
  - `int unlink(char *nome)`



## Un esempio (i)

---

```
/* esempio di uso delle operazioni Linux sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
void main()
{
    int fd1;
    int i, pos;
    char c;
    fd1 = open("fileprova", O_RDWR); /* apertura del file */
    printf("pos=%ld\n", lseek(fd1,0,1)); /* visualizza posizione corrente */

    for (i=0; i<20;i++) /* scrittura del file */
    {
        c = i + 65; /*caratteri ASCII a partire da 'A' */
        write(fd1, &c, 1);
        printf("%c",c);
    }
    /* visualizza posizione corrente; riportala a 0 */
    printf("\npos=%ld \n", lseek(fd1,0,1));
    lseek(fd1,0,0);
}
```



## Un esempio (ii)

---

```
/* lettura e visualizzazione del file */
for (i=0; i<20;i++) {
    read(fd1, &c, 1);
    printf("%c",c);
}
printf("\n");

/* posizionamento al byte 5 del file e stampa posiz.*/
printf("\npos= %ld\n", lseek(fd1,5,0));

/*lettura di 5 caratteri */
for (i=0; i<5;i++) {
    read(fd1, &c, 1);
    printf("%c",c);
}
printf("\n");

/*scrittura di 5 caratteri x*/
c= 'x';
for (i=0; i<5;i++)
    write(fd1, &c, 1);
```

---



## Un esempio (iii)

---

```
/* lettura del file dall'inizio */
lseek(fd1,0,0);
for (i=0; i<20;i++)
{
    read(fd1, &c, 1);
    printf("%c",c);
}
printf("\n");
/* chiusura file */
close(fd1);
}
```



## Directory: struttura gerarchica del file system

---

- Un file system può risiedere su una o più **partizioni** (porzione contigua di dispositivo fisico scelta da chi ha configurato il sistema);
  - **Directory**: Le directory contengono informazioni sui file e sulle directory in esse incluse, e fungono da indice per il proprio contenuto;
  - **File (regolari)**: contengono dati o programmi;
  
- **Pathname** (nome completo di un file): è il nome (percorso) ottenuto concatenando tutti i nomi delle directory che devono essere percorse dalla radice fino al file, quindi il nome del file
  - esempio: /home/jsmith/work/report2004.tex



# Directory

---

- ❑ Le directory sono un gruppo di nomi di file;
- ❑ le **directory** sono esse stesse dei file, quindi sono caratterizzate da nome e diritti d'accesso;
- ❑ contengono informazioni sui file (e quindi anche delle directory) in esse contenute (nome, attributi, proprietario);
- ❑ forniscono una corrispondenza tra nomi di file e contenuto;
- ❑ il loro contenuto è una **tabella**, con una voce per ogni file
- ❑ ogni voce è del tipo:

*<nome file, numero di i-node>*



## Contenuto di una directory (/etc)

---

riferimento (*i-node*)

nome del file

3	<i>.</i> ( <i>direttorio corrente</i> )
2	<i>..</i> ( <i>direttorio padre</i> )
4	passwd
7	shadow
...	...
...	...



## Operazioni sulle directory

---

- Sulle directory, così come sui file, è possibile compiere operazioni:
  - **Ricerca di un file**: Sulla base di un nome o una espressione regolare consente di recuperare le informazioni su uno o più file;
  - **Creazione di un file**: aggiunta alla directory di un elemento che contiene le informazioni sul nuovo file;
  - **Rimozione di un file**: eliminazione di un nome collegato a file, ed eventualmente del file stesso.
  - **Elenco dei file**: produce l'elenco dei nomi ed eventualmente altre informazioni relative ai file memorizzati
  - **Rinomina di un file**: modifica il nome di un file già presente nella directory.
- Per creare una directory si usa la funzione `mkdir()`



## Possibili organizzazioni di directory

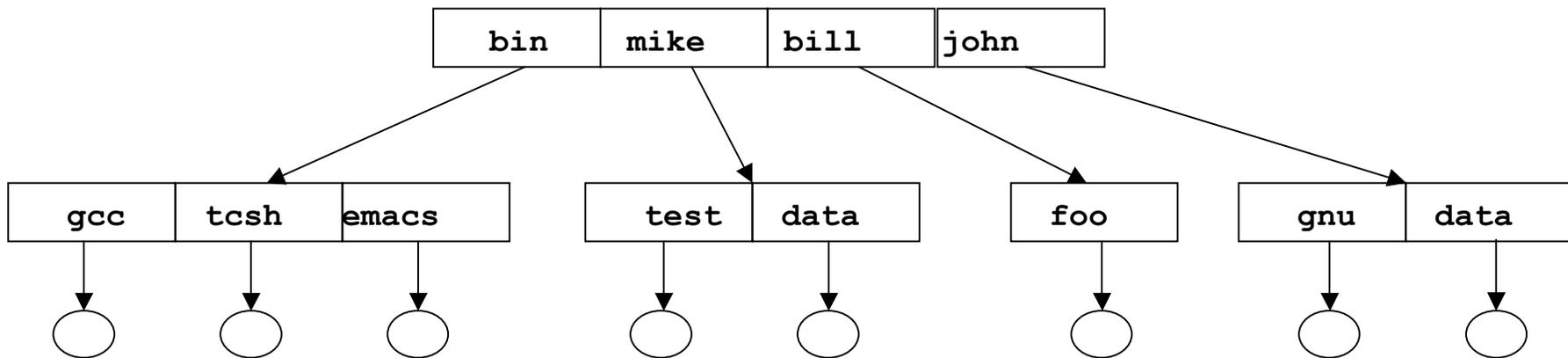
---

- Una sola directory
  - Tutti i file sono contenuti in essa
  - Struttura molto semplice ma con alcuni problemi (unicità dei nomi, ricerca inefficiente, accesso indifferenziato per tutti gli utenti);
- Due livelli di directory
  - **directory principale**: (*root*) contiene un elenco di directory, una per ogni utente
  - **directory utente**: contiene i file di un singolo utente (*home*).
  - I singoli utenti potrebbero vedere e gestire solo i propria file, posti nella propria home
  - La gestione della directory root è affidata ad un amministratore.



## Organizzazione a due livelli

- Quando un utente richiede l'accesso ad un file, il file system cerca il nome nella directory home dell'utente.
- Per accedere ai file di altri utenti si utilizza il *pathname* cioè la composizione del nome dell'utente e del nome del file.
- Spesso gli applicativi sono accessibili a tutti gli utenti e memorizzati in una directory specifica (*bin*).
- I file vengono prima cercati prima nella directory home dell'utente, quindi, se la ricerca fallisce, nella directory degli applicativi





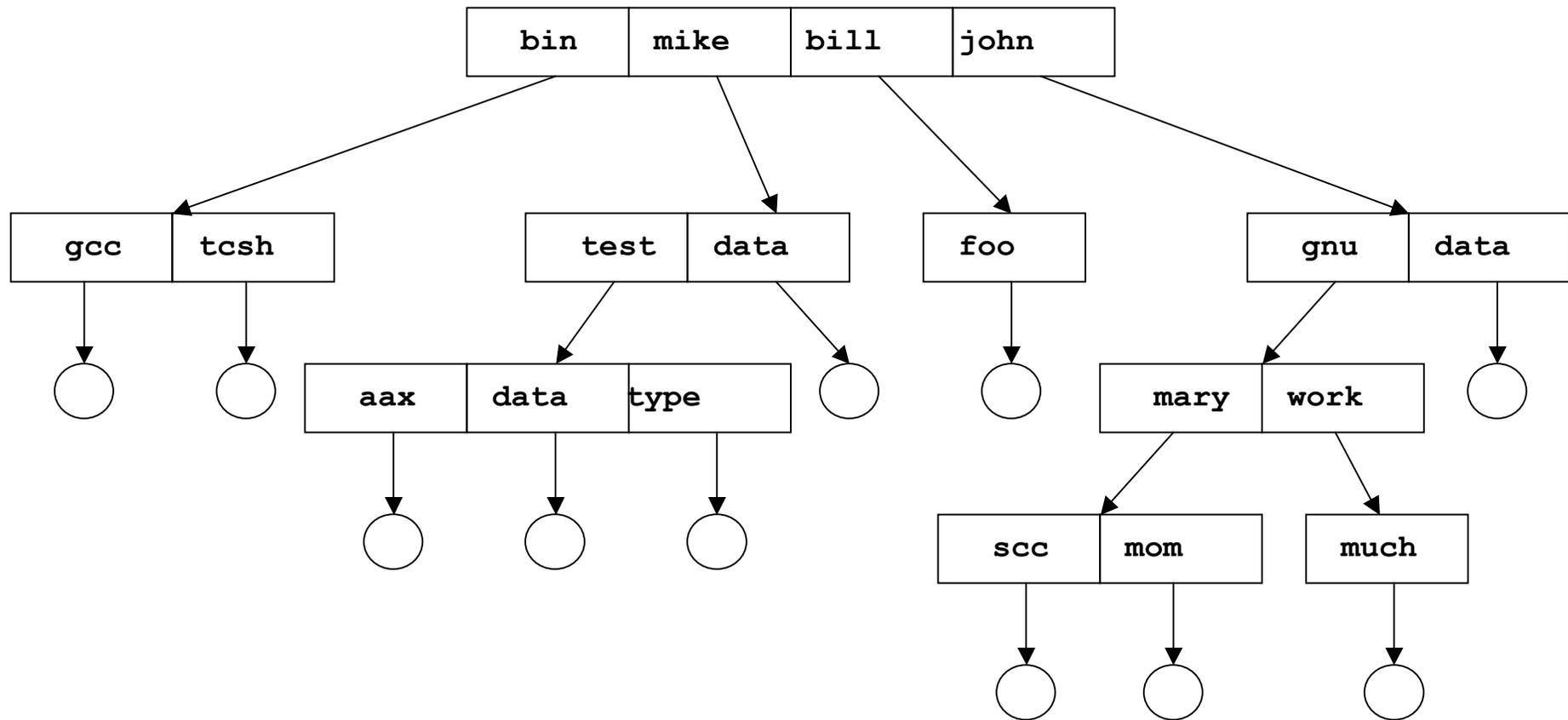
## Struttura di directory ad albero

---

- E' una estensione del caso precedente, con un numero arbitrario di livelli
- Un utente accede ai file attraverso il loro *pathname*
- Per semplificare l'accesso ai file viene definito il concetto di *current working directory* (*cwd*) cioè di directory corrente;
- Si intende che i file indicati senza percorso devono essere cercati nella directory corrente;
- Comandi offerti dal sistema operativo a riguardo:
  - *cd <dir>*: change directory. La directory indicata diviene la nuova directory corrente
  - *pwd*: Print Working Directory. Mostra la directory corrente.



# Esempio di directory ad albero





## Periferiche e file speciali

---

- In Unix le periferiche sono viste come **file speciali**, presenti nella directory */dev*.
- Sulle periferiche è possibile eseguire **open**, **read** e **write** (purchè abbia senso), **close** ma non **creat**
- Ogni programma all'avvio, dispone di **3 descrittori di file** aperti: **stdin (0)**, **stdout (1)**, **stderr (2)** associati alla console corrente (tastiera e video). Nella tabella dei file aperti di un processo, i primi 3 elementi sono associati a questi descrittori.
- I descrittori standard possono essere rediretti su altri file. Ad esempio:

- ```
close (1);          /* chiude stout e libera il descrittore 1 */  
fd = open ("./outputfile", O_WRONLY);  
/* il descrittore 1 viene associato a outputfile, quindi ogni  
operazione su 1 viene eseguita su outputfile */
```
- (stile di programmazione disdicevole, usate invece `dup2 ()`)



## Mapping del file system logico sul dispositivo fisico

---

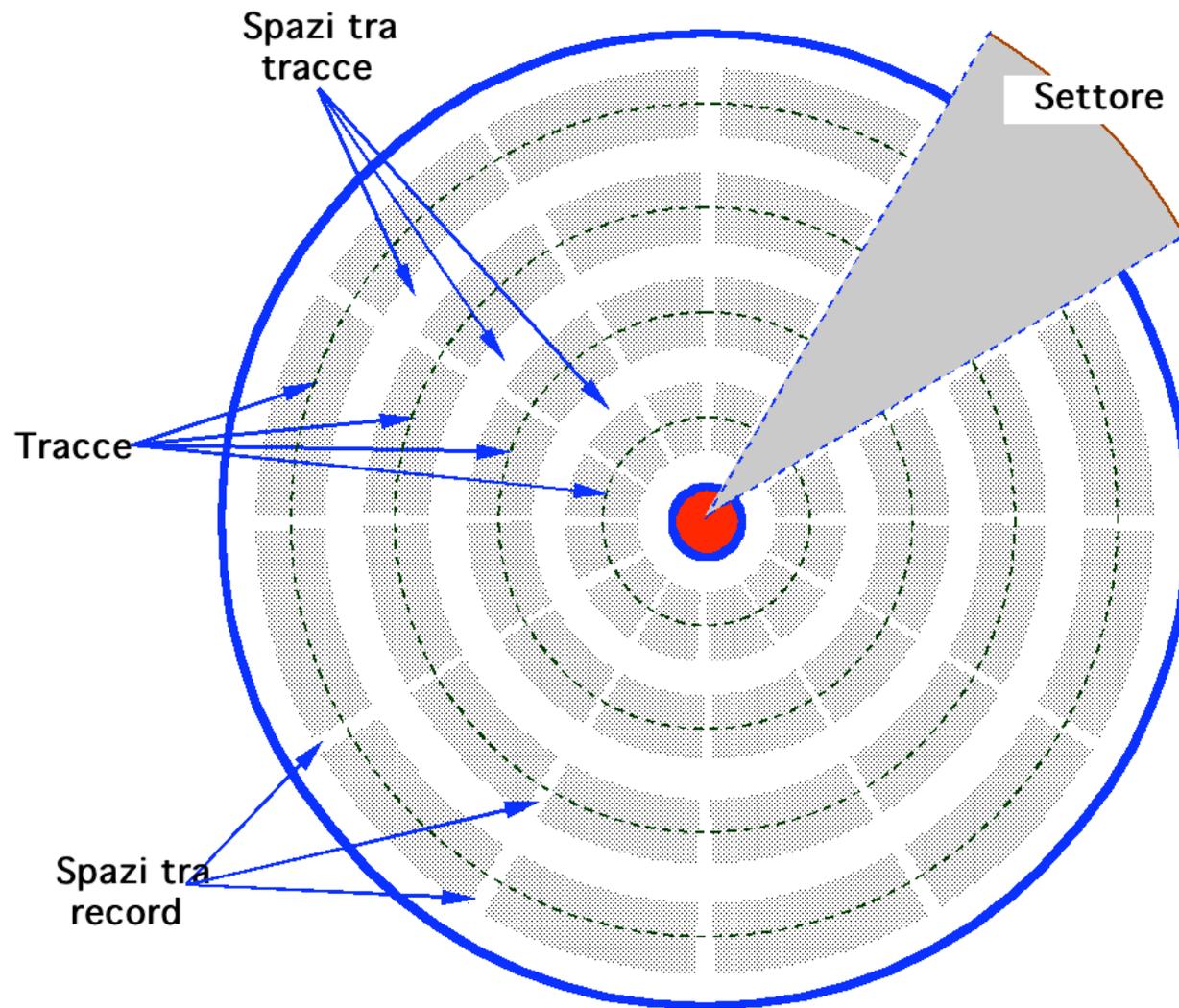
- Il file system svolge le sue funzioni accedendo ai dispositivi tramite l'invocazione di routine messe a disposizione dal **driver** del disco
- Il driver del disco fornisce una interfaccia di accesso ai dati su disco nascondendone le caratteristiche fisiche
- Il disco, a livello di file system, è rappresentato da un dispositivo logico, chiamato **volume** composto da un array di **blocchi**
  - Il **blocco** è una porzione di disco costituita da un numero intero di byte che viene trasferita in memoria con un'unica operazione



## Struttura del dispositivo fisico: tracce e settori

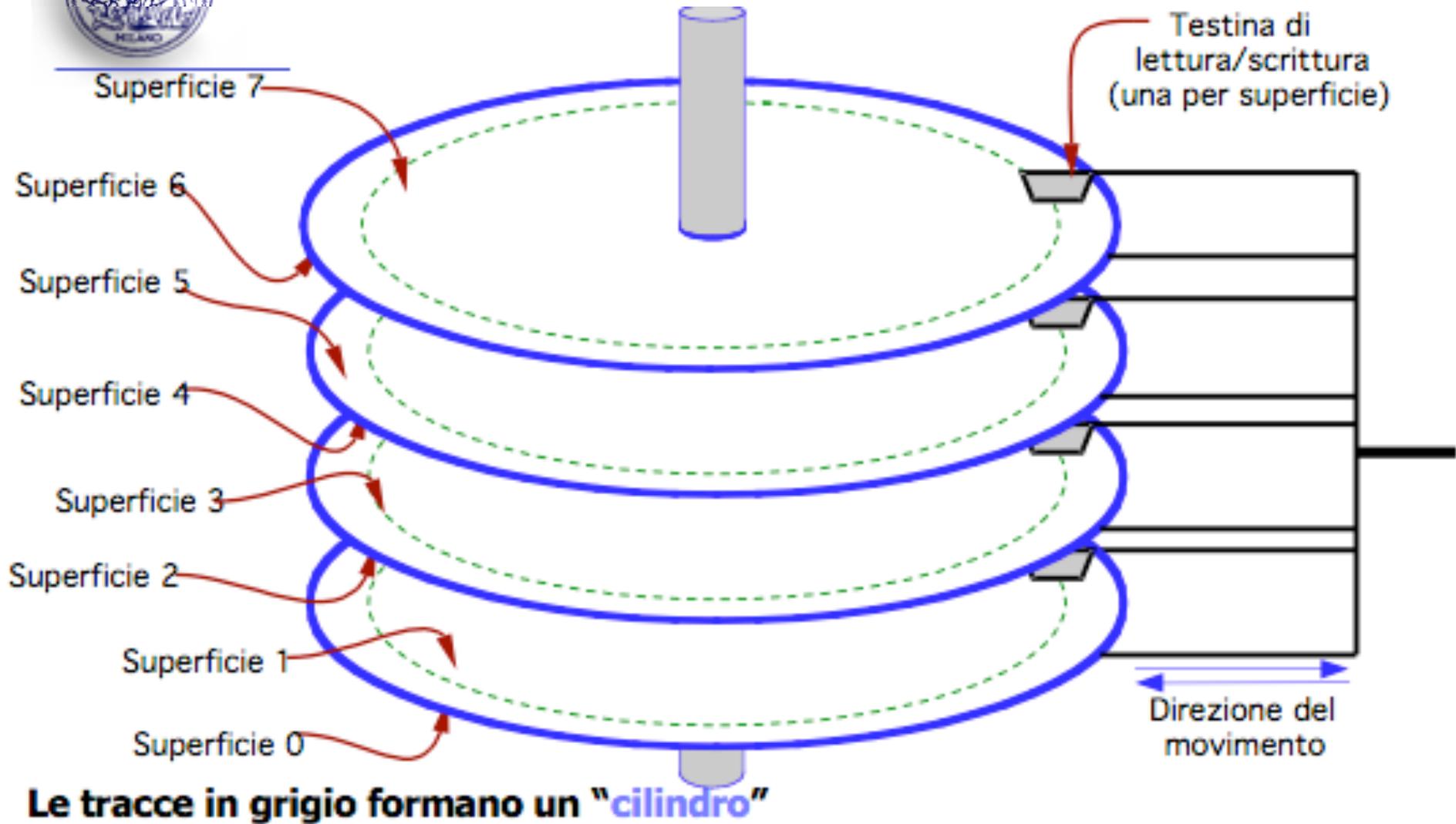
---

- **Traccia (track):** sequenza circolare di bit scritta mentre il disco compie una rotazione completa:
    - la larghezza di una traccia dipende dalla dimensione della testina e dall'accuratezza con cui la si può posizionare;
    - la densità radiale va da 800 a 2000 tracce per centimetro (5-10  $\mu\text{m}$  per traccia);
    - tra una traccia e l'altra c'è un piccolo spazio di separazione (**gap**).
  - **Settore (sector):** parte di una traccia corrispondente a un settore circolare del disco:
    - un settore contiene 512 byte di dati, preceduti da un preambolo, e seguiti da un codice di correzione degli errori;
    - la densità lineare è di circa 50-100kbit per cm (0.1-0.2  $\mu\text{m}$  per bit);
    - tra settori consecutivi si trova un piccolo spazio (**intersector gap**).
  - **Formattazione:** operazione che predispone tracce e settori per la lettura/scrittura.
-





# Geometria di un Hard Disk





# Prestazioni dei dischi

---

- **Tempo di accesso (ms o  $10^{-3}s$ )**
  - **Seek time**
    - la testina deve arrivare alla traccia giusta;
    - dipende dalla meccanica (5-15 ms, 1 per tracce adiacenti).
  - **Latency**
    - il disco deve ruotare fino a portare il dato nella posizione giusta;
    - dipende dalla velocità di rotazione (5400-10800 RPM  $\Rightarrow$  2.7-5.4ms).
- **Transfer Rate (MB/s)**
  - **Velocità di trasferimento del disco**
    - dipende dalla densità di registrazione e dalla velocità di rotazione;
    - un settore di 512 byte richiede fra 25 e 100  $\mu$ sec (5-20 MB/sec).
  - **Velocità di trasferimento del sistema di controllo**
    - SCSI vs. EIDE



## Struttura del dispositivo logico: Volume

---

- UNIX gestisce i file system a livello logico considerandoli come dispositivi logici (**volumi**) identificati da un numero
- Il volume è composto da una sequenza di **blocchi** logici (vettore di blocchi), di dimensioni fisse pari a un multiplo di 512 byte (configurabile a livello di sistema).
- La conversione tra indirizzi del dispositivo logico (**volumi e blocchi**) e indirizzi fisici (**tracce e settori**) è realizzata dal **driver del disco**.



## Blocco di un volume

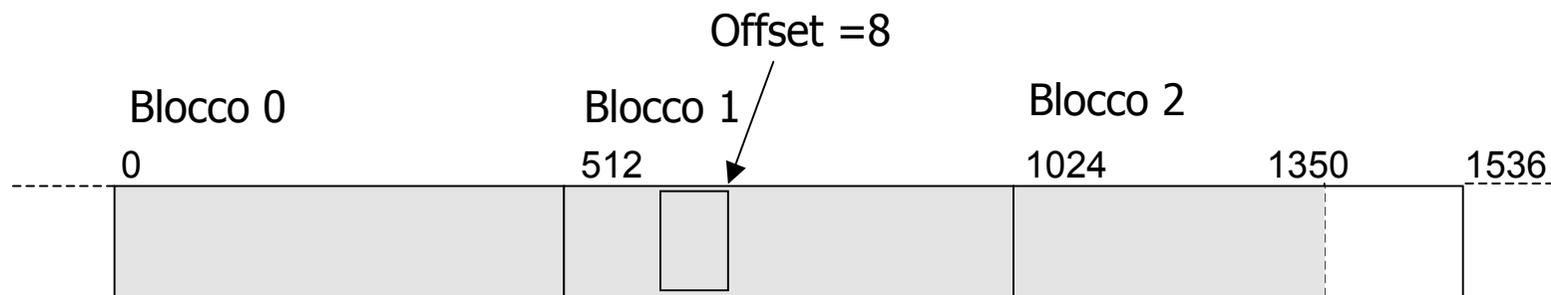
---

- Il blocco è l'unità di informazione trasferita con un solo accesso a disco, quindi è un multiplo del settore
- Un blocco può essere letto o scritto in un buffer di memoria centrale di ugual dimensione
- Il **gestore dei buffer** (buffer cache - parte del gestore della memoria virtuale) gestisce le aree di buffer cercando di minimizzare le letture multiple a disco
- Quando il **file system** richiede la lettura di un blocco interagisce con il **gestore dei buffer** che, a sua volta, può interagire con il **driver del disco**



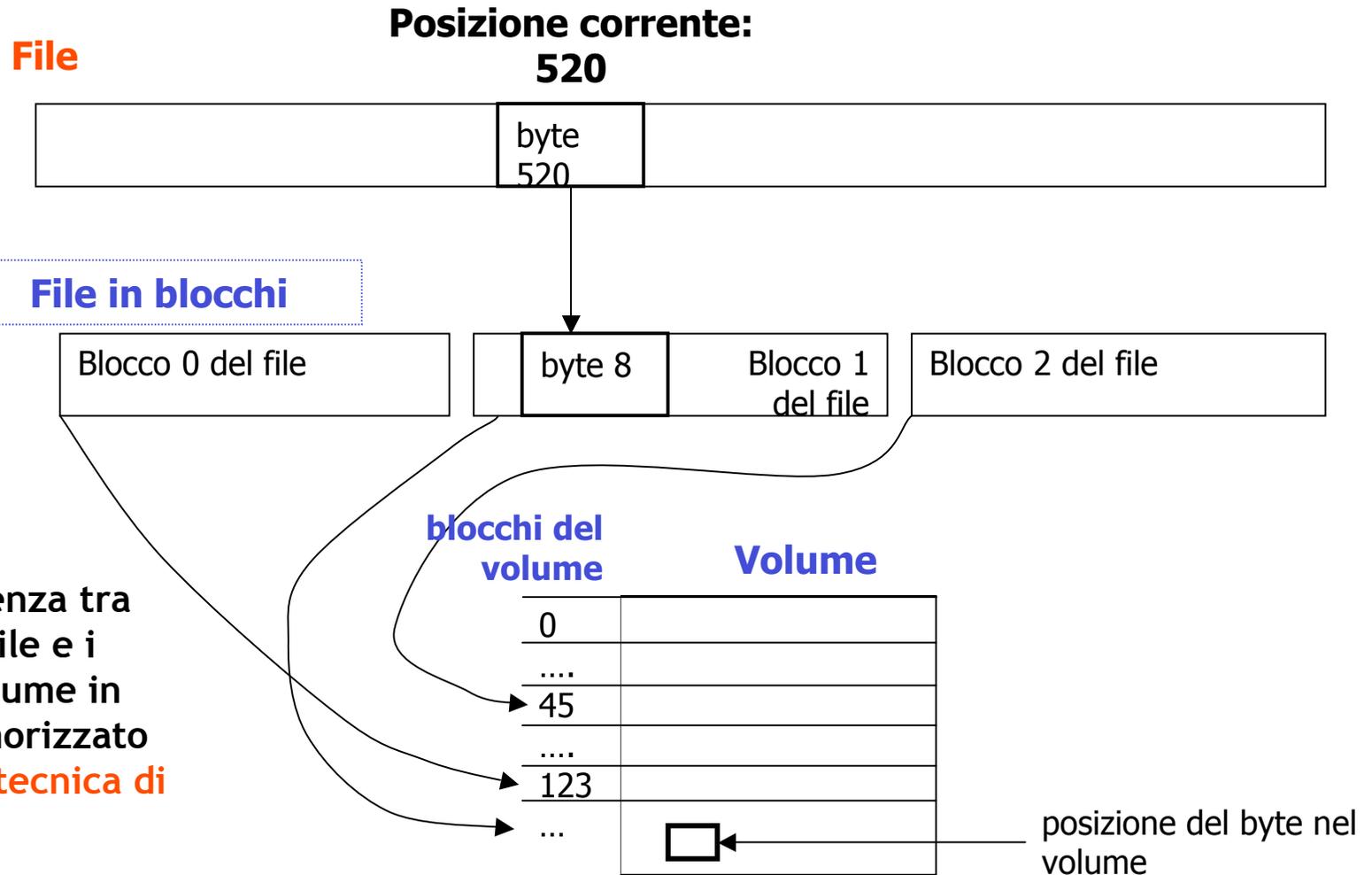
## Struttura logica di un file

- Il file è visto dal file system come una sequenza di **blocchi** numerati logicamente a partire da 0 all'interno del file
- Le operazioni read, write etc. fanno riferimento alla **posizione corrente** nel file
- Ad esempio: ricerca del byte 520 (posizione corrente):
  - Blocco =  $520 / 512 = 1$  (divisione intera)
  - Offset nel blocco =  $520 \% 512 = 8$  (resto della divisione intera)





# Allocazione del file nel volume



La corrispondenza tra blocchi di un file e i blocchi del volume in cui viene memorizzato dipende dalla **tecnica di allocazione**



## Tecnica di allocazione: i-node

---

- ❑ I sistemi operativi unix-like utilizzano uno **schema di allocazione indicizzata combinata** dei blocchi dei file sui blocchi del volume
- ❑ La tecnica è realizzata mediante una **lista (*i-list*)** i cui elementi sono chiamati ***i-node* (*index-node*)**
- ❑ Ogni *i-node* contiene la lista degli attributi e degli indirizzi dei blocchi di disco a cui sono associati i blocchi del file
- ❑ Un file esiste nel file system se esiste il corrispondente *i-node*.



## Struttura del volume

---

|                |             |              |              |
|----------------|-------------|--------------|--------------|
| Blocco di boot | Super block | Lista i-node | Blocchi dati |
|----------------|-------------|--------------|--------------|

(Viene presentata una possibile soluzione semplificata, non corrispondente ad una particolare implementazione).

Un volume contiene l'**immagine del file system** ad esso associato

- ❑ **Blocco 0**: tipicamente contiene il boot record, cioè codice di avvio del sistema operativo (altrimenti allocato ma non utilizzato)
- ❑ **Superblock**: contiene lo stato del file system (dimensioni, spazio libero, timestamp, contatore montaggi, ...)
- ❑ **Tabella degli i-node**: ogni elemento è un **i-node**
- ❑ **Blocchi di dati**: che contengono i blocchi dei file



## Implementazione dei file

---

- Ogni file **regolare**, **directory** o **speciale** ha associato un i-node
- Gli i-node dei file in un disco sono memorizzati in sequenza all'inizio del volume e costituiscono la i-list
- In una directory, i file contenuti (siano essi regolari, directory o speciali) sono rappresentati da una tabella le cui voci sono coppie **< i\_node, nome\_file >**
- Dato il numero dell'i-node, si può localizzare la posizione del file calcolandone l'indirizzo su disco



## Informazioni nell'i-node

---

- I dettagli implementativi dipendono dal filesystem utilizzato.
- Esempio: facciamo riferimento all'*ext2fs*:
  - Tipo di file (regolare/speciale/directory)
  - Diritti (9 bit rwx) e altri flag
  - Contatore di hard link al file
  - Identificatore utente e gruppo proprietari del file
  - Lunghezza del file in byte e in blocchi (valido solo per file regolari e directory)
  - 15 indirizzi di blocchi dati, che contengono i dati del file. Non sono significativi nel caso di file speciale
  - Timestamp in cui il file è stato creato, letto e scritto per l'ultima volta;



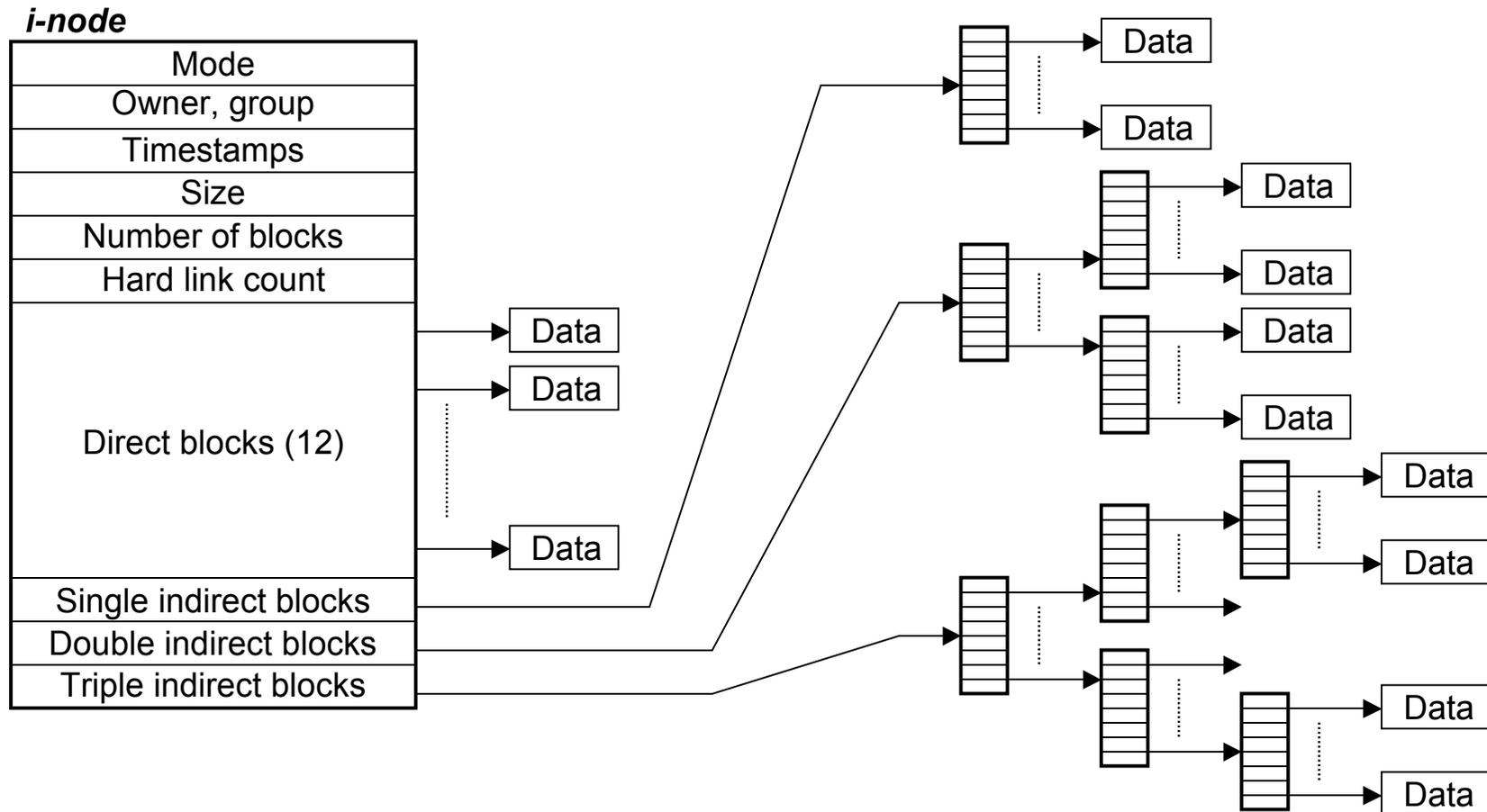
## Allocazione indicizzata combinata in ext2fs

---

- ❑ I primi 12 indirizzi su disco ( $i\_block[0] \dots i\_block[11]$ ) rappresentano direttamente blocchi del file su disco
  - ❑ Per file di dimensioni maggiori,  $i\_block[12]$  contiene l'indirizzo di un blocco del disco chiamato **single indirect block** che a sua volta contiene altri indirizzi diretti di blocchi del disco
  - ❑ Se questo non è sufficiente,  $i\_block[13]$  mette a disposizione un altro blocco, chiamato **double indirect block** che contiene l'indirizzo di un blocco che a sua volta contiene una lista di single indirect block
  - ❑ Esiste anche un **triple indirect block** ( $i\_block[14]$ ), nel caso la doppia indicizzazione non sia sufficiente
  - ❑ (confrontare con pag. 469 [Bovet])
-



# Rappresentazione dell'i-node (ext2fs)





## Contenuto del superbloc

---

Contiene informazioni necessarie a gestire il filesystem:

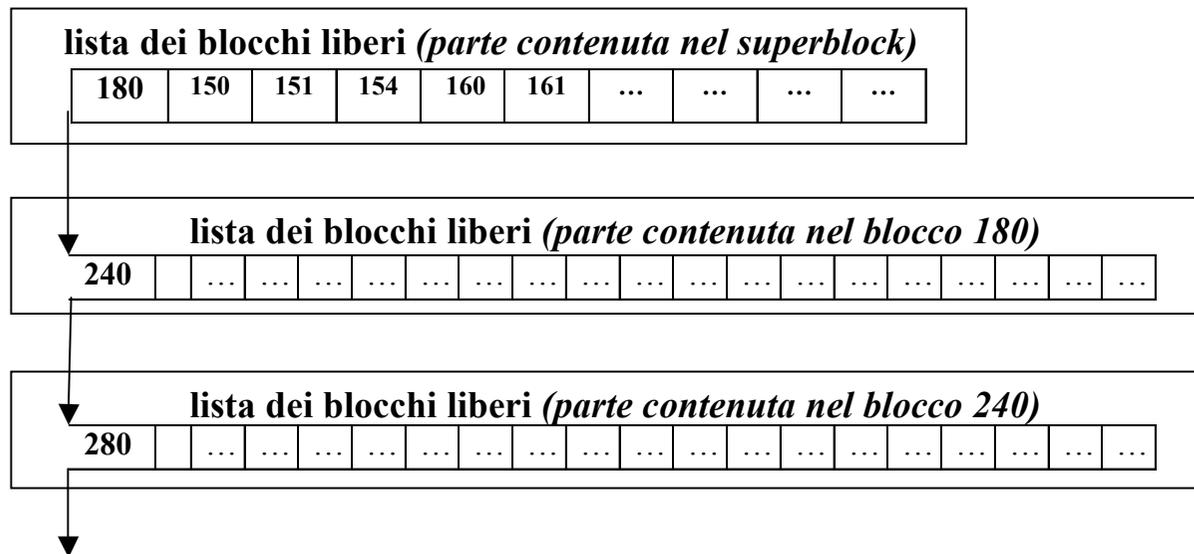
- ❑ dimensione del volume in blocchi;
- ❑ numero totale degli i-nodes;
- ❑ numero di blocchi liberi;
- ❑ numero di i-nodes liberi;
- ❑ dimensione del blocco;
- ❑ nome del volume;
- ❑ conteggio dei montaggi e timestamp dell'ultimo montaggio
- ❑ numero di elementi della **lista o bitmap dei blocchi liberi**;
- ❑ altre informazioni ausiliarie.

(vedere pagina 455 [Bovet])



# Gestione dello spazio libero

- All'atto della **creazione** e **scrittura** di un file è necessario individuare sul disco il primo blocco **libero** da allocare al file
- una soluzione possibile è una **lista concatenata di blocchi liberi** (in genere il primo elemento della lista punta alla prossima lista ...)



- ext2fs usa invece una bitmap per indicare i blocchi liberi (1 bit indica la disponibilità di 1 blocco)



# Strutture dati del file system

---

UNIX dispone di tre tabelle per la gestione dei file

□ **Tabella dei descrittori di file utente:**

- tabella associata ad ogni processo utente contenente una riga per ogni file aperto dal processo (indice = descrittore del file) con l'indirizzo della riga della tabella globale dei file aperti relativa al file

□ **Tabella globale dei file aperti:**

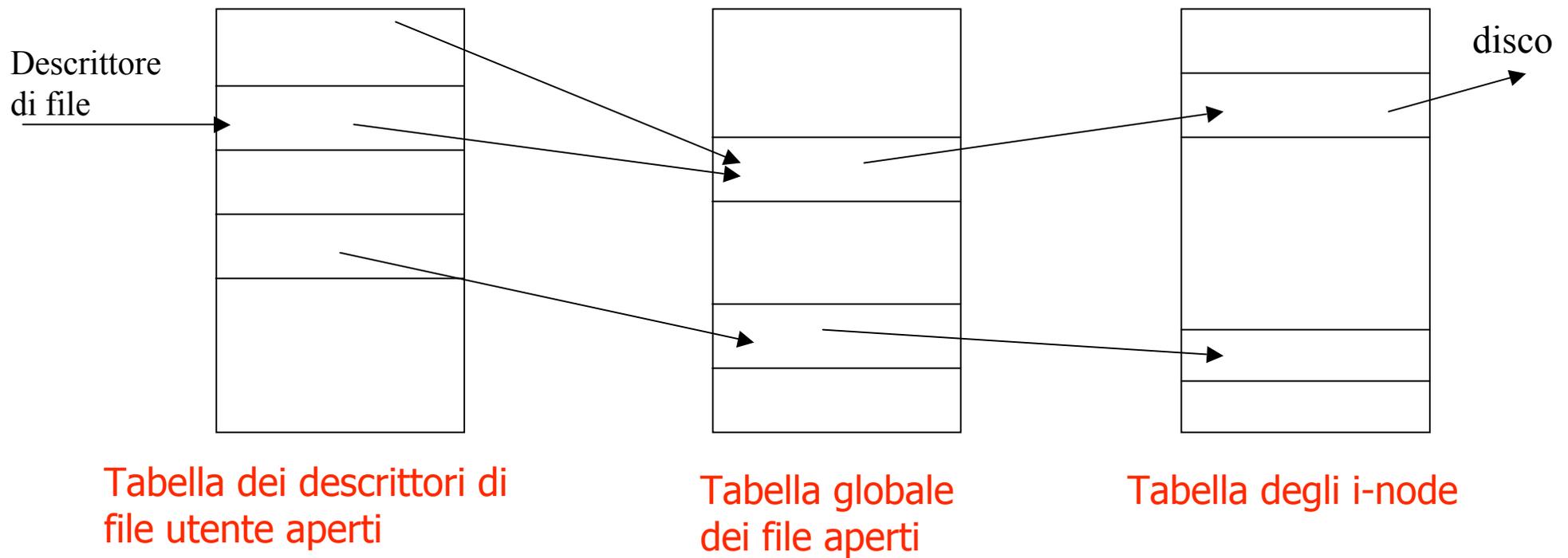
- tabella del sistema operativo che contiene una riga per ogni file aperto nel sistema;
- ogni riga contiene l'indirizzo del corrispondente i-node nella tabella degli i-node, l'indicatore della posizione corrente del file e il contatore al numero di riferimenti (link) da parte dei processi a questo file

□ **Tabella degli i-node:**

- le righe contengono la copia in memoria degli i-node del volume per maggiore efficienza nei riferimenti.



# Strutture dati del file system





## Operazioni sui file e strutture dati del filesystem

---

- **open ( )** : la richiesta di apertura di un file al sistema operativo prevede le seguenti operazioni:
  - riservare una riga nella tabella globale dei file aperti e porre il contatore dei riferimenti a 1 e l'indicatore di posizione a 0
  - riservare un riga nella tabella del processo corrente dei file aperti del processo e scrivervi l'indice della riga corrispondente nella tabella globale dei file aperti;
  - determinare l'i-node corrispondente al file il cui path name è stato indicato nella richiesta e scriverlo nella riga della tabella globale dei file aperti allocata. Per fare ciò, per ogni directory che appare nel pathname, in sequenza:
    - determinare quali sono i blocchi dati associati ad essa (consultando l'i-node) e caricarli in memoria;
    - cercare in tali blocchi il componente successivo del pathname (file o dir);
    - determinare l'i-node di tale componente; se è una dire, ripetere dal passo 1;
  - restituire al processo un descrittore che è l'indice della riga di cui sopra nella tabella dei file aperti del processo



## Operazioni sui file e strutture dati del file system

---

- **creat()** : richiesta di creazione di un file al sistema operativo:
  - verificare che esista un i-node libero
  - verificare che esistano blocchi liberi
  - riservare una riga nella tabella globale dei file aperti e porre il contatore dei riferimenti a 1 e l'indicatore di posizione a 0;
  - riservare un riga nella tabella dei file aperti dal processo e scrivervi l'indice della riga corrispondente nella tabella globale dei file aperti
  - allocare l'i-node al file, inizializzando le informazioni in esso contenute e aggiorna le informazioni relative alla i-list
  - aggiornare il contenuto della directory che contiene il file
  - scrivere nella riga della tabella globale dei file aperti associata al file il valore dell'i-node corrispondente al file
  - restituire al processo un descrittore costituito dal numero di riga della tabella dei file aperti dal processo



## Operazioni sui file e strutture dati del file system

---

### ❑ **dup () :**

- genera un nuovo descrittore per un file già presente nella tabella dei file aperti del processo (=duplica un descrittore);
- il nuovo descrittore è il primo numero di descrittore libero;
- la posizione corrente dell'originale e del duplicato è sempre unica, visto che esiste una sola riga nella tabella globale dei file aperti;

### ❑ **fork () :**

- vengono duplicate anche le tabelle dei file aperti da un processo;
- la posizione corrente è unica, esiste una sola riga nella tabella globale dei file aperti;

### ❑ **Apertura dello stesso file da parte di 2 processi:**

- si aggiunge una riga nella tabella globale dei file aperti;
- ogni processo ha una propria posizione corrente del file;



## Operazioni sui file e strutture dati del file system

---

### □ **Letture o scrittura:**

- `read()` e `write()` prendono per argomento il descrittore del file su cui operare, che permettono di seguire i puntatori nelle tre tabelle e, tramite l'i-node, identificare i blocchi di dati del file

### □ **Chiusura di un file:**

- Il sistema operativo libera la corrispondente riga dei descrittori di file utente aperti
- Decrementa il contatore dei riferimenti nella tabella globale dei file e se =0, libera la riga



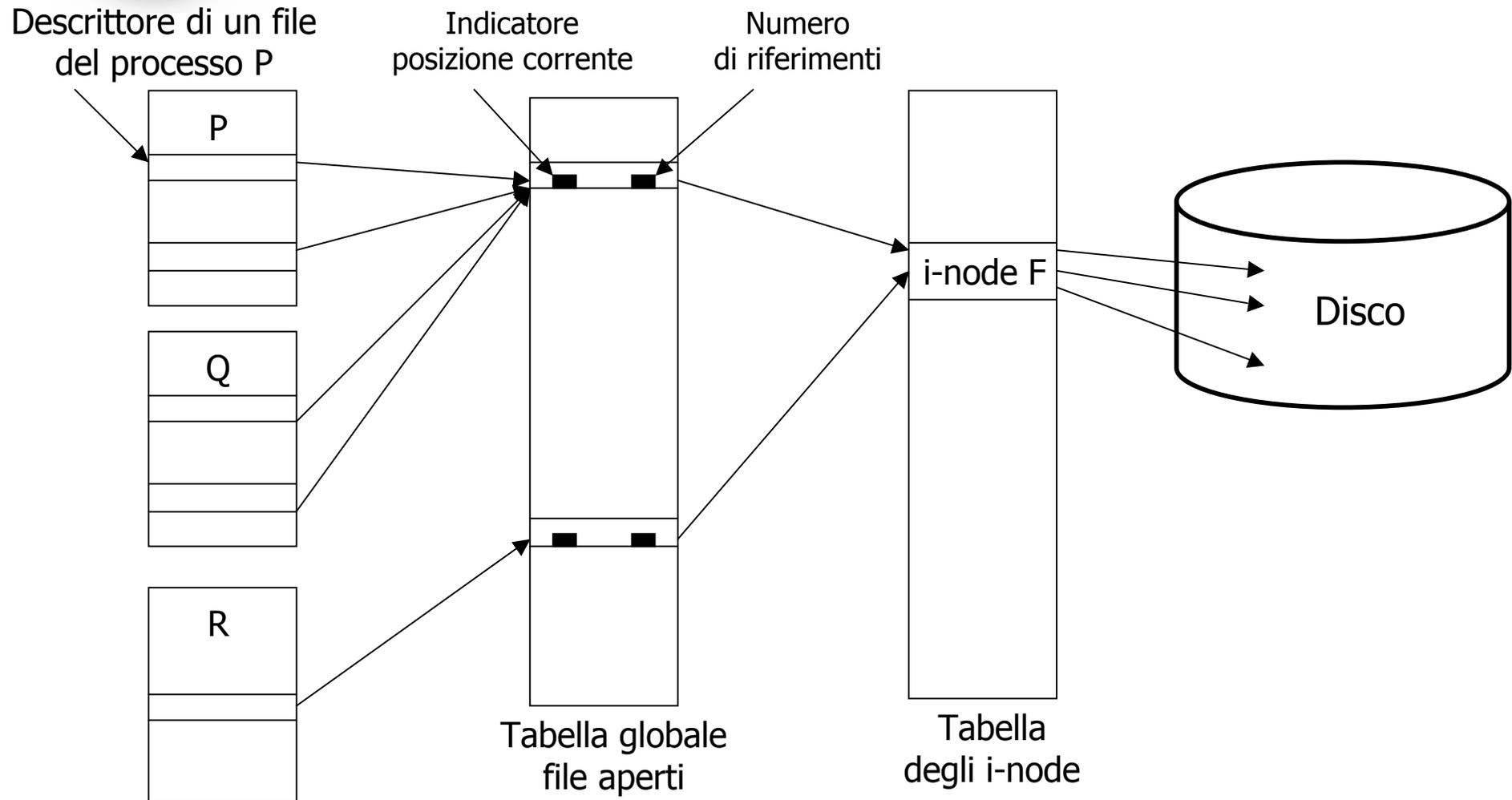
## Esercizio:

---

- Considerate il seguente scenario:
  - Il processo P apre un file: `fd = open ( . . . ) ;`
  - Il processo P esegue una dup: `fd2=dup ( fd ) ;`
  - Il processo P genera un figlio Q: `fork ( ) ;`
  - Un terzo processo, R apre lo stesso file F: `fd = open ( . . . ) ;`
  
- Rappresentate lo stato delle strutture dati del file system descritte in precedenza.



# Soluzione





## Due esempi di codice

---

- Confrontiamo il comportamento di due programmi che compiono le stesse azioni:
  - il primo fa uso di stream e funzioni della libreria C (`fopen()`, `fclose()`, `fprintf()`, `fseek()`, `ftell()`)
  - il primo fa uso di file descriptor e funzioni di sistema operativo (`open()`, `close()`, `write()`, `lseek()`)



# Esempio 1

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

int main(int argc, char* argv[])
{
    int error, status;
    long posiz;
    pid_t pid;
    FILE *fp;
    char *s1 = "Nel mezzo del cammin di nostra vita";
    char *s2 = "mi ritrovai in una selva oscura";
    char *s3 = "che la diritta via era smarrita";

    printf("Verifica con fopen\n\npadre: apro il file \n");
    fp=fopen("DivCom", "w");
    if (fp==NULL) {
        fprintf(stderr, "Errore di apertura del file\n");
        exit(EXIT_FAILURE);
    }

    printf("padre: posizione corrente = %d\n", ftell(fp));

    printf("padre: scrivo sul file\n");
    fprintf(fp, "%s\n", s1);
    fprintf(fp, "%s\n", s2);
    fprintf(fp, "%s\n", s3);

    printf("padre: posizione corrente = %d\n", ftell(fp));
```



## Esempio 1 (continua)

---

```
printf("padre: eseguo la fork\n");
pid = fork();
if (pid == 0) {
    /* nel figlio: legge posizione corrente, che sarà identica a quella del padre */
    printf("figlio: posizione corrente = %d\n", ftell(fp));

    printf("figlio: riposiziono all' inizio del file\n");
    fseek(fp, (long) 0, SEEK_SET);

    /* legge posizione corrente, non più identica a quella del padre */
    printf("figlio: posizione corrente = %d\n", ftell(fp));

    /* chiude il file e termina */
    close(fp);
    exit(0);
} else {
    /* nel padre */
    printf("padre: generato figlio: %d\n",pid);
    /* legge la posizione corrente del file, dopo la fseek del figlio */
    wait(&status);
    printf("padre: figlio terminato: %d\n",pid);
    printf("padre: posizione corrente = %d\n", ftell(fp));
}
close(fp);
return 0;
}
```

---



## Risultato dell'esecuzione - esempio 1

---

Verifica con fopen

```
padre: apro il file
padre: posizione corrente = 0
padre: scrivo sul file
padre: posizione corrente = 100
padre: eseguo la fork
padre: generato figlio: 22783
figlio: posizione corrente = 100
figlio: riposiziono all' inizio del file
figlio: posizione corrente = 0
padre: figlio terminato: 22783
padre: posizione corrente = 100
```

**Morale:** i figli ereditano la gli stream aperti dal padre. Tuttavia gli stream contengono il proprio stato nello spazio di indirizzamento privato di ciascun processo, quindi le posizioni correnti sono **indipendenti**.

---



## Esempio 2

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

int main(int argc, char* argv[])
{
    int error, status, fd;
    pid_t pid;
    char *s1 = "Nel mezzo del cammin di nostra vita";
    char *s2 = "mi ritrovai in una selva oscura";
    char *s3 = "che la diritta via era smarrita";

    printf("Verifica con utilizzo open\n\npadre: apro il file \n");
    fd = open("DivCom", O_RDWR | O_CREAT, 0666);
    if (fd < 0) {
        fprintf(stderr, "Errore di apertura del file\n");
        exit(EXIT_FAILURE);
    }
    printf("padre: posizione corrente = %d\n", lseek(fd, 0, SEEK_CUR));
    printf("padre: scrivo sul file\n");
    write(fd, s1, strlen(s1));    write(fd, "\n", 1);
    write(fd, s2, strlen(s2));    write(fd, "\n", 1);
    write(fd, s3, strlen(s3));    write(fd, "\n", 1);
    printf("padre: posizione corrente = %d\n", lseek(fd, 0, SEEK_CUR));
```



## Esempio 2 (continua)

---

```
/* il padre genera un figlio, che eredita file e posizione corrente del padre */
printf("padre: eseguo la fork\n");
pid = fork();

if (pid == 0) {
    /* nel figlio: legge posizione corrente, identica a quella del padre */
    printf("figlio: posizione corrente = %d\n", lseek(fd, 0, SEEK_CUR));

    /* il figlio si riposiziona all'inizio del file */
    printf("figlio: riposiziono all' inizio del file\n");
    lseek(fd, 0, SEEK_SET);

    /* legge posizione corrente, non piu' identica a quella del padre */
    printf("figlio: posizione corrente = %d\n", lseek(fd, 0, SEEK_CUR));
    close(fd);
    /* chiude il file e termina */
    exit(0);
} else {
    printf("padre: generato figlio: %d\n",pid);
    /* legge la posizione corrente del file, dopo la fseek del figlio */
    wait(&status);
    printf("padre: figlio terminato, posizione corrente = %d\n", lseek(fd, 0, SEEK_CUR));
}
close(fd);
return 0;
}
```

---



## Risultato dell'esecuzione - esempio 2

---

Verifica con open

```
padre: apro il file
padre: posizione corrente = 0
padre: scrivo sul file
padre: posizione corrente = 100
padre: eseguo la fork
padre: generato figlio: 22786
figlio: posizione corrente = 100
figlio: riposiziono all' inizio del file
figlio: posizione corrente = 0
padre: figlio terminato, posizione corrente = 0
```

Morale: i figli ereditano la tabella dei file aperti del padre. Hanno gli stessi file descriptor, che puntano alle stesse celle della tabella globale dei file aperti. **Condividono** quindi le posizioni correnti nei file.

---



## Protezione e diritti d'accesso

---

- E' utile che un filesystem **protegga** i propri dati da *accessi indesiderati* (per garantire riservatezza o integrità, o impedire eliminazioni accidentali);
- Quindi è necessario definire una **politica di accesso** e implementarla
- Esempi di banali politiche di accesso:
  - *Ogni utente accede solo ai propri file*
  - *Ogni utente accede a tutti i file*
- **Accesso controllato**: si definiscono **regole di accesso** ai file sulla base di:
  - identità dell'utente e gruppo di lavoro dell'utente
  - proprietà dei file
  - **tipo di operazione** richiesta: lettura, scrittura, esecuzione



## Protezione con liste di controllo di accesso

---

- ❑ L'accesso e le operazioni consentite dipendono dall'identità dell'utente
- ❑ Ad ogni file è associata una *lista di accesso*, cioè una matrice che indica se un certo utente può eseguire una certa operazione;
- ❑ Quando un'operazione viene richiesta il sistema operativo controlla la lista di accesso per verificare se il richiedente ha il permesso di compierla.
- ❑ Questa soluzione presenta alcuni svantaggi:
  - le liste di accesso possono essere di dimensioni notevoli;
  - le liste devono essere create e mantenute per ogni file;
  - il tempo di accesso ad un file si allunga.



## Protezione dei file nei sistemi unix-like

---

- I sistemi operativi unix-like adottano una soluzione **semplificata** delle liste di accesso
- Gli utenti sono identificati in base a:
  - **Username**: identificativo dell'utente
  - **Group**: identificativo di gruppo, condiviso da più utenti
- Ciascun utente rispetto a un file cade in una delle seguenti classi (nell'ordine di valutazione elencato):
  - **Owner**: il proprietario del file
  - **Group**: i membri del gruppo proprietario del file
  - **All**: tutti gli altri utenti



## Protezione dei file nei sistemi unix-like

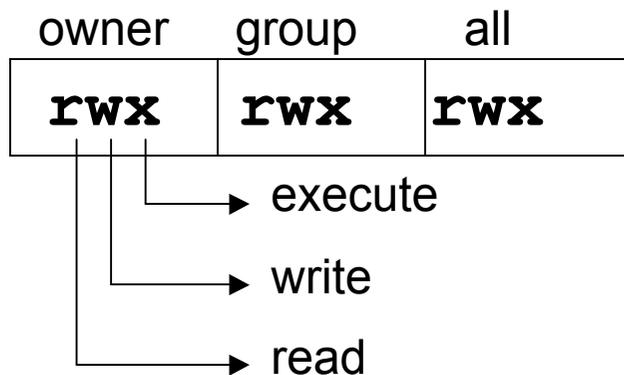
---

- Le operazioni su file sono raggruppate in tre classi:
  - *Read*: Lettura, copia
  - *Write*: Scrittura, modifica, eliminazione
  - *Execute*: Esecuzione
- Ad ogni file sono associati:
  - **Proprietario**
  - **Gruppo proprietario**
  - **Permessi**



## Protezione dei file nei sistemi unix-like

- I *permessi* sono tre gruppi di bit:
  - Ogni gruppo di bit si riferisce ad una delle tre classi di utenti
  - Ogni bit del gruppo si riferisce ad una delle tre operazioni



| owner      | group      | all        |
|------------|------------|------------|
| <b>rwX</b> | <b>rwX</b> | <b>rwX</b> |
| <b>111</b> | <b>101</b> | <b>101</b> |
| <b>7</b>   | <b>5</b>   | <b>5</b>   |