

## **u4-s1-file-system**

### **Sistemi Operativi**

#### **Unità 4: Il File System**

## **I dischi e i file system**

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## **Argomenti**

1. Il disco nei sistemi di elaborazione
  2. Dati e disco
  3. I file
  4. I Direttori
  5. Allocazione dei blocchi
  6. File System Comuni
  7. Tabella delle partizioni
- 

## **Definizione di Disco negli Elaboratori**

### **Definizione (*disco*)**

Dal modulo sulle [\*architetture degli elaboratori\*](#) (1), ricordiamoci che il *disco* è un *componente fondamentale dei sistemi di elaborazione*

- Permette di avere una memoria persistente
  - Sopravvive al riavvio dell'elaboratore
- È una memoria riscrivibile
  - Diversamente da ROM, PROM e EPROM

## **Tipologie di Disco**

### **Tipologie di Disco**

Ci ricordiamo che ci sono diverse tecnologie per costruire i dischi

- *Nastri magnetici*: obsoleti/storici
- *Dischi magnetici*: (o note come "HDD") i più usati
- *Stato solido (memorie flash)*: (o note come "SSD/formalmente EPROM") in ascesa ancora oggi

Ogni tipologia si differisce per *prestazioni, costi, affidabilità, meccanismo di accesso*.

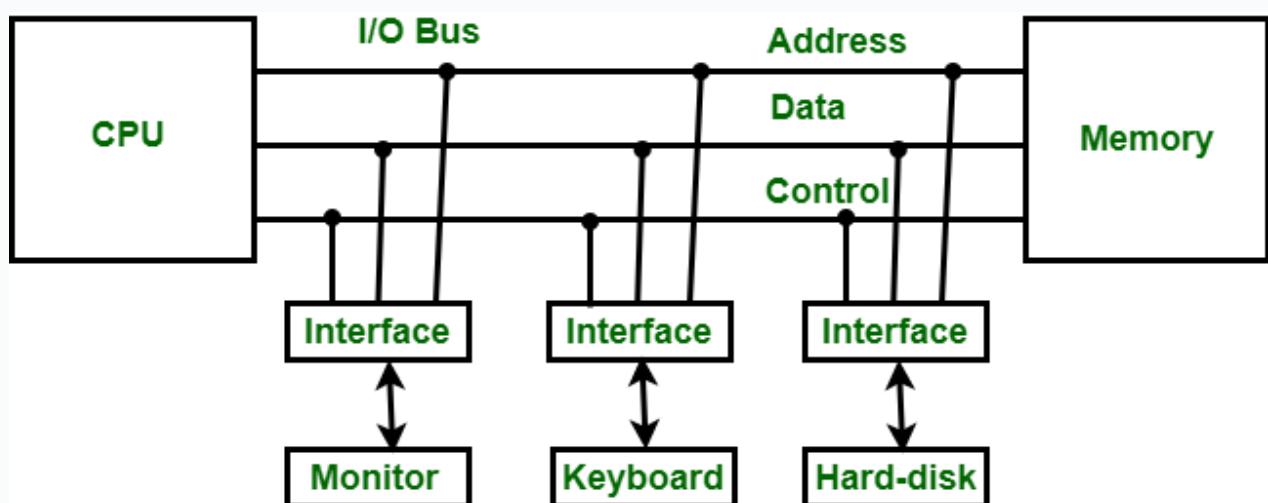
- Nei dischi magnetici la testina si sposta (HDD)
  - Accesso al disco non ha un tempo costante
  - Letture sequenziali preferite
  - Costo minore
- Memorie flash: tempo di accesso quasi costante (HDD)
  - Scrittura più lenta di lettura
  - Costo maggiore

## Accesso al Disco

### Definizione (interfaccia)

Il disco è un dispositivo *di I/O*, quindi la CPU lo utilizza attraverso un'*interfaccia*

### FIGURA (Schema di architettura)



### modo approssimato dell'accesso al disco

In prima approssimazione (ad alto livello), la CPU accede al disco nel seguente modo:

1. La CPU scrive nell'*interfaccia del disco* la *locazione di memoria* che vuole leggere o scrivere
  - Assieme a *informazioni di controllo* (e.g., se Read o Write)
  - L'accesso all'interfaccia avviene come a una qualunque locazione di memoria
2. Il disco *esegue* l'operazione
3. Il disco *setta dei flag* nell'interfaccia che segnalano che l'*operazione è conclusa*
4. La CPU, leggendo i flag, realizza che l'*operazione è terminata*
  - Eventualmente legge i dati dall'interfaccia (in caso di Read)
  - Nota come la tecnica del "*polling*" (1)

# Ottimizzazione dell'Accesso ai Dischi

Esistono altre tecniche per rendere **più efficiente** l'accesso al disco. Le accenniamo, e sono le seguenti.

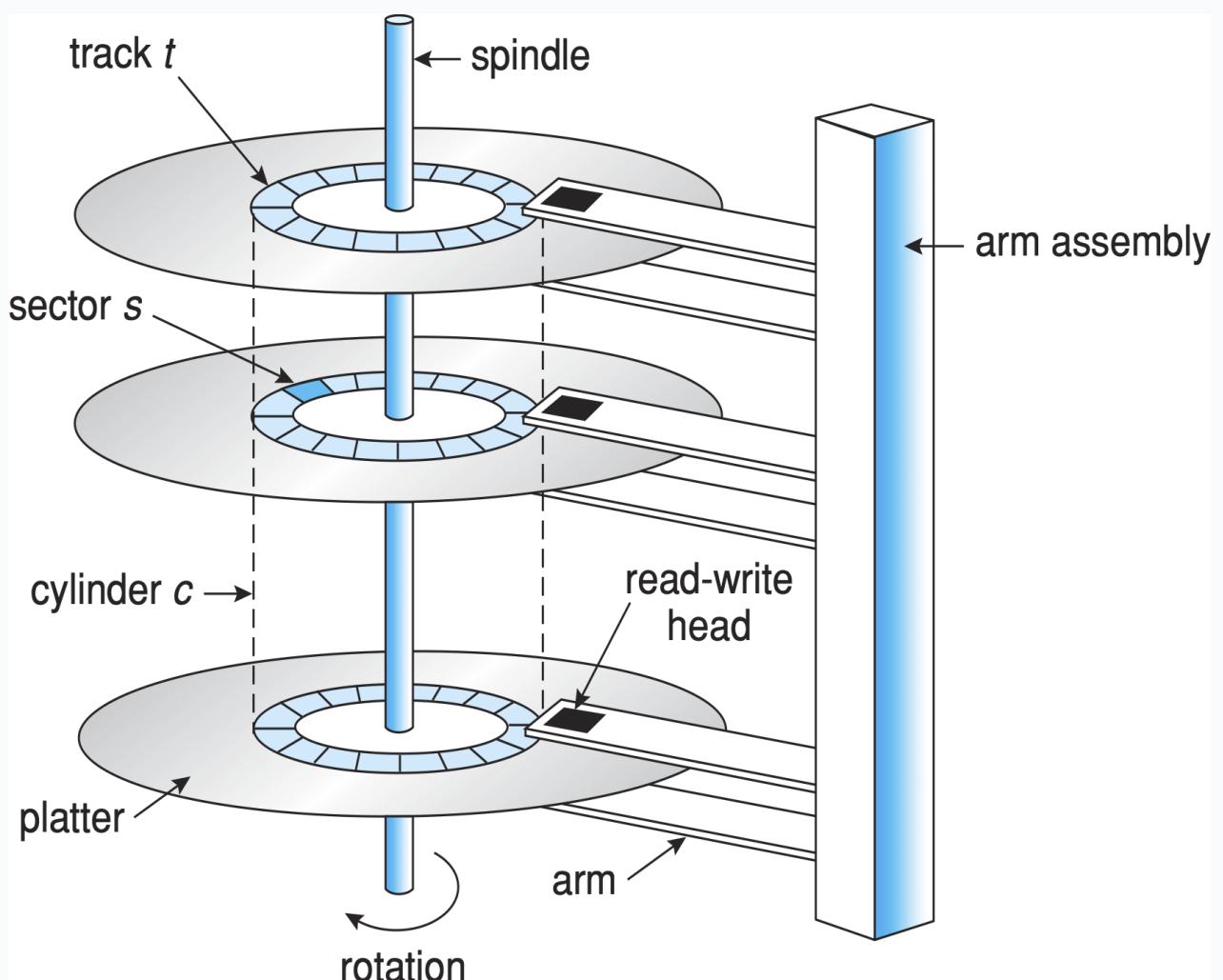
- **DMA**: Direct Memory Access
  - La CPU istruisce il disco sul compito da effettuare
  - Il DMA controller legge/scrive autonomamente i dati in memoria
- **Caching**: Il sistema operativo tiene in RAM **le porzioni di disco più lette** (quindi abbiamo delle zone già preventivamente copiate)

## Organizzazione dei Dati nei Dischi

### Dischi Magnetici (HDD)

In disco magnetico, i dati sono organizzati in **tracce concentriche e settori**. Abbiamo dei **dischi in parallelo**, con **due facciate** ognuna.

**FIGURA** (*Schema di un disco magnetico*)



### Dischi a stato solido (SSD)

I dischi a stato solido invece sono **simili alle celle di memorie RAM**.

- Matrice di celle
- 

## Dati e Dischi

### Definizione di Blocco

#### Definizione (*blocco*)

I dischi sono utilizzabili come un *vettore di blocchi*

- Blocchi di  $512B - 8KB$

#### FIGURA (*L'idea dei blocchi*)

Blocco 0	Blocco 1	Blocco 2	Blocco 3	Blocco 4
Blocco 5	Blocco 6	Blocco 7	Blocco 8	Blocco 9
		.....		

#### Definizione (*File System, cenno*)

Il *File System* permette di organizzare questi blocchi per avere

- *File* di grandezza variabile (quindi la sostanza)
  - *Organizzati* in un albero di cartelle (quindi la struttura)  
Quindi possiamo definire il *file system* come un *algoritmo per rappresentare le cartelle in un disco*.
- 

## I File

#### Definizione. (*File*)

I *file* sono una *sequenza ordinata di bit* che contengono delle *informazioni*.

Hanno un *nome* e degli *attributi* (ovvero metadati):

- Identificativo nel SO
- Permessi
- Tempo di creazione, di ultimo accesso

I file sono *organizzati* in *direttori* o *cartelle* o *folder* o *directory*

- Possono essere create, modificate o cancellate *come i file* (vedremo che in realtà le cartelle non sono altro che dei "*file speciali*")
- A differenza dei file non contengono byte *ma altri direttori o file*

## Operazioni sui File.

Sui file, un programma (tramite System Call del SO) può effettuare le *operazioni* di:

- Creazione
- Lettura
- Scrittura
- Cancellazione
- *Seek* (movimento del cursore) (ricordiamoci che l'accesso ai file è *sequenziale*)

### Osservazione. (*Accesso sequenziale dei file*)

Le operazioni di lettura e scrittura sono sempre *sequenziali*. Il file viene letto/scritto byte per byte, tramite un *cursore*. E' possibile riposizionare il cursore tramite l'operazione di *seek*

---

## I Direttori

### Definizione. (*Directory/Direttori*)

Una *Directory* è un contenitore di nodi (file o altre Directory)

- Organizzazione tipicamente ad albero.
- Tipicamente è un grafo *senza cicli* (a meno che consideriamo i *link*, come faremo dopo)

### Definizione. (*Partizioni*)

Un disco è diviso in una o più *partizioni*

Ogni partizione contiene un *albero di direttori*, in particolare

- Vi è un direttorio *radice*
- *Tutti* i file e direttori vi son contenuti

## Operazioni con le cartelle.

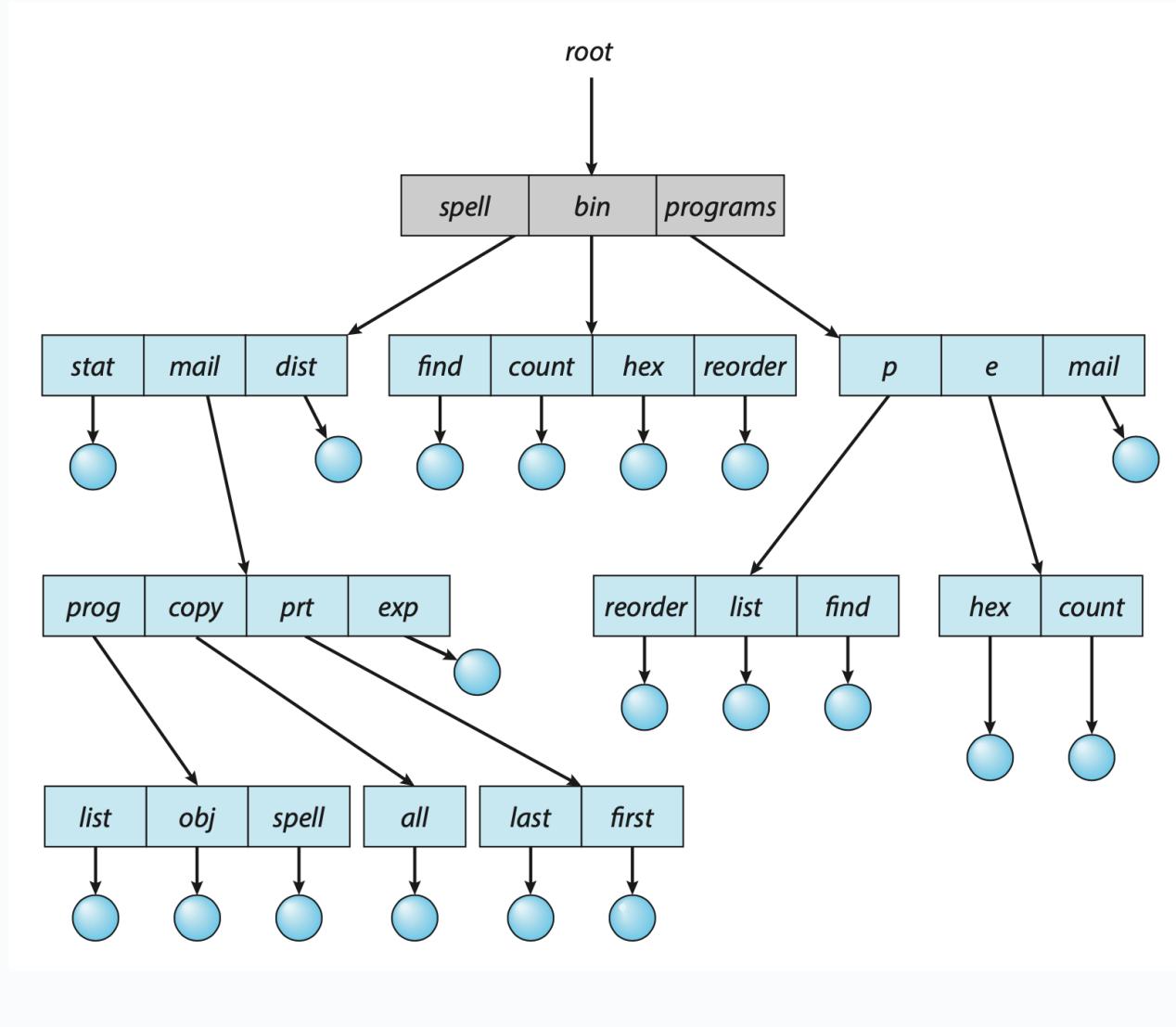
Operazioni sui direttori sono *simili a quelle su file*:

- Creazione
- Rimozione
- *Listing*
- *Renaming*

*Ricordare!* Il SO mette a disposizione delle *System Call* per queste operazioni.

- Esse sono a **basso livello**. Possono essere difficili da usare
- La libreria standard del C mette a disposizione delle **funzioni a più alto livello** (più facili da usare) che al loro interno **utilizzano le necessarie System Call** (quindi non c'è mai scampo da queste System Call!)
  - Ricordiamo che abbiamo **sempre** una situazione del tipo
  - **Hardware  $\leftrightarrow$  Sistema Operativo  $\leftrightarrow$  System Call  $\leftrightarrow$  Libreria  $\leftrightarrow$  Software**

**FIGURA** (*L'albero delle cartelle*)



## Link e cicli

### Problema (*i link*)

L'albero è l'organizzazione **più naturale**.

Tuttavia i **link** (1, 2) possono creare dei **cicli**

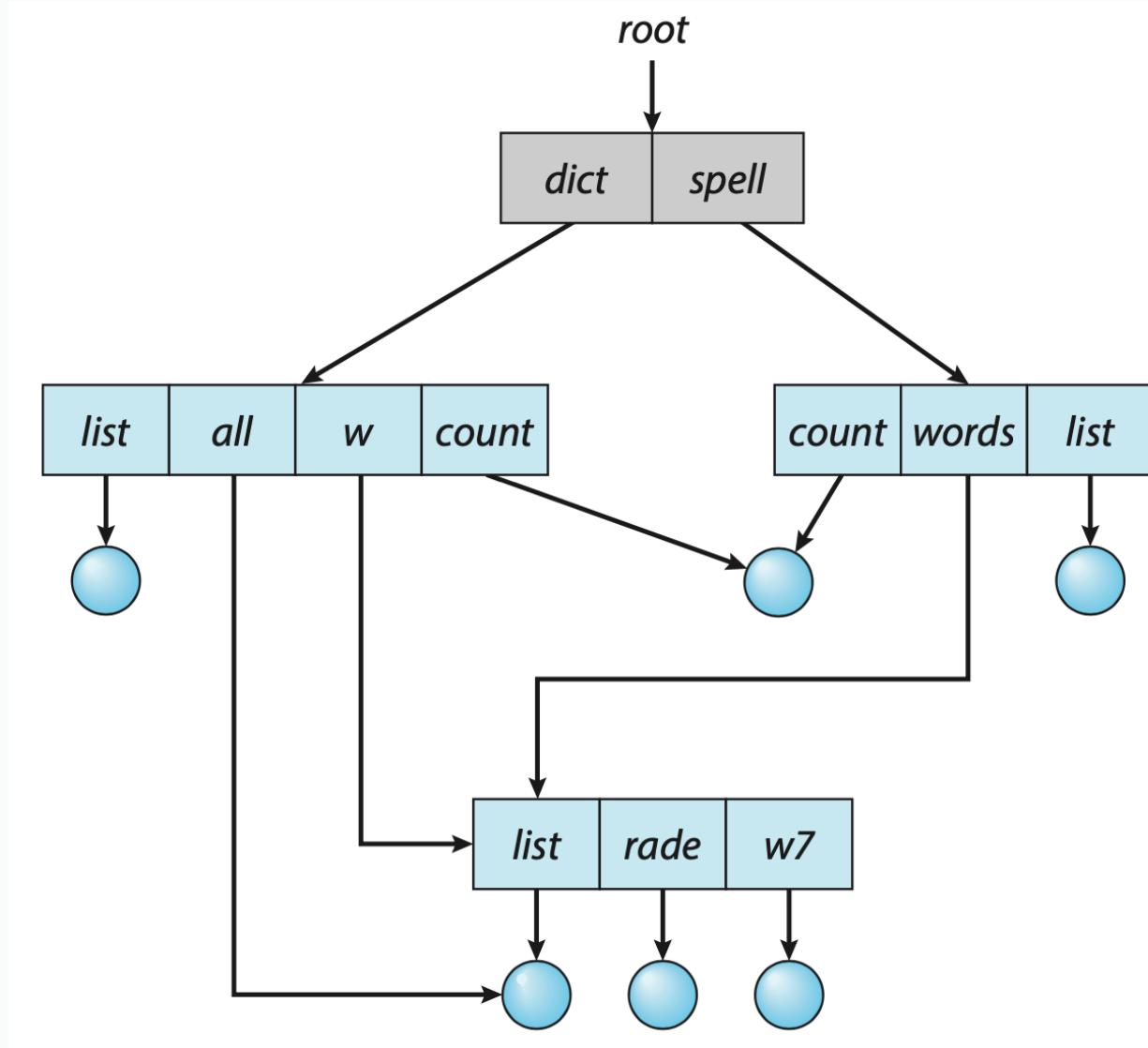
- Coi link, **cancellazione più complessa**
  - Sto cancellando il file originale o una copia?
  - Vedremo in fondo dopo

I cicli **complicano** molto la **gestione del File System**

- Immaginare un processo di **ricerca ricorsiva** in una cartella con cicli
  - Processo **potenzialmente infinito se non gestito correttamente!** (pericoloso)

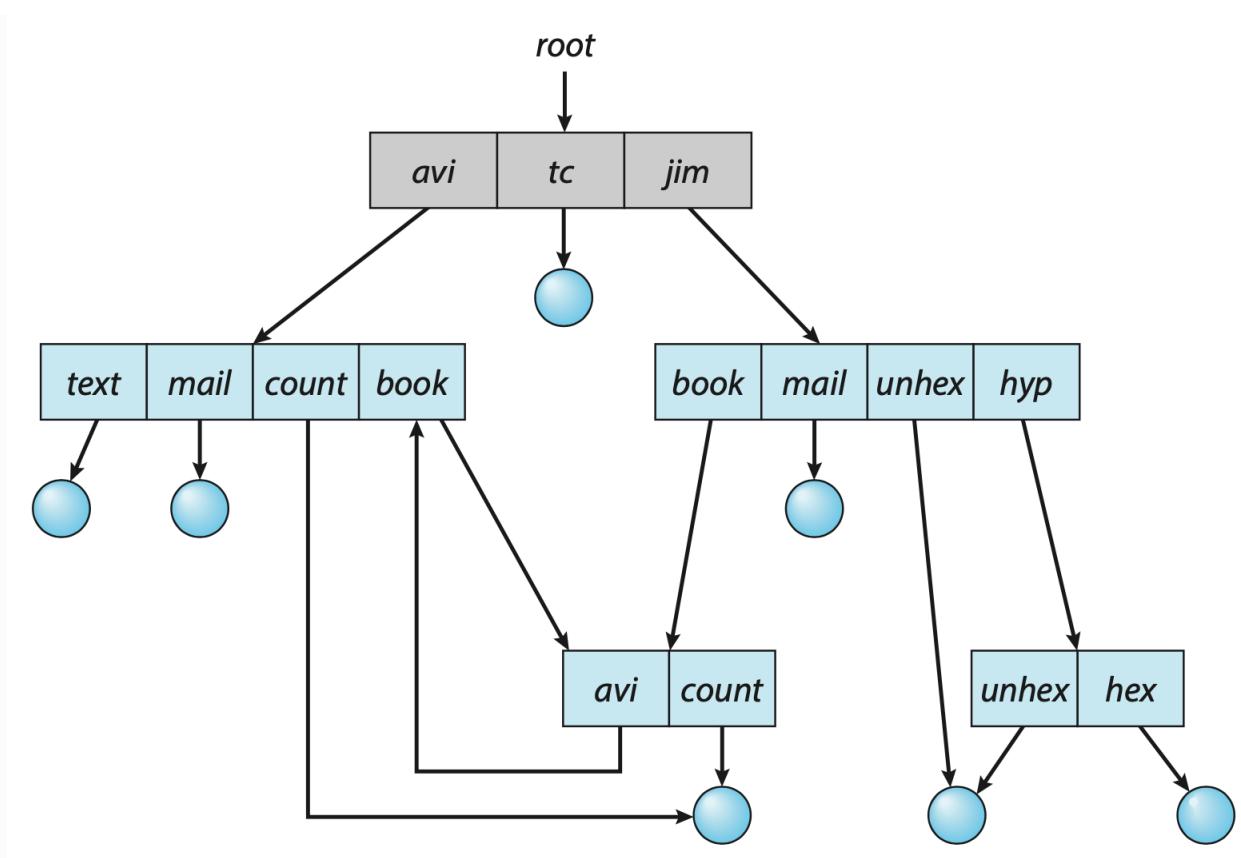
**Osservazione 1.** (*i link ai file vanno bene*)

I link a file non possono generare cicli. Infatti abbiamo una situazione del tipo



**Osservazione 2.** (*i link a direttori vanno gestiti*)

I link a direttori possono generare cicli:



*Possibili soluzioni:* mai visitare i link durante ricerca (questa è la soluzione comune); altrimenti avrei un *ciclo infinito*.

## Allocazione dei blocchi

**Richiamo.** (*il concetto dei blocchi*)

I file system risiedono su disco

I dischi permettono letture e scritture a *blocchi*

- Tipicamente da *512B a 8KB*
- E' possibile *leggere/scrivere un blocco per volta, e interamente*

Blocco 0	Blocco 1	Blocco 2	Blocco 3	Blocco 4
Blocco 5	Blocco 6	Blocco 7	Blocco 8	Blocco 9
.....	.....			

## Accesso ai Blocchi

Il *driver* (1) del disco permettono di accedere a un blocco.

- Ricevono comandi del tipo:

```
read block 431 to memory address 0x5984
write block 126 from memory address 0x9163
```

Un *File System* mappa *accessi a file e direttori in comandi per il driver*

- Quindi abbiamo una situazione del tipo
- APP → System Call → (OS) → Kernel → File System → Driver → (I/O) Disco

## Allocazione dei Blocchi: Definizione

### **Definizione** (*allocazione dei blocchi*)

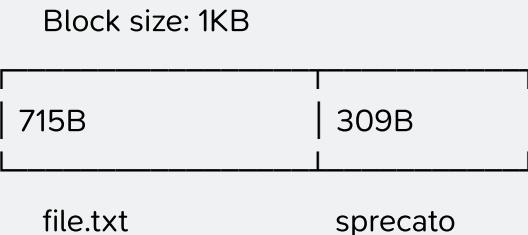
L'*allocazione* è il meccanismo con cui i blocchi sono allocati ai file.

- Ogni file *occupa 1 o più blocchi* ( $\geq 1$ )

### **Problema.** (*frammentazione interna*)

Come conseguenza del meccanismo dell'allocazione dei blocchi abbiamo un noto problema, detto come "*frammentazione interna*": in sostanza è lo spreco intrinseco di capacità quando un file non ha dimensione multipla della grandezza dei blocchi. La parte sprecata del disco non può essere *mai* usato per altri file

### **FIGURA.** (*Frammentazione interne*)



## Metodi per l'Allocazione dei Blocchi

Adesso iniziamo a vedere dei metodi per l'*allocazione dei blocchi*, quindi degli *algoritmi specifici di File System*.

## Allocazione Continua

### **Metodo.** (*Allocazione Continua*)

Ogni file *occupa un insieme di blocchi contigui* (ovvero una dopo l'altra)

### Vantaggi

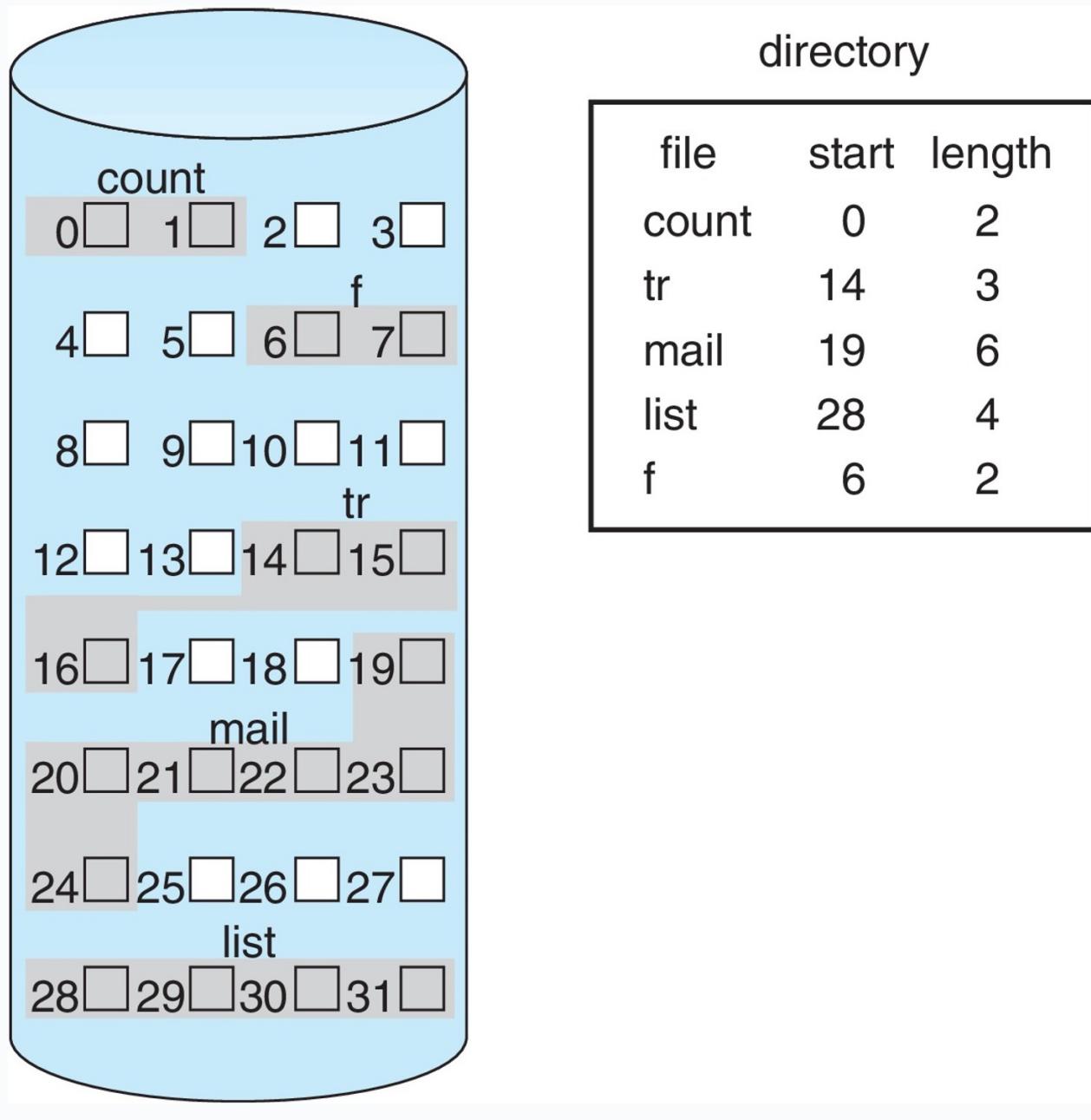
- *Semplice e veloce* (forse *troppo* semplice!)

- *Banale accedere* al byte  $N$ , visto che il file è memorizzato in *maniera contigua* sul disco
  - Ad esempio, per accedere ai byte *centrali* di un file basta andare al blocco *centrale*.
- *Pochi metadati* per file sono necessari
  - Solo *file*, *start* e *length*.

## Svantaggi

- Crea la *Frammentazione Esterna*: rimangono *blocchi vuoti sparsi* per il disco, che *non possono essere utilizzati* che per file molto piccoli.
- Grave spreco! I buchi diventano *sempre* più grandi col *tempo*.
  - Una *soluzione storica* era quella di fare la c.d. "*deframmentazione del disco*" (defrag); comunque per fare un'operazione del genere ci si può mettere un sacco di tempo.

**FIGURA.** (*Allocazione continua*)



## Allocazione Concatenata

### Modello. (*allocazione concatenata*)

Ogni File è una *Linked List* di blocchi.

- Ogni blocco contiene il *numero di blocco successivo*; l'equivalente del puntatore **NEXT**
- L'ultimo blocco contiene un *numero speciale che indica la fine*; l'equivalente del puntatore nullo **NULL**

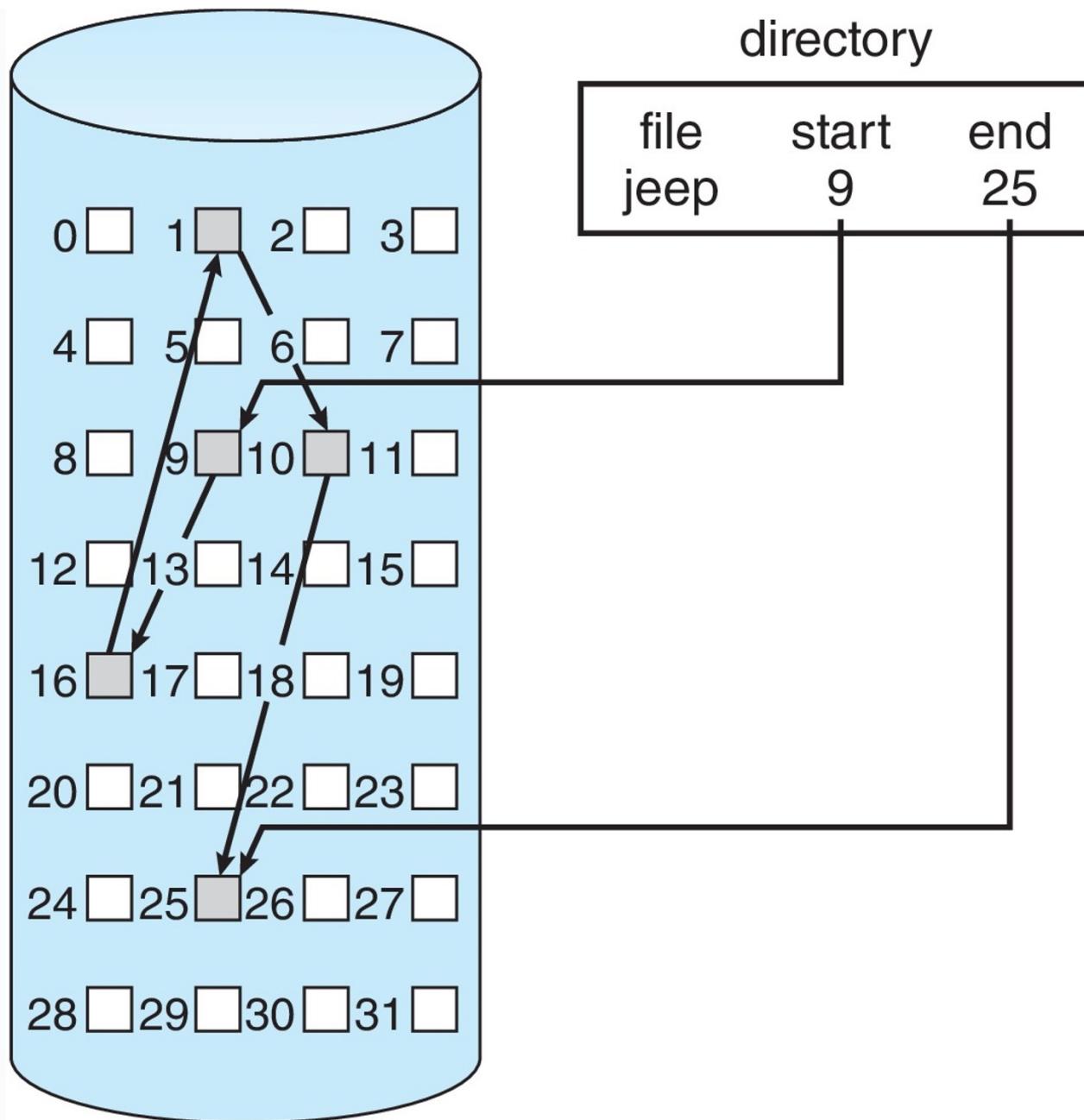
### Vantaggi:

- No *Frammentazione Esterna*!
- *Tutti i blocchi sono usabili* per ogni file
- Ancora meno metadati necessarie: bastano solo "*file*" e "*start*" (volendo anche "*end*")

### Svantaggi:

- Efficiente solo per *accesso sequenziale*
- Accedere ai byte *centrali* del file *richiede di scorrere tutta lista*, dalla *testa* del file.
  - Anche sono leggere il *puntatore* richiedere di leggere tutto il blocco
  - Ricorda: i dischi permettono di *leggere/scrivere un blocco per volta*
- *Poco affidabile!* Un blocco invalido invalida tutto il file
  - Problema per file grandi, che col tempo diventano sempre più soggetti a falle.
  - Sono come delle *candele* in serie: se si rompe una, il circuito non gira piùPer ovviare ad alcuni di questi problemi useremo una versione *ottimizzata* di questo modello. In particolare l'allocazione **FAT**.

### FIGURA. (*Allocazione concatenata*)



## Allocazione FAT (File Allocation Table)

### **Modello.** (*Allocazione FAT*)

Come detto prima, questo è una *variante* dell'*allocazione concatenata*.

I *primi blocchi* del disco contengono una *tabella della FAT* ("File Allocation Table")

- E' una *Linked List* di blocchi
- Approccio *simile a Allocazione concatenata*
  - Ma la lista contenuta nei primi blocchi
  - Più veloce, la FAT può essere in cache
- Usato in Windows e MS-DOS coi File System *FAT* e *FAT32*

### **Vantaggi**

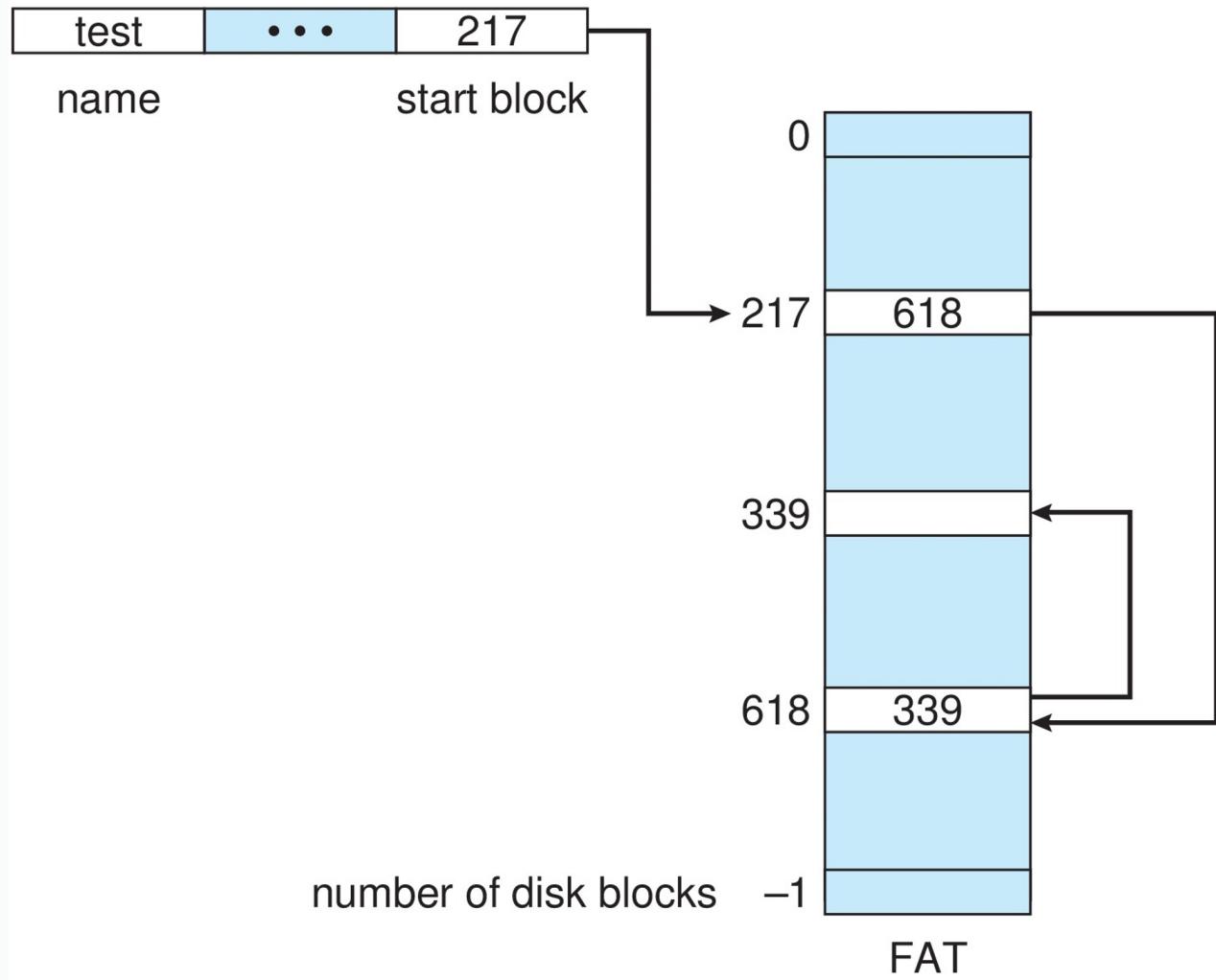
- La FAT è *cache-abile* (quindi ottimizzabile con pochi KB)
- Mi servono ancora meno metadati; basta aver salvato la *start block* della *FAT*.

## Svantaggi

- Lento accedere a ultimi byte del file (come allocazione concatenata)
- Se perdo la FAT perdo tutto!
- La *FAT* diventa grossa per dischi grandi (difficilmente *scalabile*)

## FIGURA. (*FAT*)

directory entry



## Allocazione Indicizzata

### Modello. (*Allocazione indicizzata*)

Ogni file ha un *blocco indice* che contiene i numeri di tutti i blocchi.

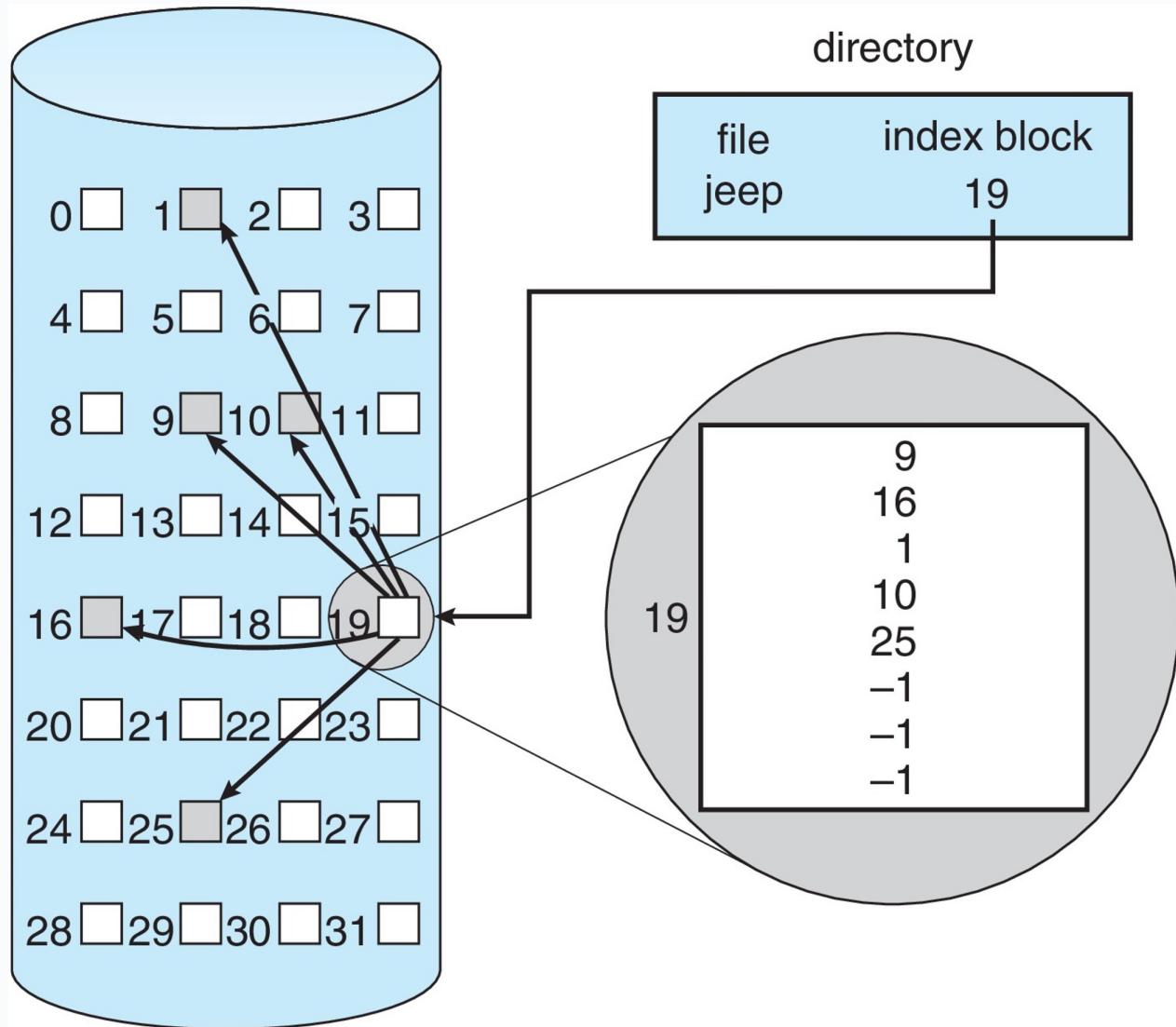
## Vantaggi

- Accedere a un *byte arbitrario* è veloce
  - Basta leggere il *blocco indice* ed il *blocco desiderato*

## Svantaggi

- Si sposta un blocco per file (oltre a quello del file)
- Se un file è troppo grande diventa impossibile

**FIGURA.** (Allocazione indicizzata)



## Allocazione Combinata

**Modello.** (Allocazione combinata)

Utilizzata in *Linux*; considerata il *migliore compromesso*

Ogni *file o directory* ha una *struttura* detta "*inode*", che contiene le seguenti informazioni.

- Metadati e permessi del file/direttorio
- I numeri dei *primi 12 blocchi*
  - Alcuni inutilizzati, se file più piccolo; vengono salvati direttamente nelle inode
- *Puntatori indiretti*:
  - Numeri di blocchi i quali contengono a loro volta *una tabella*
  - Su *uno, due e tre livelli* (quindi abbiamo *indici ai dati, indici agli indici ai dati e indici agli indici agli indici ai dati*)

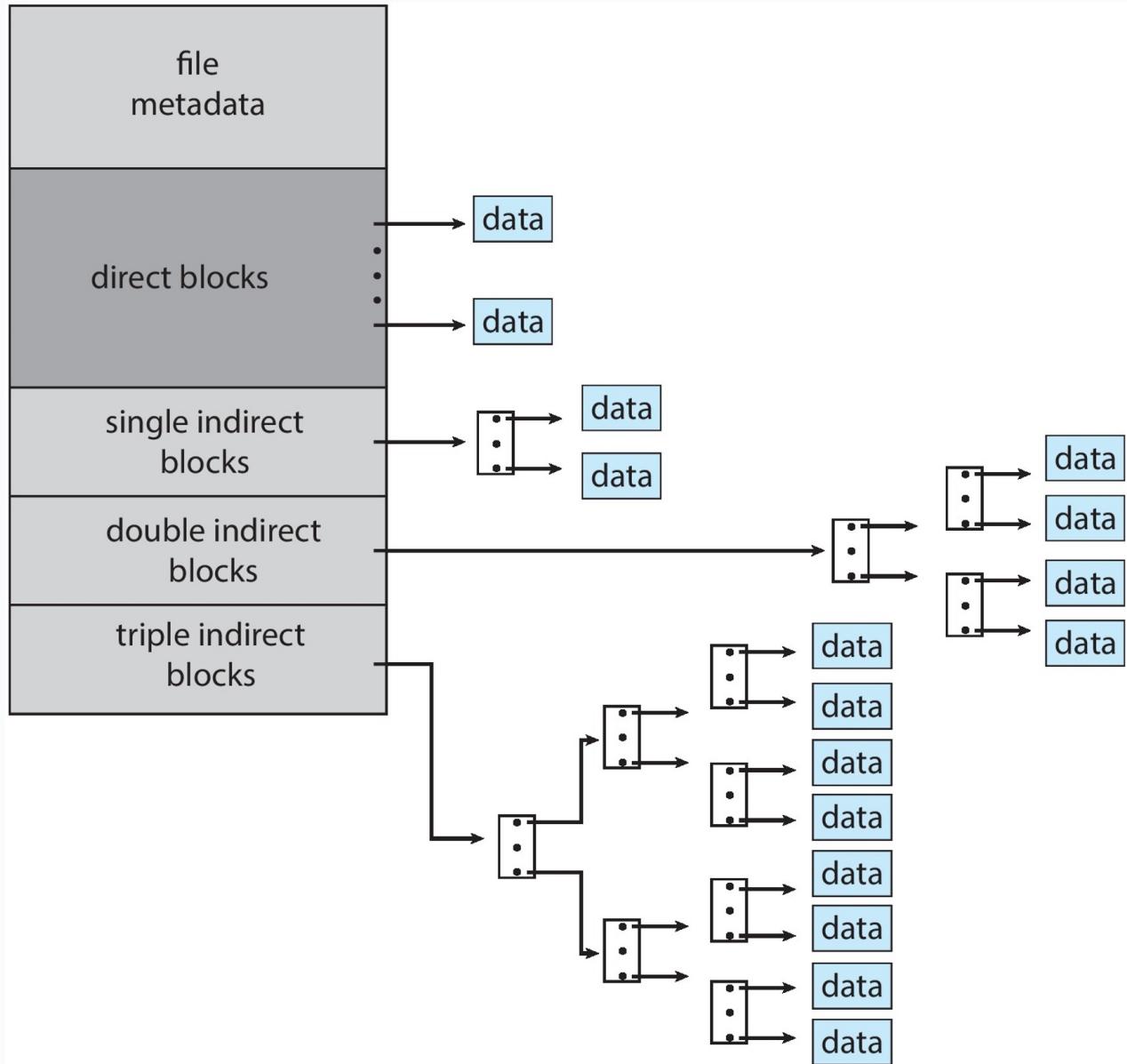
**Vantaggi:**

- Buon compromesso
- *No frammentazione esterna*
- Come *indexed* (indicizzata) per file piccoli
- Può *indicizzare file anche molto grandi*
- Per file di *medie dimensioni* ho l'*allocazione indicizzata* lo stesso

### Svantaggi:

- Può richiedere di leggere più di un blocco per accedere a posizioni avanzate nel file (quindi abbiamo tempo di lettura *non costante*)
- C'è una *grandezza massima* per i file, di circa *4TB*.

**FIGURA.** (*Allocazione Combinata con gli inode*)



## File system comuni

Adesso vediamo degli *algoritmi comuni* per i l'*allocazione dei blocchi*.

# Esempi di File System

## Esempi. (File System comuni)

Ogni OS si porta dietro i suoi File System

- **Unix:** UFS, FFS
- **Linux:** tantissimi.
  - **ext3** and **ext4** sono gli *standard di fatto*. Usano *allocazione Combinata*
- **Windows:**
  - FAT, FAT32 basati su FAT
  - NTFS: con indirizzamento ad albero
- **Apple:** HFS, HFS+, APFS (Hierarchical File System)
- **File System distribuiti per Big Data:** GoogleFS, HDFS, CEPH. Sono recenti e vengono utilizzati negli *ambienti professionali*. Li approfondiremo alla fine.

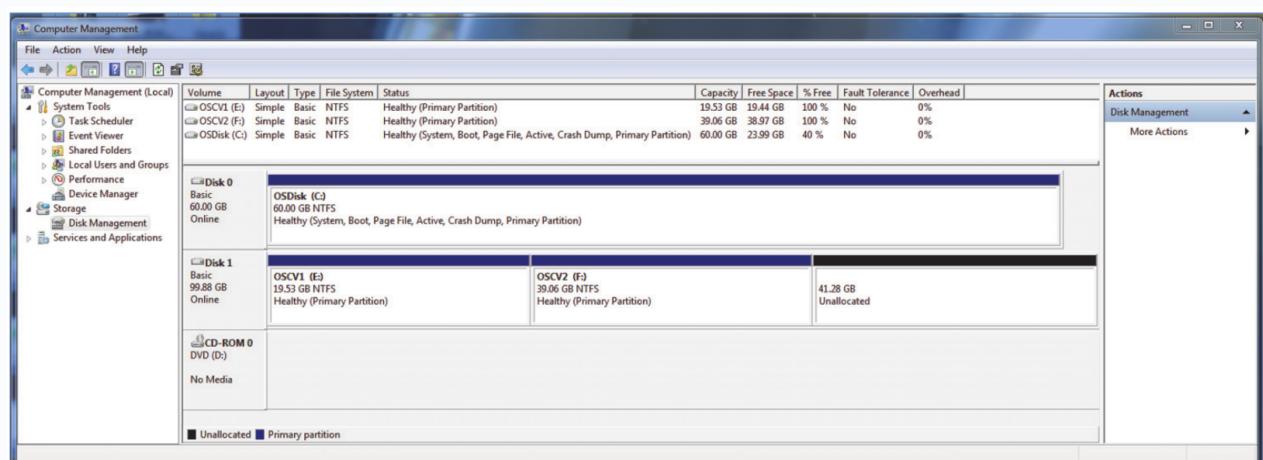
## Tabelle delle partizioni

### Definizione. (Partizione dei Blocchi)

Oltre ai FS, esistono degli standard per *partizionare i dischi in più partizioni* (quindi dare una *grandezza fissa* per i dischi).

- **Master boot record (MBR):** metodo classico. I primi blocchi del disco *indicano le partizioni*
  - Contiene anche il *codice iniziale* per avviare l'elaboratore
  - Massimo dischi da **2 TB e 4 partizioni** (limitata)
- **GUID Partition Table (GPT)** moderno, parte dello Unified Extensible Firmware Interface (**UEFI**) standard
  - Supera le limitazioni di MBR

### FIGURA. (Management delle partizioni su Windows)



### Esempio. (Linux)

Su Linux si opera da riga di comando con **fdisk** o **parted** o con l'utilità grafica **gparted**

## Domande

La CPU accede al disco:

- Direttamente
- Attraverso un'interfaccia
- Attraverso la memoria

### Attraverso la cache

Risposta: Attraverso un'interfaccia

I link a file possono generare cicli:

- Vero
- Falso

Risposta: Falso

I link a directory possono generare cicli:

- Vero
- Falso

Risposta: Vero

Un disco ha blocchi grandi 4KB (4096B). Un file di grandezza 510B. Quanto spazio viene sprecato a causa della frammentazione **interna**:

- 4606B
- 3586B
- Impossibile da stabilire

Risposta: 3586B

Un disco ha blocchi grandi 4KB (4096B). Un file di grandezza 510B. Quanto spazio viene sprecato a causa della frammentazione **esterna**:

- 4606B
- 3586B
- Impossibile da stabilire

Risposta: Impossibile da stabilire

Con l'allocazione concatenata si ovvia al problema della:

- Frammentazione interna
- Frammentazione esterna

Risposta: Frammentazione esterna

In Linux, il FS usa lo schema:

- FAT
- Allocazione concatenata
- Allocazione combinata
- Allocazione continua

Risposta: Allocazione combinata

Quali tra questi é un formato per le tabelle delle partizioni:

- Ext
- MBR
- FAT
- NTFS

Risposta: MBR

## u4-s2-file-system-linux

### Sistemi Operativi

#### Unità 4: Il File System

## I file system in Linux

## Argomenti

1. File System in Linux
  2. Permessi in Linux
  3. Comandi Bash per i dischi
- 

## File System in Linux

---

### Richiami Storici

#### **Richiamo.** (*Storia del Linux*)

Nel mondo Unix/Linux, esistono molti file system.

- Il primo è stato lo Unix File System (UFS)
- Da esso si sono gli Extended File System (**ext**) per Linux
- Ora siamo alla versione **ext4**  
Come visto, basato sul *concetto di inode* (1) che rappresenta *un file o un directory*

## Virtual File System (VFS)

#### **Definizione.** (*Virtual File System*)

Si possono usare *diversi file system*.

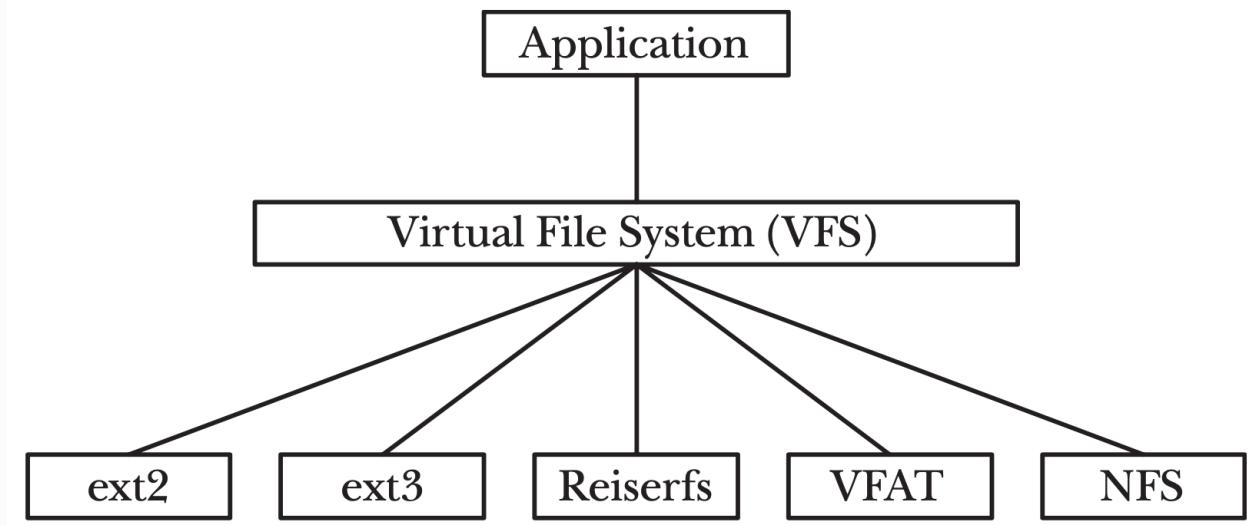
Devono implementare l'interfaccia standard *Virtual File System (VFS)*

- Ovvero permettano di effettuare alcune *funzioni fondamentali* e di rendere *generico* il kernel Linux:

C  
`open(), read(), write(), lseek(), close(), truncate(), stat(),  
mount(), umount(), mmap(), mkdir(), link(), unlink(), symlink(), rename()`

- Quindi si ha una situazione del tipo *Applicazione → VFS → File System → ... → Hardware*

#### **FIGURA.** (*L'idea del VFS*)



## Montare File su Linux

**Richiamo.** (*Cartella root*) & **Definizione.** (*Operazione di montaggio dischi*)

Su Linux, tutti i file da ogni disco sono sotto un unico albero di cartelle, detto "root" (1)

- Che nasce da 1; tutto è figlio di questa cartella
- Dischi aggiuntivi vengono *montati* come sotto alberi di 1

**Operazione.** (*Montare dischi*)

Per *montare* un *FS* si usa il comando

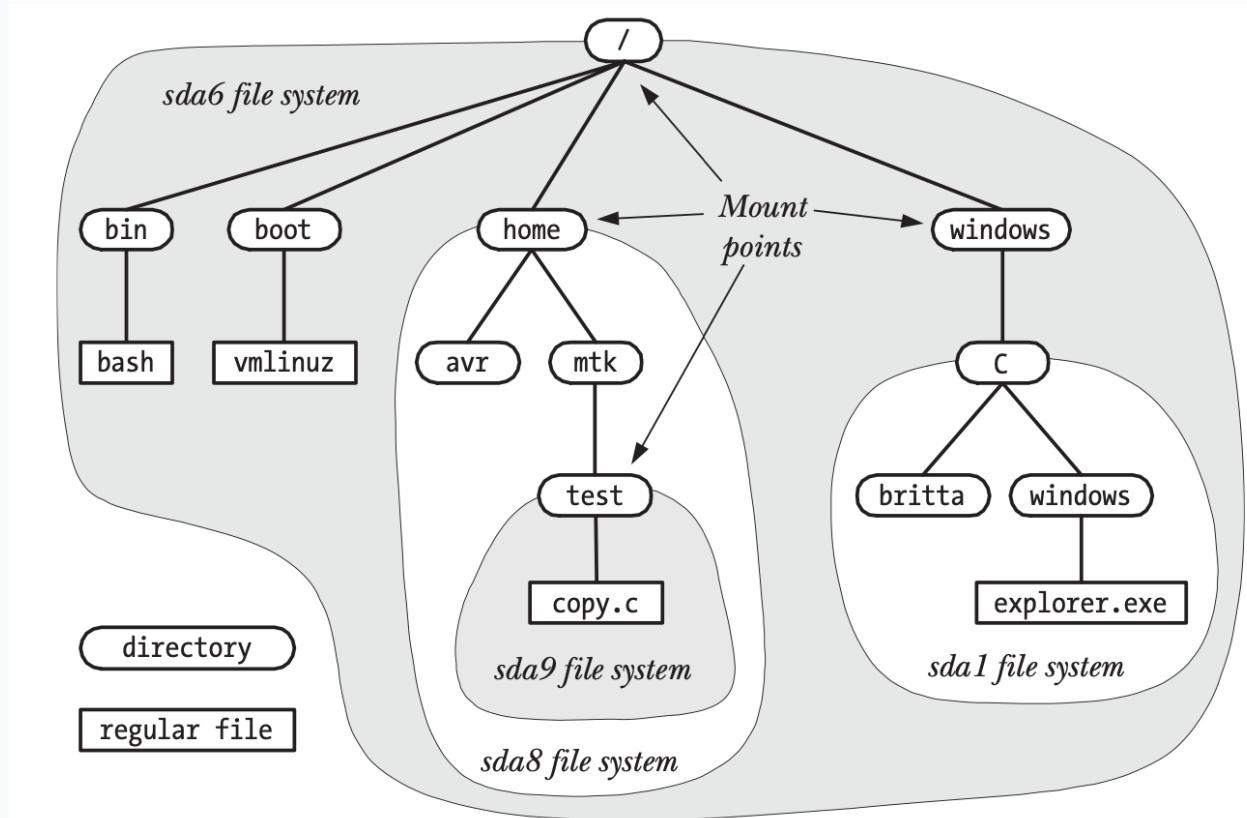
mount device directory

**Operazione.** (*Vedere i file montati*)

Per vedere il contenuto di un disco, esso va *montato*. Per vedere i dischi montati si usa lo stesso comando, omettendo tutti gli argomenti.

```
$ mount
/dev/sda6 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
/dev/sda8 on /home type ext3 (rw,acl,user_xattr)
/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 on /home/mtk/test type reiserfs (rw)
```

**FIGURA.** (Esempio di una gerarchia di FS montati)



### Definizione. (File di configurazione **fstab**)

In un sistema Linux, i *dischi che vengono montati automaticamente all'avvio* sono specificati nel file **/etc/fstab**

- Contiene una riga per ogni disco
- Formato **<disk> <mount point> <type> <options> <dump> <pass>**
  - Ovvero "disco X" sulla "cartella Y" del "tipo Z" con certe cose...
- Fondamentale saperlo: se questo file ha certi casini, l'elaboratore non sarà più in grado di partire!

### Esempio:

SHELL

```

/dev/sda1 /          ext4  errors=remount-ro      0  1
/dev/hda1 /media/hda1  vfat   defaults,utf8,umask=007,gid=46  0  0
  
```

## Inode, Partizione e Blocco Dato

Nota: Si trovano in sezione A - OK

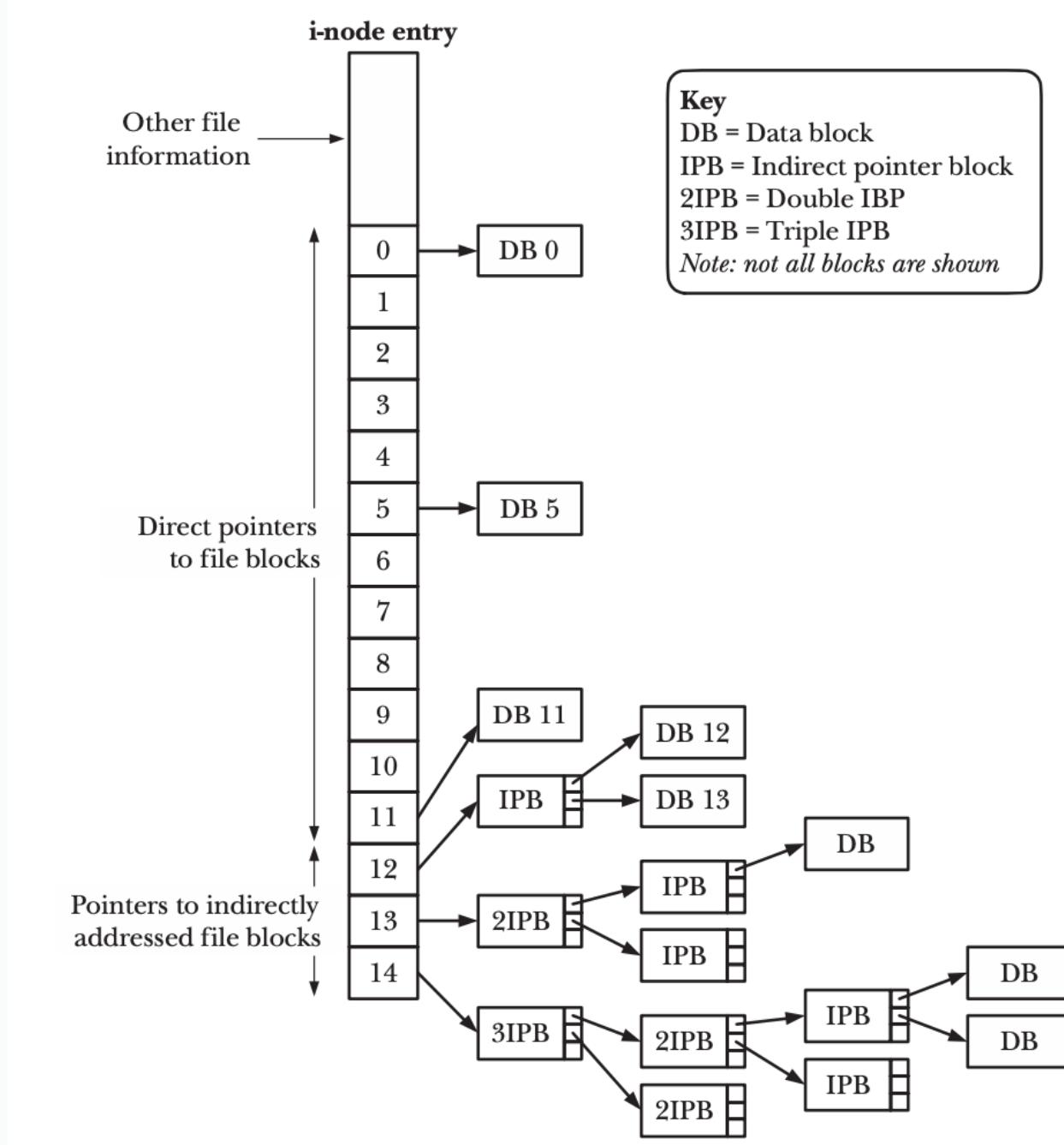
### Inode

#### Definizione. (Inode)

Rappresentano un *file/cartella* (1). Memorizzati in una tabella nei primi blocchi

- Ogni inode è una struttura di pochi byte (ha una dimensione fissa)
- Identificati da *inode number*
  - **ATTENZIONE!** Ciò vuol dire che l'inode *non* contiene il *nome del file*! Per trovare ciò bisogna reperirlo con altre funzioni.
- Sono *in numero finito e immutabile*
  - Non si possono memorizzare infiniti file minuscoli

**FIGURA.** (*Struttura dell'i-node*)



## Layout di un disco, definizione di partizione

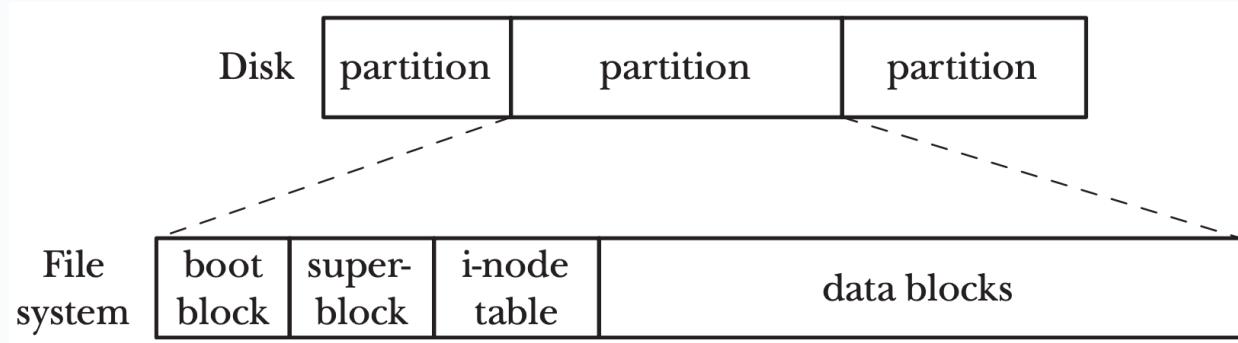
**Definizione.** (*Partizione di un disco*)

Un disco è diviso in *partizione* (1)

Ogni partizione contiene

- Informazioni di controllo; tra cui informazioni per far *partire* il *sistema operativo*, dei *metadati* e così via...
- *Tabella degli inode*
- Blocchi di dato (ovvero quelli in cui mettiamo i *dati*)

**FIGURA.** (*L'idea della partizione*)



## Tipi di Blocco Dato

**Definizione.** (*Data block, Directory block*)

I blocchi di dato sono di due tipi:

- *Data Block*: hanno il contenuto di un file. *Dati binari*
- *Directory Block*: hanno il contenuto di una cartella. *Lista di coppie (nome, inode)*

**FIGURA.** (*Esempio generale di un disco*)



## File System in Linux (ulteriori dettagli)

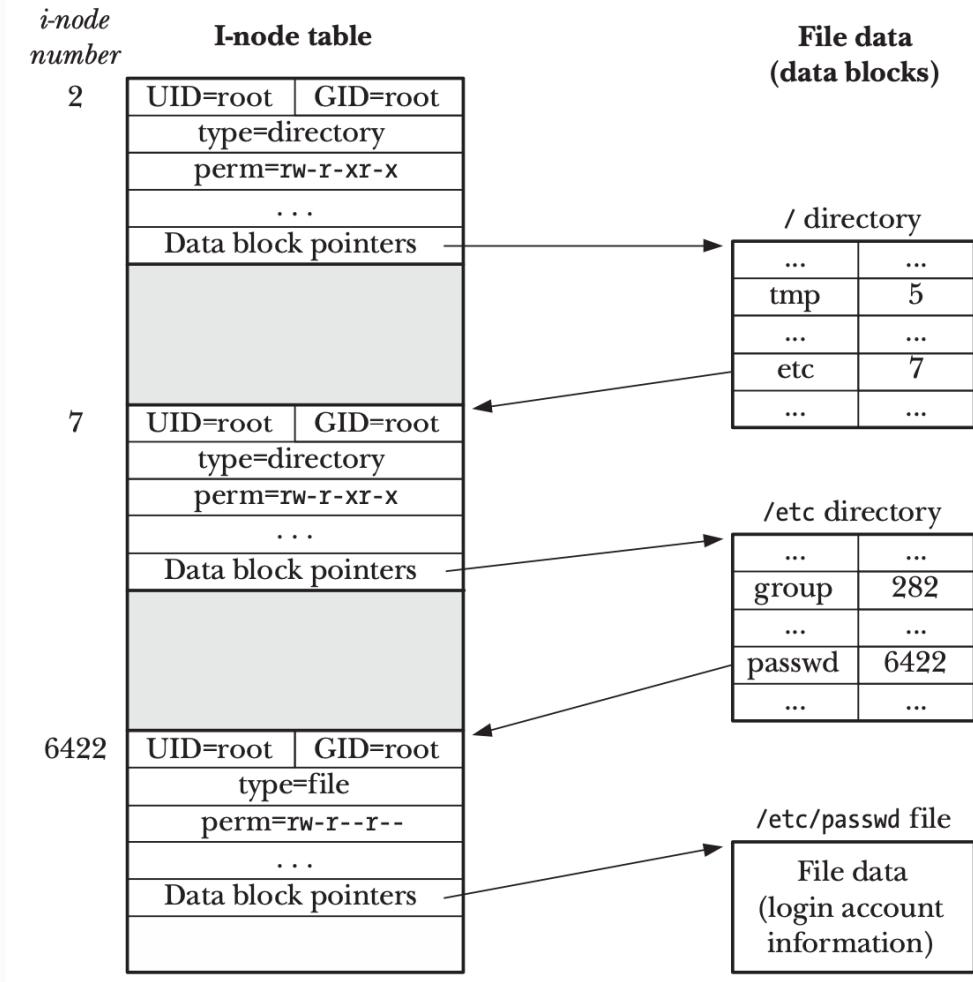
### Definizione di Directory per Linux

**Corollario.** (*Definizione del directory su Linux*)

Ogni directory é un inode (1).

- Occupa almeno un blocco contenente la tabella dei nodi che contiene Pertanto possiamo considerarla *come un file "speciale"*:
- Il cui contenuto non é un insieme di byte
- Ma una *lista di coppie (nome, inode)*

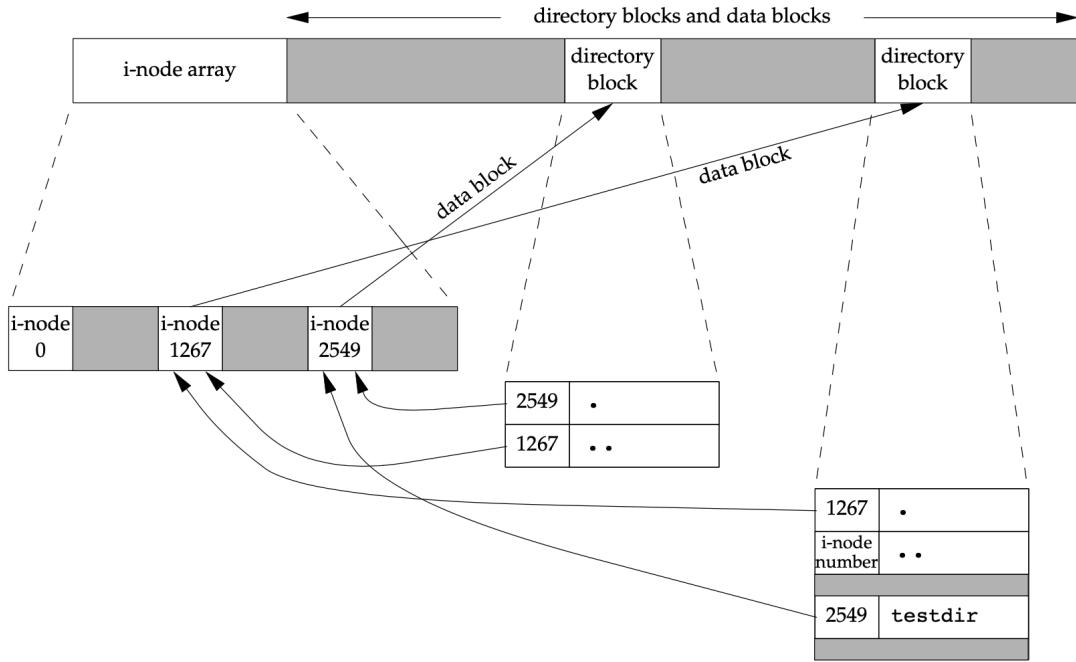
**Esempio.**



Dalla figura vediamo che

- inode 2 é la directory **/**
  - Contiene **etc**: inode 7
- inode 7 é la directory **/etc**
  - Contiene **passwd**: inode 6442
- inode 6442 é il file **/etc/passwd**
  - Il contenuto é in un blocco dati

**Esempio.**



- La cartella **d** (inode 1267) contiene la cartella **testdir** 2549 che é vuota

## Permessi in Linux (ulteriori dettagli)

### Utenti e Gruppi

In parte visto all'inizio del corso ([Utenti e Permessi Linux](#))

In Linux, esistono:

- **Utenti:** account che possono utilizzare il sistema, creare processi, accedere a file
- **Gruppi:** insiemi di utenti. Ogni utente ha un:
  - **Gruppo principale:** solo uno (= 1)
  - **Gruppi secondari:** senza limiti di numero ( $\geq 0$ )

Ogni utente e gruppo è identificato da un *nome* e da un *id* numerico

#### Esempio:

- **Utenti:** **martino**, **luca**, **paolo**
- **Gruppo principale:**
  - **martino** → **docenti**
  - **luca** → **studenti**
  - **paolo** → **studenti**
- **Gruppo secondario:**
  - **martino luca** → **sistemioperativi**
  - **martino paolo** → **reti**

# Utente root

## Definizione. (*Utente root*)

L'utente *root* esiste su tutti i sistemi (1)

- Ha *id* 0
- Bypassa tutti i controlli sui permessi (ovvero fa ciò che vuole, nei limiti del software)

*Nota:* *root* è un utente con privilegi illimitati. NON è né parte del kernel, né il suo codice esegue in modalità kernel.

*Errore comune:* dire che l'utente *root* esegue processi in kernel-mode! Non può mica accedere alla *memoria* o ai *dispositivi I/O*!

# File per utenti e gruppi

## File di configurazione per utenti e gruppi.

Le informazioni su utenti e gruppi attivi salvate in file di configurazione accessibili *solo a root*:

- **/etc/passwd**: lista di utenti e dettagli (*ID*, *home directory*)
  - Nota: Una volta c'erano le *password* dentro, adesso non più per ovvi motivi di sicurezza (nonostante il nome ci dia un'idea del genere).
- **/etc/shadow**: password cifrate con *hash*.
- **/etc/group**: lista di *gruppi* e dei rispettivi componenti

---

# Permessi per i File o Cartelle

## Definizione. (*Utente e gruppo proprietario*)

Ogni file/cartella ha uno e uno solo utente *proprietario*, e uno e uno solo *gruppo proprietario* (1).

E' possibile separare i permessi per classi di utenti.

Ovvero si specificano permessi separatamente per:

- **Utente proprietario**: si applica quando il proprietario tenta di fare accesso
- **Gruppo proprietario**: si applica quando un utente del gruppo proprietario accede
  - Nota: il gruppo proprietario non è necessariamente il **Gruppo principale** del proprietario, ma può essere un **Gruppo secondario**
- **Altri**: tutti gli altri utenti

# Permessi di base

## Definizione. (*Permessi base*)

Richiamiamo i *permessi base* (1, 2, 3), tenendo conto del fatto che le *cartelle* non sono

altro che dei *file contenenti* delle tabelle di i-node.

- **Lettura:** per i file, *leggere il contenuto*. Per le cartelle, elencare *nodi contenuti*
- **Scrittura:** per i file, *scrivere il contenuto*. Per le cartelle, *aggiungere/rimuovere nodi contenuti*
- **Esecuzione:** per i file, *eseguirli*. Per le cartelle, attraversarle, ovvero *accedere a file (o sottocartelle) contenuti*.
  - Nota: Per le cartelle, attraversare è diverso da listare (permesso lettura)!

**Nota:** non sono i permessi su un file a determinare se esso può essere *cancellato*, ma sono i *permessi sulla directory* che lo contiene a farlo.

## Permessi speciali

### Definizione. (*Permessi speciali*)

Oltre i 3 permessi di base, esistono altri tre permessi speciali (o flag) che si possono applicare a file/cartelle. Essi sono "*set user ID (suid)*", "*set group ID (guid)*" e lo "*sticky bit*".

- **set user ID (suid):** per i *file*, se eseguito, il processo è eseguito coi privilegi di *utente proprietario*, non di esecutore. Per le *cartelle, non ha effetto*.
  - **Utilizzo:** sui PC, i comandi di sistema (e.g., **reboot**) hanno il **suid**, per permettere riavvio senza chiedere password. Sui server, di solito no!
- **set group ID (guid):** per i *file*, se eseguito, il processo è eseguito coi privilegi di *gruppo proprietario*, non di esecutore; questo di solito è *inutile*. Per le *cartelle*, i *file creati hanno il gruppo della cartella* e non il gruppo principale del creatore (che è azione di default); questo è l'uso *principale* del guid.
  - **Utilizzo:** quando si creano *cartelle condivise* tra utenti che appartengono a un gruppo creato ad hoc.
- **sticky bit:** per i *file*, non ha *più effetto*. Per le *cartelle*, i file in essa contenuti possono essere *cancellati e spostati solamente dagli utenti che ne sono proprietari*, o dall'*utente proprietario della cartella*.
  - **Utilizzo:** nelle cartelle **/tmp** e **/var/tmp** tutti gli utenti devono poter creare e modificare dei file. Nessuno eccetto il *superuser* (o il proprietario) deve poter rimuovere o spostare file temporanei di altri utent

**Esempio-Esercizio.** Per un progetto si crea il gruppo **progettoSysOp**, che contiene 3 utenti. Si crea la cartella condivisa **/share/progetto** e la si assegna al gruppo **progettoSysOp**

## Esempi dei Permessi Speciali

**SUID.** Abbiamo il comando **reboot** in **\bin\reboot** e il suo codice sorgente è qualcosa del tipo

```
main()
{
    ...
    reboot();
    ...
}
```

dove **reboot()**; è una *System Call*, che può essere eseguita *soltanto* da root. Vediamo il comportamento *default* e il comportamento col *suid*.

- *Default*

SHELL

```
DINO:$\bin\reboot # → rwx r-x r-x
> Fallisce!
```

- *Con suid*

SHELL

```
PAOLO:$\bin\reboot # → swx r-x r-x
> Ok! Il PC si riavvia
```

**GUID.** Il nostro computer contiene i seguenti utenti e gruppi:

- *Utenti*: MARTINO, LUCA
- *Gruppi*: DOCENTI, SISOP, STUDENTI

Denotiamo " $\Rightarrow$ " per "X ha il gruppo primario Y" e " $\rightarrow$ " per "X appartiene al gruppo secondario Y".

Abbiamo le appartenenze ai gruppi come:

- MARTINO  $\Rightarrow$  DOCENTI, MARTINO  $\rightarrow$  SISOP, LUCA  $\Rightarrow$  STUDENTI

Voglio creare una cartella comune chiamata **/SHARED**.

Dentro questa cartella il prof. MARTINO ci inserisce una sottocartella

**/SHARED/PROGETTO**, con appartenenza **MARTINO:SISOP**.

Adesso vediamo i comportamenti di questa sottocartella.

- *Default* (caso **rwx rwx ---**)

SHELL

MARTINO:\$ `mkdir /SHARED/PROGETTO/DATI`  
> OK! File creato con permessi rwx rwx --- appartenente a MARTINO:DOCENTI  
LUCA:\$ ... # ci fa qualcosa in nella cartella ./DATI  
> No! Non ha i permessi sufficienti  
> Per ovviare a questo bisogna usare il comando chgrp, che è molto scomodo da fare. Quindi modiflico il comportamento default come segue

- Con *GUID* (caso **rwx rws ---**)

SHELL

MARTINO:\$ `mkdir /SHARED/PROGETTO/DATI`  
> OK! File creato con permessi rwx rwx --- appartenente a MARTINO:SISOP  
LUCA:\$ ... # ci fa qualcosa in nella cartella ./DATI  
> Ok! Va bene

**STICKY-BIT.** Prendiamo la cartella **/tmp**

- Default (**rwx rwx rwx**)

SHELL

MARTINO:\$ `touch /TMP/F1 # → MARTINO:DOCENTI; RWX --- ---`  
LUCA:\$ `rm /TMP/F1`  
> Comando eseguito con successo! Ma non dovrebbe essere così. Ognuno può fare ciò che vuole con i **file** degli altri.

- Con *sticky bit* (**rwT rwx rwx**)

SHELL

MARTINO:\$ `touch /TMP/F1 # → MARTINO:DOCENTI; RWX --- ---`  
LUCA:\$ `rm /TMP/F1`  
> Eh no! Non può, cicce.

## Permessi dei Link

### Caso Symlink (caso simbolico).

Non hanno permessi propri, ma *ereditano i permessi del file/cartella linkato*.

**Nota:** la loro creazione/distruzione resa possibile dai permessi della cartella in cui si trovano

Esempio:

SHELL

```
$ ls /lib  
...  
lrwxrwxrwx 1 root root 16 feb 24 2020 sendmail → ../sbin/sendmail*
```

...

### Caso Hard-link.

La modifica dei permessi su un hard link, affligge anche il file originario; infatti con un *hard link* stiamo creando un *file* che punta allo *stesso* ed *identico* inode (1).

Motivazione:

- Un hard link non è altro che un riferimento allo stesso *inode* con un nome differente
  - e/o in una cartella differente
- File originario e suo hard link hanno la *stessa importanza*
  - Il file originale è *indistinguibile da un suo hardlink*
- I permessi di un file sono memorizzati *nel suo inode*

## Rappresentazione dei Permessi in Linux

Esistono due notazioni per indicare i permessi di un file/cartella in Linux.

### Rappresentazione simbolica.

Usata da **ls -l** e la più diffusa

Il primo carattere indica il tipo di file o directory elencata, e non rappresenta propriamente un permesso:

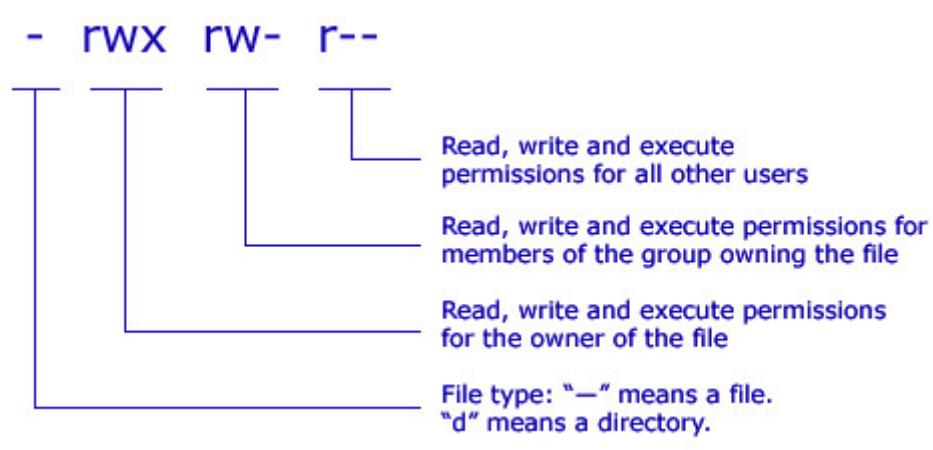
- **-**: file regolare
- **d**: directory
- **b**: dispositivo a blocchi
- **c**: dispositivo a caratteri
- **l**: collegamento simbolico
- **p**: named pipe
- **s**: socket in dominio Unix

Dopodiché abbiamo altri *9 caratteri*, che rappresentano i *permessi di lettura, scrittura ed esecuzione* per l'*utente proprietario*, il *gruppo proprietario* e *tutti gli altri*.

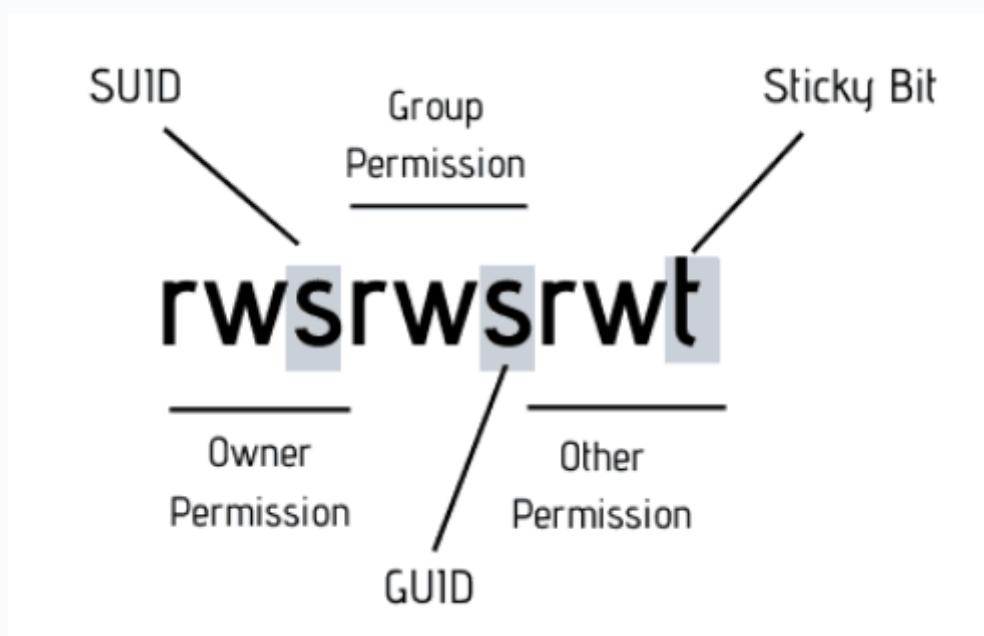
Inoltre permessi speciali vengono *aggiunti* a questa notazione sostituendo alcune lettere degli ultimi nove caratteri, come segue.

- **s** al posto del primo **x** per il SUID
- **s** al posto del secondo **x** per il GID
- **t** al posto del terzo **x** per lo sticky bit

**FIGURA.** (Schema generale della rappresentazione simbolica)



**FIGURA.** (Schema di rappresentazione simbolica per permessi speciali)



### Rappresentazione Ottale.

La stessa informazione può essere rappresentata con 4 cifre in base 8. Dove:

- La prima cifra rappresenta i permessi speciali
  - Le altre tre cifre rappresentano i permessi per l'utente proprietario, il gruppo proprietario e tutti gli altri.
- La prima cifra è quasi sempre 0 ed è *omessa*

**FIGURA.** (Schema della rappresentazione ottale)

Octal:	0	6	4	0
Binary:	000	110	100	000
Symbolic:	s s t	r w x	r w x	r w x
	Special attributes	User (u)	Group (g)	Other (o)
All (a)				

## Esempio

750 equivale a **rwx r-x ---**

- $7 = 4 + 2 + 1$  mentre  $5 = 4 + 1$   
644 equivale a **rw- r-- r--**
- $6 = 4 + 2$

## Trucchetto.

Possiamo ricordarci la tabella di conversione dall'ottale al binario che è

- **0  $\Leftrightarrow$  000**
  - **1  $\Leftrightarrow$  001**
  - **2  $\Leftrightarrow$  010**
  - **3  $\Leftrightarrow$  011**
  - **4  $\Leftrightarrow$  100**
  - **5  $\Leftrightarrow$  101**
  - **6  $\Leftrightarrow$  110**
  - **7  $\Leftrightarrow$  111**
- 

## Comandi Bash per i dischi

### Utility GNU.

I SO Linux/Posix hanno dei programmi *pre-installati* per gestire i file (1)

- Sono delle *utility*, parte della *GNU*, che permettono di svolgere compiti semplici e ripetitivi da riga di comando
- Senza dover scrivere un programma apposito che chiami le *System Call* o *Funzioni di Libreria necessarie*.
- Documentati nella sezione 1 di **man** (*User Commands*) e nella sezione 8 (*System Administration tools and Daemons*)

## Comandi Bash per i dischi

Enchiammo alcuni comandi importanti.

- **df**: visualizza dischi e loro occupazione ("disk free")
- **mount**: permette di:
  - vedere quali dischi sono in uso
  - *montare* un disco, ovvero agganciarlo all'albero di file della macchina
  - usa la System Call **mount**
- **fdisk**: visualizza dischi e partizioni e crea partizioni
- **lsblk**: visualizza in maniera semplice le partizioni e i dischi ("list block devices")
- **mkfs**: formatta e inizializza un File System su un disco ("make file system"\*)

- **lspci** e **lsusb**: lista dispositivi PCI e USB, tra cui dischi ("list PCI/USB")
- 

## Domande

Un inode può rappresentare:

- File
- Cartelle
- File o cartelle
- Link
- Partizioni

Risposta: "File o cartelle"

Il comando **mount** serve a:

- Montare dischi
- Formattare dischi
- Manipolare directory

Risposta: "Montare dischi"

Gli inode possono essere memorizzati:

- In qualsiasi posizione del disco
- All'inizio
- Alla fine

Risposta: "All'inizio"

Gli elementi di una cartella sono memorizzati:

- All'interno del suo inode
- In un blocco dati separato

Risposta: "In un blocco dati separato"

In Linux, ogni utente può appartenere a un solo gruppo:

- Vero
- Falso

Risposta: "Falso" (domanda tricky!)

L'utente root esegue i suoi processi in kernel-mode?

- Si
- No

Risposta: "No"

Il permesso di esecuzione sulle directory:

- Non ha effetto
- Permette di eseguire i programmi contenuti
- Permette di attraversare la cartella

Risposta: "Permette di attraversare la cartella"

La cartella **d** contiene un file **f**. L'utente **u** ha permessi sulla cartella **d r-x** e sul file **f rw-**. L'utente può rimuovere il file?

- Si
- No

Risposta: "No"

La cartella **d** contiene un file **f**. L'utente **u** ha permessi sulla cartella **d r-x** e sul file **f rw-**. L'utente può eseguire il file?

- Si
- No

Risposta: "No"

L'utente **u** appartiene ai gruppi **g1** e **g2**. Una cartella contiene i seguenti file.

```
-rw-r--r-- 1 u g3 2577901 Jul 28 2013 f1.txt  
-r--r--r-- 1 v g1 5634545 Jul 13 2013 f2.txt  
-rwxrwxrwx 1 z g4 8753244 Jul 29 2013 f3.txt
```

Su quali di questi file **u** ha permesso di scrittura?

- Tutti
- Solo f1
- Solo f3
- f1 e f3

Risposta: "f1 e f3"

## u4-s3-file

### Sistemi Operativi

#### Unità 4: Il File System

##### I file

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

##### Argomenti

1. Funzioni di libreria per file
  2. System Call per file
  3. Funzioni di libreria vs System Call
  4. Comandi Bash per File
- 

## Funzioni di libreria per file

### Funzioni di libreria per file: Recap

Abbiamo già visto le funzioni di libreria più comuni per leggere su file.  
Esse sono parte della *libreria standard* **libc**.

- Sono utilizzabili su *qualsiasi SO* (portabile)
- E' sufficiente ricompilare il programma
- Utilizzano *System Call diverse a seconda del SO*

Funzioni principali:

- Puntatore a file: **FILE \***
- Apertura/chiusura: **fopen**, **fclose**
- Lettura/scrittura di caratteri: **fgetc**, **fputc**
- Lettura/scrittura di righe: **fgets**, **fputs**
- Lettura/scrittura con formato: **fscanf**, **fgets**
- Lettura/scrittura grezza **fread**, **fwrite**
- Riposizionamento: **fseek**, **rewind**, **ftell**

**Esempio:** si legga un path da tastiera e se ne stampi il contenuto come file di testo

```
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Inserisci un path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossibile aprire %s\n", s);
        return 1; /* Errore */
    }

    /* Non è importante la lunghezza di buffer */
    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout); // Equivale a printf("%s", s);

    fclose(f);
    return 0;
}
```

## Funzioni per manipolare il Cursore

**Nota.** (*sequenzialità*)

La lettura/scrittura su file è *sequenziale*.

Un file aperto ha un *cursore* simile a quello di un editor testuale

- Determina la posizione di partenza per la prossima operazione
- Esso è posizionato a un *offset* preciso all'interno del file

- Una lettura sposta in avanti il cursore dei caratteri letti
- Una scrittura inserisce i caratteri nella posizione del cursore

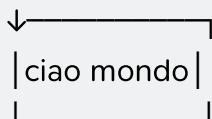
Esistono delle funzioni per spostare arbitrariamente il cursore, come viste prima.

### Esempio.

Consideriamo un file 10 caratteri contenente la frase **ciao mondo**. Viene aperto con:

```
FILE * fp = fopen("ciao.txt", "r");
```

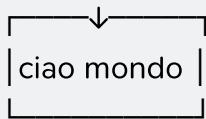
Il cursore a questo punto è all'inizio del file.



Effettuando una lettura con la **fscanf** si legge una parola (spazio finale incluso).

```
fscanf(fp, "%s", buffer);
```

Il cursore sarà quindi all'inizio della parola successiva. Una successiva **fscanf** leggerebbe **mondo**



### Elenco.

1. E' possibile *spostare manualmente il cursore* con la funzione

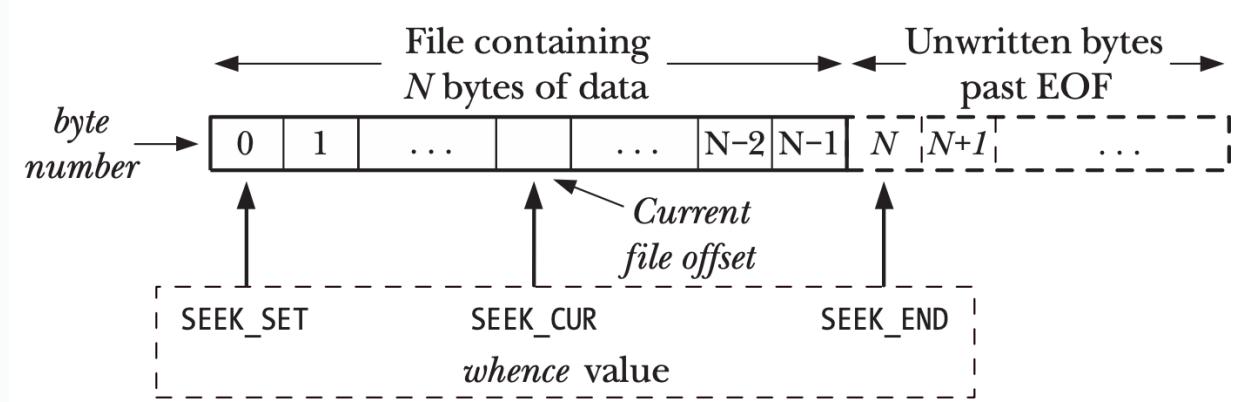
```
int fseek(FILE *fp, long distanza, int partenza)
```

Dove:

- **FILE \*fp**: è il puntatore a file sul quale agire
- **long distanza**: è il nuovo offset.

- **int partenza** o **whence** indica da dove **distanza** viene calcolata. Può assumere
  - **SEEK\_SET**: inizio del file
  - **SEEK\_END**: fine del file
  - **SEEK\_CUR**: posizione corrente del cursore
 Nota: tutti i tre e valori sono delle **costanti** definite nella libreria.

### Figura. (funzionamento di **fseek**)



2. Per sapere *in che posizione è il cursore*:

```
long ftell(FILE *stream);
```

### Esempio.

Consideriamo un file 10 caratteri contenente la frase  
**ciao mondo**.

```
FILE * fp = fopen("ciao.txt", "r");
fscanf(fp, "%s", buffer); // Buffer contiene "ciao". Il cursore è prima di "mondo"

fseek (fp, 0, SEEK_SET); // Il cursore torna all'inizio del file
fscanf(fp, "%s", buffer); // Leggo di nuovo "ciao"
```

Similmente:

```
FILE * fp = fopen("ciao.txt", "r");
fgets(buffer, 2, fp); // Buffer contiene "ci". Il cursore è tra "ci" e "ao mondo"
fseek (fp, -3, SEEK_END); // Il cursore si posiziona tra "ciao mo" "ndo"
fgets(buffer, 2, fp); // Leggo "ndo"
```

# Funzioni per la lettura e la scrittura su File Binari

## Funzioni **fread** e **fwrite**

Simili a **fgets** e **fputs**, ma *per file binari*.

- Ignorano il ritorno a capo **\n**.
- Leggono/scrivono *una quantità fissa di byte*
- Ottime per file binari: *contengono caratteri non stampabili*, e anche tanti '**\0**' (*terminatori*).
  - Impossibile leggere correttamente i terminatori.
  - Possiamo scrivere qualsiasi cosa! Permettono di leggere/scrivere file binari (che contengono '**\0**').
    - Oppure **int**, **float**, **struct** e vettori

**Funzionamento:** leggi/scrivi nella **nmemb** oggetti, ognuno grande **size** byte dal puntatore a file **stream** e scrivilo/leggilo da **ptr**.

**Valore di ritorno:** il numero di elementi effettivamente letti/scritti.

```
C  
size_t fread(void *restrict ptr, size_t size, size_t nmemb,  
            FILE *restrict stream);  
size_t fwrite(const void *restrict ptr, size_t size, size_t nmemb,  
             FILE *restrict stream);
```

## Esempio.

Lettura di un vettore di interi.

Si supponga un file contenente (in binario) i 2 interi che rappresentano i numeri 1990 e 2023. In esadecimale e considerando **int** su 4 byte, avremo nel file:

```
0x00 0x00 0x07 0xC6 0x00 0x00 0x07 0xE7
```

**Suggerimento:** crea il file con **echo -n -e '\x00\x00\x07\xC6\x00\x00\x07\xE7' > ciao.txt**

Per leggerli in C, si procede con la funzione **fread**

```
FILE * fp = fopen("ciao.txt", "rb"); // Notare modalità "rb"
int v [2];
fread(v, sizeof(int), 2, fp);
```

**Nota:** è errato usare **fscanf(... , "%d", ...)** che si aspetta dei numeri scritti come stringhe!

### Osservazioni dall'esempio:

- **size\_t** è un alias per il tipo di dato *intero senza segno* che viene usato per rappresentare grandezze di strutture dati.
- **sizeof** è un operatore che ritorna il numero di byte che un tipo di dato occupa
  - Durante la *compilazione*, il compilatore sostituisce l'espressione col suo risultato

## Bufferizzazione

Adesso evidenziamo un *aspetto particolare* della lettura/scrittura di file su C

### Definizione. (*Bufferizzazione*)

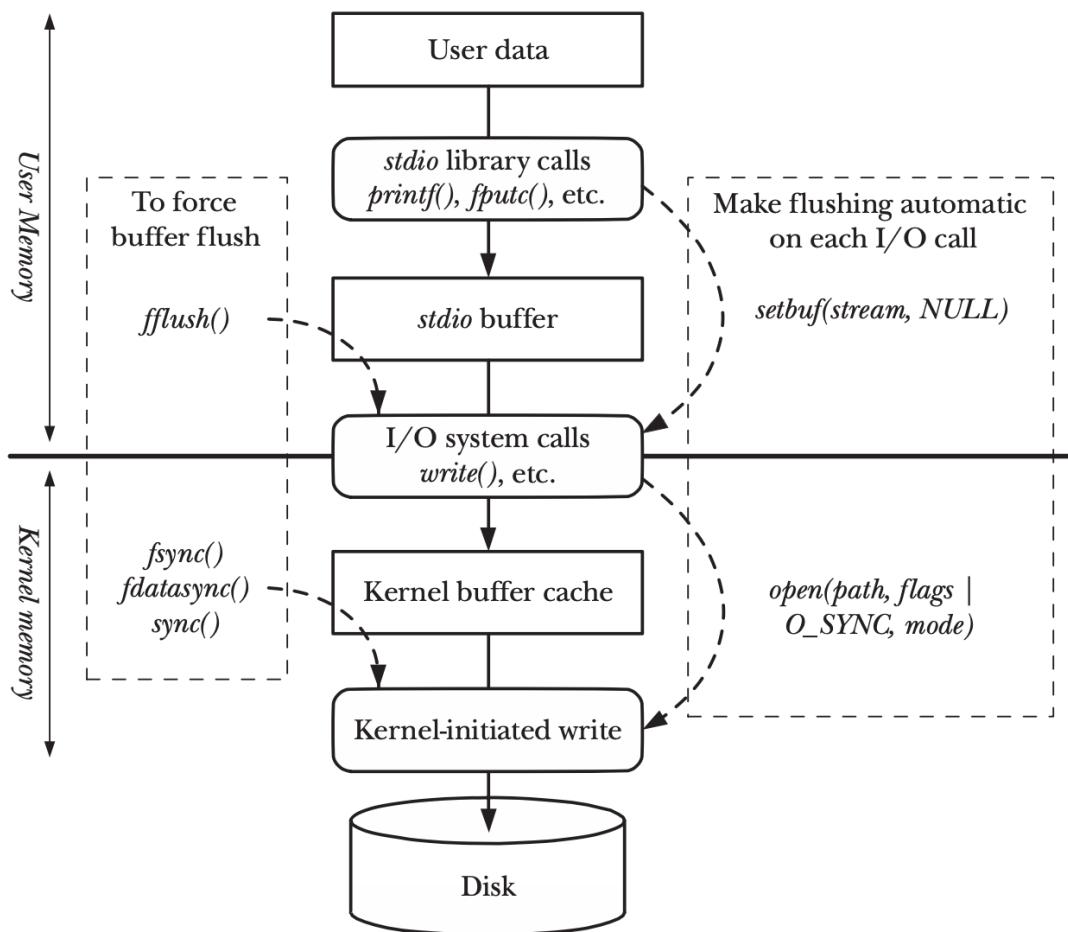
In C, l'input/output su file è *bufferizzato*

- Le funzioni di scrittura come **fprintf** *non invocano sempre la System Call* per la scrittura su file
- Esse scrivono *su un buffer in memoria*
- Quando esso è  *pieno*, il contenuto è *effettivamente scritto* su file

### Motivazione.

Questo comportamento *migliora significativamente le prestazioni* in caso di tante scritture di piccole dimensione: bisognerebbe accedere al disco ad ogni operazione! Quindi in un'unica andata scriviamo ciò che ci interessa.

### Figura. (*Bufferizzazione*)



## Manipolazione delle Bufferizzazioni

**Definizione.** (*Costante BUFSIZ*)

La dimensione di default è la costante **BUFSIZ**, che è di circa **65kB**.

E' possibile *settare la dimensione del buffer* per casi particolari:

```
void setbuf(FILE *stream, char *buf);
```

### Funzione.

1. Si può *forzare una scrittura su file*:

```
int fflush(FILE *stream);
```

**Nota:** quando lo standard output è su console esso è *new line-buffered*. Un ritorno a capo `\n`, forza il *flush*.

Per rendere uno stream *new line-buffered* si usa la funzione **setvbuf**.

# Funzione per la Rimozione dei File

## Rimozione

```
C  
int remove(const char *pathname);
```

Non è necessario che il file sia aperto.

In Linux, **remove()** usa la *System Call*:

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

Vedremo come mai si chiama "**unlink**" in seguito, quando approfondiremo gli *hard/soft link*.

---

## System Call per file

### System Call per file: Recap

Le funzioni viste precedentemente *sono parte della libreria standard del C*.

- Esse *utilizzano delle System Call* per compiere le operazioni richieste
- Le System Call *variano* a seconda del SO

System Call usate, esempi.

- Linux/POSIX: **open**, **read**, **write**, **lseek**, **close**
- Windows: **CreateFile**, **WriteFile**, **ReadFile**, **CloseHandle**, **SetFilePointer**

### System Call di Linux.

Noi ci soffermiamo sulle *System Call di Linux*, usabili includendo **<unistd.h>**.

- Esse sono *simili alle funzioni di libreria*, ma sono di più basso livello.
- Si possono usare su *sistemi Linux e Posix*
- *Non esistono su Windows*. Non sono parte di Libreria Standard del C

**Figura.** (*Primo confronto tra System call e Funzioni di Libreria*)

System calls	Library functions
file descriptor ( <i>int</i> )	file stream ( <i>FILE *</i> )
<i>open()</i> , <i>close()</i>	<i>fopen()</i> , <i>fclose()</i>
<i>lseek()</i>	<i>fseek()</i> , <i>ftell()</i>
<i>read()</i>	<i>fgets()</i> , <i>fscanf()</i> , <i>fread()</i> . . .
<i>write()</i>	<i>fputs()</i> , <i>fprintf()</i> , <i>fwrite()</i> , . . .
—	<i>feof()</i> , <i>ferror()</i>

## Definizioni Relative all'Apertura di File

### Definizione. (*file descriptor*)

In Linux, un *file aperto* è identificato da un *file descriptor*.

Esso è un *numero intero non negativo*, che per convenzione hanno tre significati:

- Standard Input: descrittore 0 (stdin)
- Standard Output: descrittore 1 (stdout)
- Standard Error: descrittore 2 (stderr)

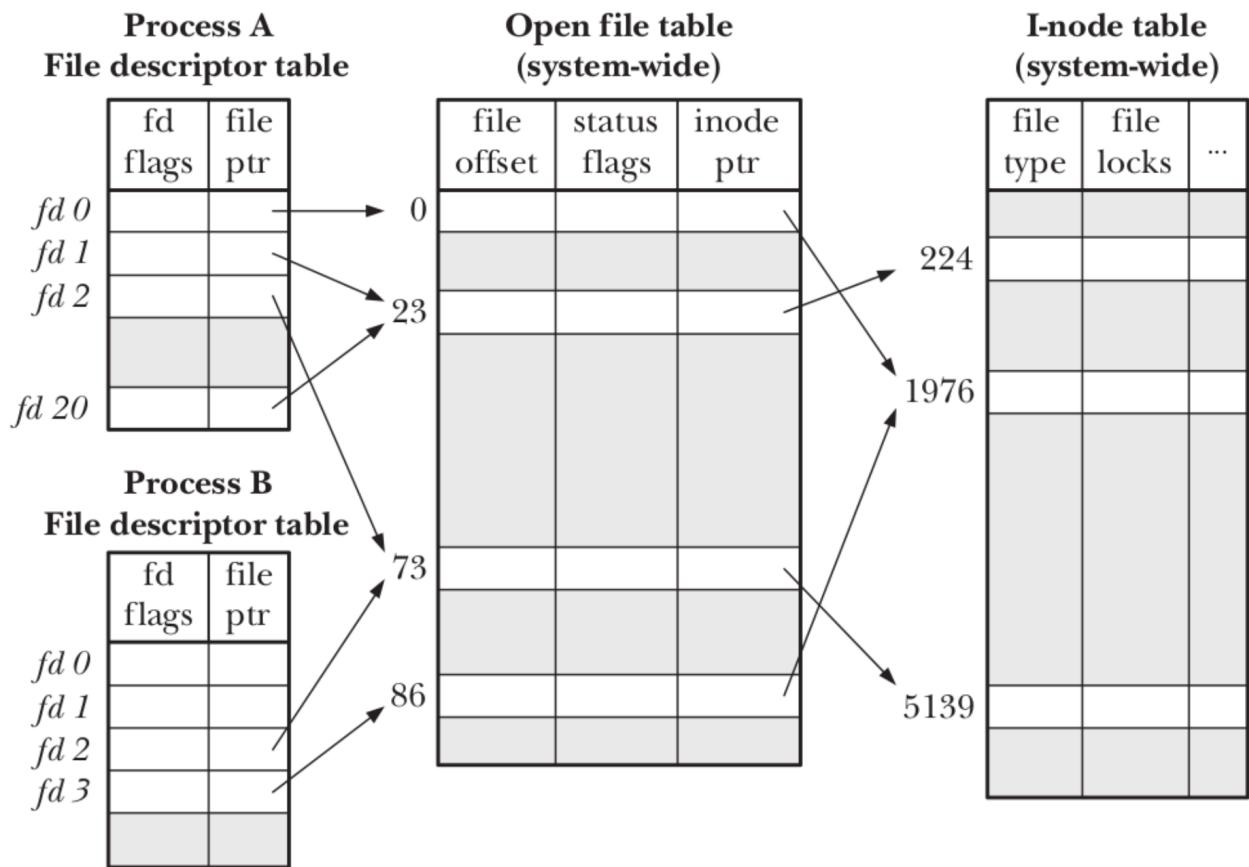
### Nota. (*differenza*)

Nelle funzioni di libreria un file aperto è un **FILE \***. Nelle System Call Linux è un **int**.

### Definizione. (*File table*)

Il sistema operativo mantiene delle *tabelle* (dette come "*file table*") che mappano i *file descriptor* ai file fisici su disco (*inode*), ovver una *serie di strutture dati*.

### Figura. (*Schema*)



**Nota.** Le tabelle dei *processi A, processi B* rispondono alla domanda: "*quali file sono aperti nei processi individuali?*"; la tabella centrale, ovvero quella generale del S.O. agisce su *dei processi* e l'ultima tabella è la *tabella degli i-node corrispondente*, che serve le richieste.

### Analisi Approfondita.

- Per ogni *processo*, vi è una tabella che contiene i *file descriptor*:
  - Contiene un *riferimento alla tabella generale*
  - E dei *flag* (come ad esempio le modalità di apertura del file)
- La *tabella generale* (una per tutto il SO), contiene:
  - Access Mode: *R, W, RW*
  - File Offset: posizione del *cursore*
- La *tabella degli inode* è semplicemente una *copia in memoria degli inode interessati* (che si trovano su disco)

## Apertura di File con System Call

### Apertura di un file

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

- Apre il file identificato dal path **pathname**.

- **flags** determinano modalità di accesso al file.
  - Uno tra **O\_RDONLY**, **O\_WRONLY**, e **O\_RDWR** deve essere *obbligatoriamente* presente.
  - Altri flag sono:
    - **O\_CREAT** crea il file se non esiste
    - **O\_APPEND** apri il file in modalità aggiunta
    - *Nota:* con le funzioni di libreria questo è il *comportamento default*.
  - I flag si sommano usando l'operatore OR bit a bit |

## Esempi.

```
C  
int fd = open("file.txt", O_RDONLY);  
int fd = open("file.txt", O_WRONLY | O_APPEND);
```

### Flag **O\_CREAT**.

Nel caso si specifichi il flag **O\_CREAT**, il file viene creato *coi permessi specificati* in **mode**.

Esistono 9 flag:

- **S\_I[RWX]USR**, **S\_I[RWX]GRP**, **S\_I[RWX]OTH**  
*Ricordiamoci* che in Linux i file hanno 3 tipi di permessi (Read, Write, Execute), gestibili separatamente per il *proprietario, il gruppo e gli altri utenti* (1).

---

## Chiusura di File con System Call

### Chiusura di un file:

```
C  
int close(int fd);
```

Chiude il *file descriptor*. Il numero **fd** non si riferisce più a nessun file aperto e *può essere riutilizzato in successive* **open** da parte del SO.

---

## Lettura e Scrittura su File con System Call

### 1. Lettura da file:

```
ssize_t read(int fd, void *buf, size_t count);
```

Leggi **count** byte da **fd** e mettili nella memoria all'indirizzo **buf**.

**Valore di ritorno:** il *numero di byte letti*. Può essere minore di **count** se il file finisce.

- In caso di errore -1
- Se si è giunti a EOF 0

## 2. Scrittura su file:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Scrivi **count** byte su **fd** e prendendoli nella memoria all'indirizzo **buf**.

**Valore di ritorno:** il *numero di byte scritti*. Può essere inferiore a **count** se il disco si riempie.

- In caso di errore -1

**Nota.** **ssize\_t** sta per "signed size\_t".

# Manipolazione Cursore con System Call

## Riposizionamento Cursore:

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

Molto simile alla funzione di libreria **fseek**.

Riposiziona il file descriptor **fd** all'offset **offset** secondo la direttiva **whence** come segue:

- **SEEK\_SET**: **offset** è rispetto a inizio file
- **SEEK\_CUR**: **offset** è rispetto a posizione corrente
- **SEEK\_END**: **offset** è rispetto a fine file. **offset** dovrà essere negativo

**Nota.** C'è una corrispondenza uno a uno con la *funzione di libreria fseek*

## Esempi Generali

**Esempio:** scrittura utilizzando System Call

```
const char *str = "Arbitrary string to be written to a file.\n";
const char* filename = "innn.txt";

int fd = open(filename, O_RDWR | O_CREAT);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

write(fd, str, strlen(str));
printf("Done Writing!\n");

close(fd);
```

**Esempio:** equivalente usando funzioni di libreria

```
const char *str = "Arbitrary string to be written to a file.\n";
const char* filename = "innn.txt";

FILE* output_file = fopen(filename, "w+");
if (!output_file) {
    perror("fopen");
    exit(EXIT_FAILURE);
}

fwrite(str, 1, strlen(str), output_file); // Si può usare fputs o fprintf
printf("Done Writing!\n");

fclose(output_file);
```

**Esempio:** sono equivalenti le seguenti forme per stampare il messaggio **Hello World** su console.

Usando la **printf** si stampa su Standard Output.

C

```
printf("Hello World\n");
```

Si può usare la **fprintf** dicendole di stampare sul file **stdout**, che è un **FILE \*** pre-definito.

C

```
fprintf(stdout, "Hello World\n");
```

Si può usare la System Call **write** (solo su Linux/POSIX).

- Si stampa sul file descriptor 1, per convenzione lo Standard Output
- E' necessario specificare quanti byte scrivere. **write** stampa dei byte, non utilizza il terminatore '**\0**' (valore 0)

C

```
write(1, "Hello World\n", 13);
```

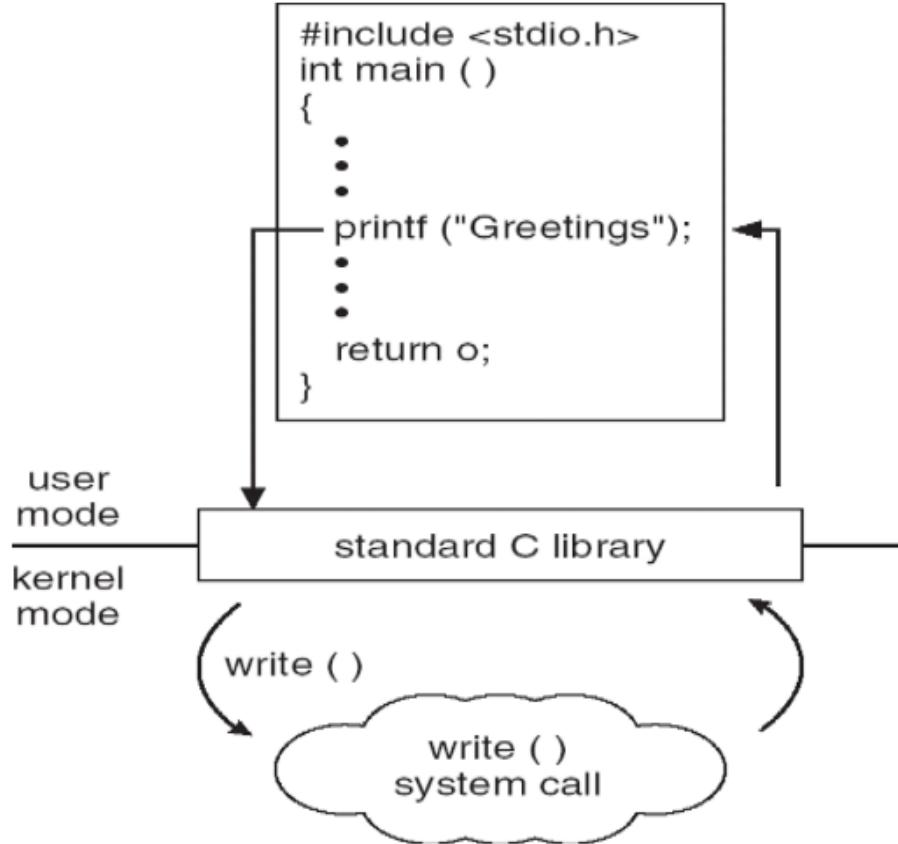
---

## Funzioni di libreria e System Call

Adesso confrontiamo le *funzioni di libreria* con le *system call* per manipolare file.

### Aspetti Diversi

1. Le funzioni di libreria utilizzando le System Call



2. Le **Funzioni di Libreria** sono eseguite in *modalità utente*. Non hanno nessun privilegio particolare.

- Sono semplicemente delle *funzioni che facilitano l'uso delle System Call*

3. Le **System Call** sono eseguite in *modalità kernel*.

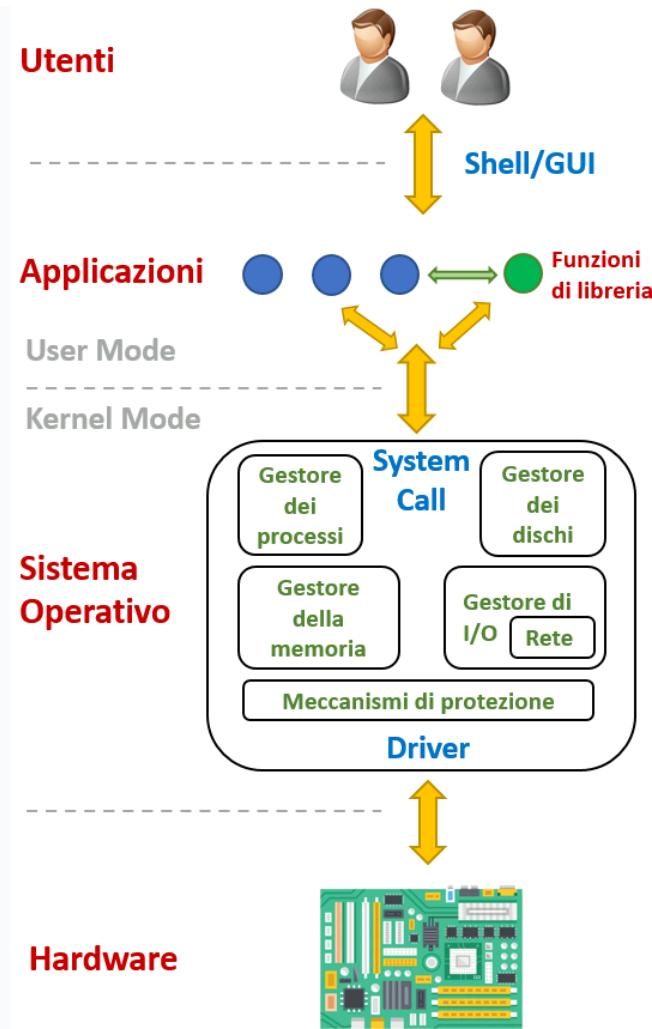
- Hanno accesso alla *memoria fisica*
- Possono accedere alle interfacce dei dispositivi di I/O
- Vengono fornite dal *Sistema Operativo*

## Aspetti Comuni

Le **Applicazioni** possono invocare *sia funzioni di libreria che System Call*.

Se vogliono usufruire dei servizi del SO è *sempre necessario usare le system call*

- Lo fanno *le applicazioni direttamente*
- Oppure lo fanno *le funzioni di libreria* invocate dalle applicazioni

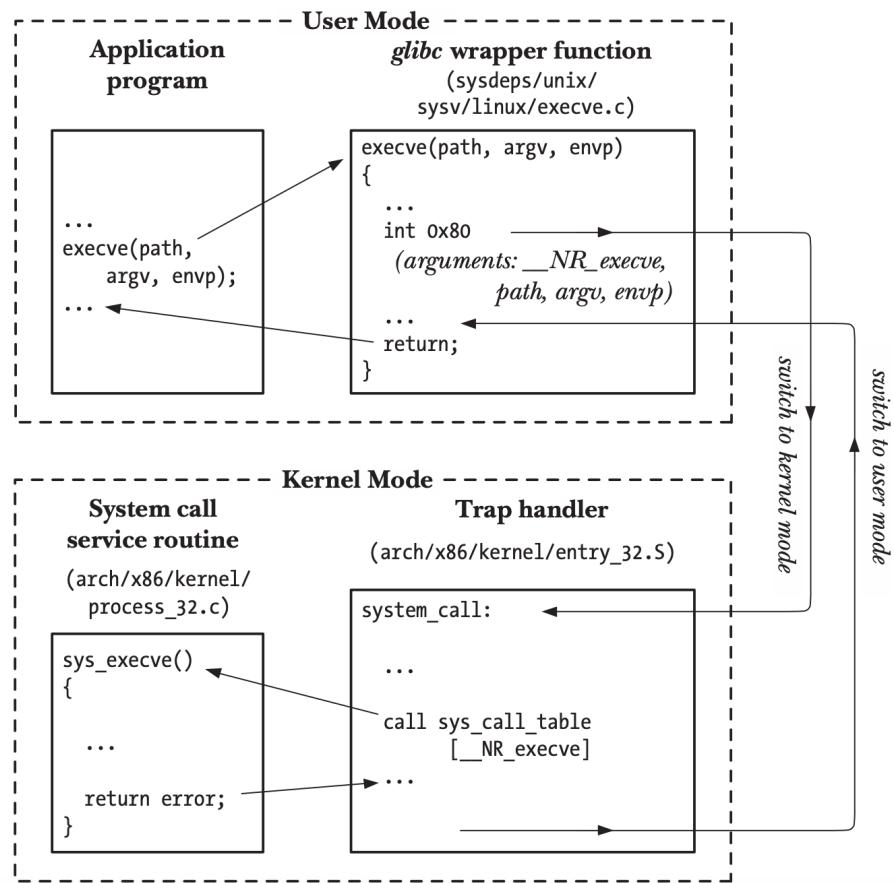


## Passaggi per l'invocazione di una System Call

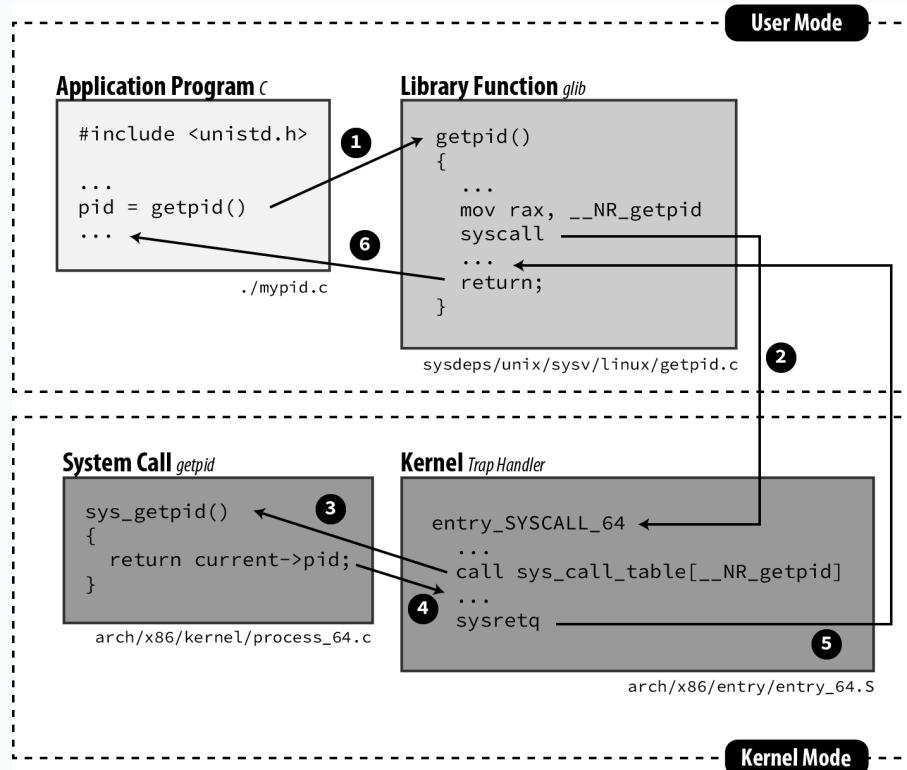
Ci sono *due metodi* per invocare una system call.

### 1. Tramite **int 0x80** (*metodo classico*)

- **Nota.** La funzione **int 0x80** non è altro che una *interrupt* (1), dal momento che viene chiamata la **CPU** viene interrotta ed esegue un pezzo di codice predefinito. Da osservare che c'è un pezzo di assembly nel codice!



## 2. Tramite **syscall** (metodo moderno)



**Osservazione.** Qui abbiamo sempre una **situazione delicata!** Se non siamo abbastanza attenti, questo potrebbe generare delle **vulnerabilità di sicurezza**, dal momento che stiamo passando da **user-mode** a **kernel-mode** in una maniera **quasi-arbitraria**.

# Manuale di Linux

## Manuale:

Il manuale di Linux è diviso in sezioni:

1. User Commands
2. System Calls
3. C Library Functions
4. Devices and Special Files
5. Eccetera...

La **fopen** è in sezione 3, la **open** in sezione 2.

Invece **printf** è sia un *comando bash* che una *funzione di libreria*

- Ha *due pagine di manuale*
  - **man 1 printf**: comando Bash
  - **man 3 printf**: funzione di libreria C

## Strumenti per Debuggare Codici con System Call

Esistono degli strumenti di debug per vedere quali System Call vengono invocate da un processo.

- Si può fare tramite **profile** (e.g., **valgrind**: utile per vedere le prestazioni), **debugger** (e.g., **gdb**, utile per vedere tutto passo a passo) o tool nativi (e.g., **strace**, per stampare una lista di System Call invocati)
  - Ci focalizziamo in particolare su **strace** e su **gdb**.
1. **strace**  
*Non richiede di ricompilare i programmi.* Funziona sempre, anche se non ho il sorgente del programma. E' un tool del SO.

**Funzionamento:** **strace comando**

**Esempio:** Di default **strace** lista le system call

SHELL

```
$ strace pwd
execve("/usr/bin/pwd", ["pwd"], 0x7ffd37cdfc80 /* 72 vars */      = 0
...
getcwd("/tmp", 4096)          = 5
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}) = 0
write(1, "/tmp\n", 5/tmp)     = 5
```

Si può semplicemente contare le invocazioni a System Call

SHELL

```
$ strace -c date
mar 25 ott 2022, 10:16:48, CEST
% time    seconds  usecs/call   calls  errors syscall
-----
0,00  0,000000      0       6      read
0,00  0,000000      0       1      write
0,00  0,000000      0       9      close
...
...
```

## 2. **ltrace**

Simile a **strace**. Permette di visualizzare le *funzioni di libreria* usate da un processo

**Funzionamento:** **ltrace comando**

**Nota:** *non funziona su tutti gli eseguibili.*

Solo quelli compilati con *lazy binding* (opzione del linker), di default fino a Ubuntu 16.

Per essere sicuri, compilare con l'opzione: **-z lazy**

Esempio: **gcc sample.c -o myprog -z lazy**

**Esempio:** Di default **ltrace** lista le invocazioni a funzione

SHELL

```
$ ltrace pwd
...
getcwd(0, 0)           = ***
puts("/home/det_user/trevisan")/home/det_user/trevisan
)                      = 24
...
```

Per contare le funzioni invocate:

```
$ ltrace -c date
Tue Oct 25 10:35:00 CEST 2022
% time    seconds  usecs/call  calls  function
-----
15.59  0.000543      67      8 fwrite
13.90  0.000484      60      8 fputc
 8.76  0.000305     305      1 setlocale
...
...
```

## Comandi Bash per File

Ripassiamo i comandi Bash per manipolare i file.

### Le Utility di GNU/Linux.

I SO Linux/Posix hanno dei *programmi pre-installati per gestire i file*:

Sono delle *utility* che permettono di svolgere compiti semplici e ripetitivi da riga di comando

Senza dover scrivere un programma apposito che chiami le *System Call o Funzioni di Libreria necessarie*.

Sono documentati nella sezione 1 di **man**

#### 1. Lettura di file:

- **cat filename**: stampa su Standard Output il contenuto di un file
- **less filename** e **more filename**: visualizzazione passo-passo
- **head filename** e **tail filename**: stampa prime/ultime righe di un file
- **grep pattern file**: stampa delle righe che contengono un **pattern**

#### 2. Scrittura di file:

- Per scrivere su file si usa tipicamente la *redirezione su file* della Bash:

```
echo "Ciao" > filename
```

- Con **touch filename** creo un file se non esiste

#### 3. Rimozione di file:

- **rm filename**: rimuove un file (se ho i permessi per farlo)

#### 4. Altre operazioni:

- **cp origine destinazione**: copia un file
- **chmod, chgrp, chown**: modificano permessi, gruppo e proprietario di un file

## Domande

La **fopen** é:

- Una funzione di libreria
- Una System Call
- Una struct

Risposta: *Funzione di Libreria*

La **read** é:

- Una funzione di libreria
- Una System Call
- Una struct

Risposta: *System Call*

Si consideri il file **f.txt** contente il testo **frase di prova**.

Dove si trova il cursore ( ) del file dopo il seguente codice?

```
FILE * fp = fopen("f.txt","r");
fscanf(fp, "%s", buffer);
fseek (fp, 3, SEEK_CUR);
```

- **fra↓se di prova**
- **frase di ↓prova**
- **frase di pr↓ova**
- **frase ↓di prova**

Risposta: *frase di ↓prova*

Quale delle seguenti linee di codice é corretta?

- **int fd = open("file.txt", O\_RDONLY);**
- **FILE \* fd = open("file.txt", O\_RDONLY);**
- **FILE \* fd = fopen("file.txt", O\_RDONLY);**
- **FILE fd = open("file.txt", "rw");**

Risposta: *int fd open("file.txt", O\_RDONLY);*

Quale relazione c'è tra la **fprintf** e **write**?

- **fprintf** é una funzione di libreria e usa la SysCall **write**
- **write** é una funzione di libreria e usa la SysCall **fprintf**
- **fprintf** e **write** sono due funzioni di libreria
- **fprintf** e **write** sono due System Call

Risposta: *\*fprintf è una funzione di libreria e usa la SysCall write"*

Quale dei seguenti comandi stampa a schermo il contenuto del file **ciao**?

- **echo ciao**
- **print ciao**
- **cat ciao**

Risposta: *cat ciao*

### Link e Directory

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

### Argomenti

1. Link
  2. Soft Link
  3. Hard Link ---
  4. System call per Directory
  5. Funzioni di libreria per Directory
  6. Comandi Bash per Link, Directory e Disco ---
- 

## Link

**Definizione.** (*Link generalizzato*)

Un link é un *nome aggiuntivo per un altro file*.

Utile per svariati compiti

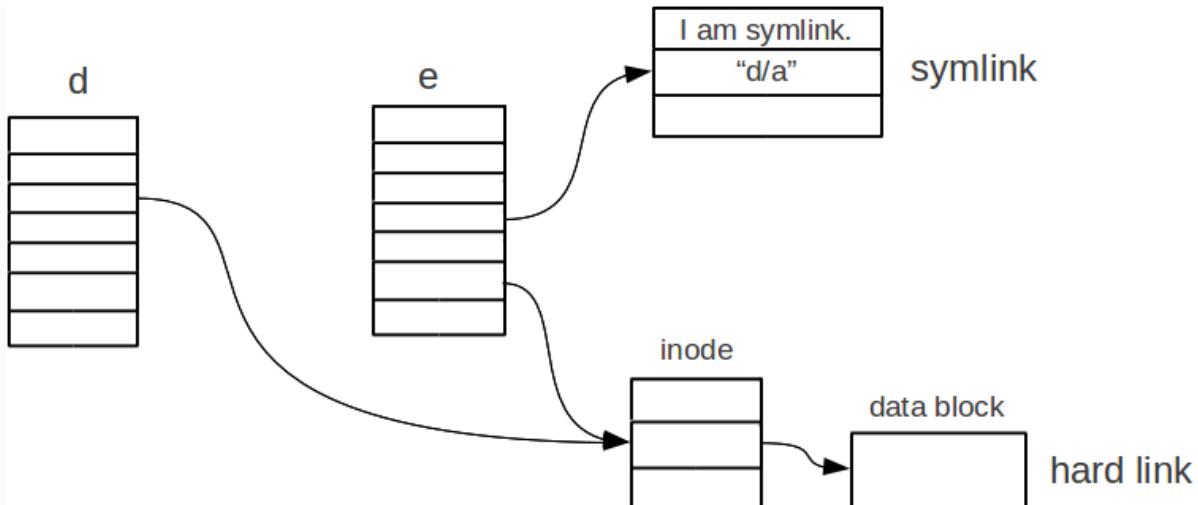
- Gestione file di configurazione
- Condivisione di informazioni tra utenti
- Mantenimento ordinato della struttura dei file
- E molto altro!

**Definizione.** (*Hard Link e Soft Link*)

Esistono due tipi di link:

- **Hard Link:** l'inode appare *in una seconda directory* che vi punta; abbiamo quindi un *riferimento aggiuntivo* allo stesso *i-node*.
- **Soft Link:** é un alias a un certo path; in questo caso è un "*file speciale*" che va a recuperare il contenuto del file puntato, se possibile.

**Figura.** (*Idea*)



## Comportamenti del Soft Link

Come detto prima, i soft link sono degli *Shortcut* per un file o una directory

- Se ho un grande file con un *path lungo e complesso*, ne posso creare un soft link nella mia Home Directory
- Se cancello un Soft Link, *non succede niente al file originale*
- Se il file originale viene cancellato, il Soft Link *continua a esistere* ma diventa *invalido*
- Se creo un altro file con quel nome, il Soft Link *torna a essere valido*
- I Soft Link sono molto flessibili

### System Call.

E' un concetto di Linux

Per creare un Soft Link si usa la *System Call*:

```
int symlink(const char *target, const char *linkpath);
```

Si rimuove un Soft Link come fosse un normale file.

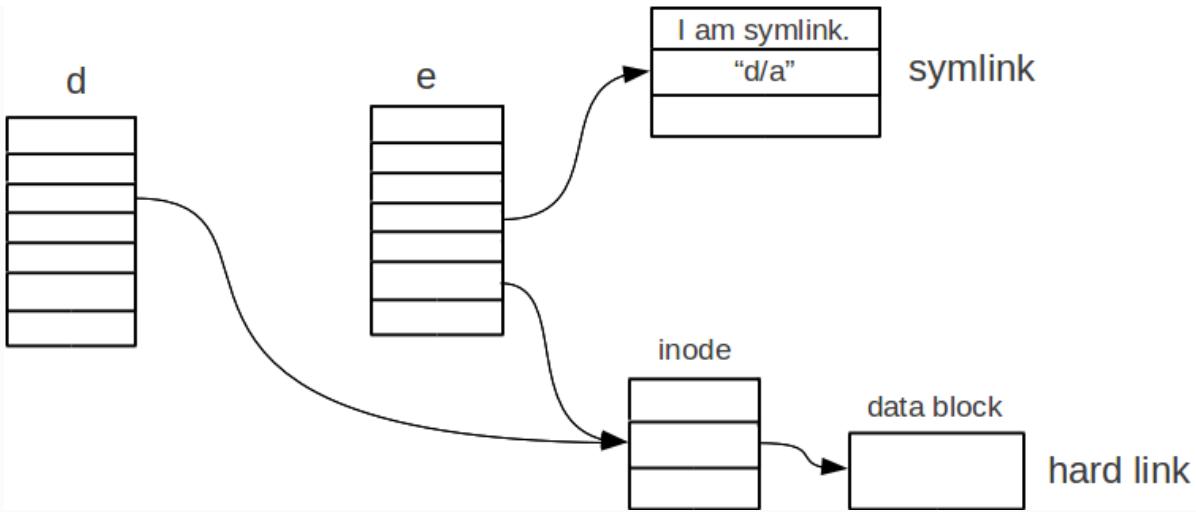
*Nota:* Si possono creare Soft Link a cartelle e verso altri dischi

Le funzioni di ricerca *non devono attraversare i Soft Link* per evitare cicli

## Comportamenti del Hard Link

Un Hard Link é un *riferimento aggiuntivo a un inode*

La directory dove viene creato, contiene *una nuova entry* che ha lo stesso *inode number*



### Implicazioni.

- Un Hard Link è *un link al contenuto del file* (quindi non un semplice rimando)
- Non può mai essere *invalido* (a meno che qualcosa di *veramente brutto* sia successo col File System)
- Hard Link e file originario hanno la *stessa importanza e la stessa natura*: in un certo senso diventano la stessa cosa
- Cancellare un Hard Link causa la cancellazione del file *solo se non vi sono altri Hard Link* (o il riferimento originale). Come si fa? Questo è il compito del *sistema operativo*.

Compiti del sistema operativo

- Mantenere un *reference count* per ogni inode
- Cancellare un inode e il suo contenuto *se esso va* a 0
- Questo è l'approccio accettato, dato che altri approcci sarebbero impensabili. Ad esempio, quello di cercare l'interno disco in ricerca di altri riferimenti non può essere fattibile.

### System Call.

Si creano con la System Call:

```
int link(const char *oldpath, const char *newpath);
```

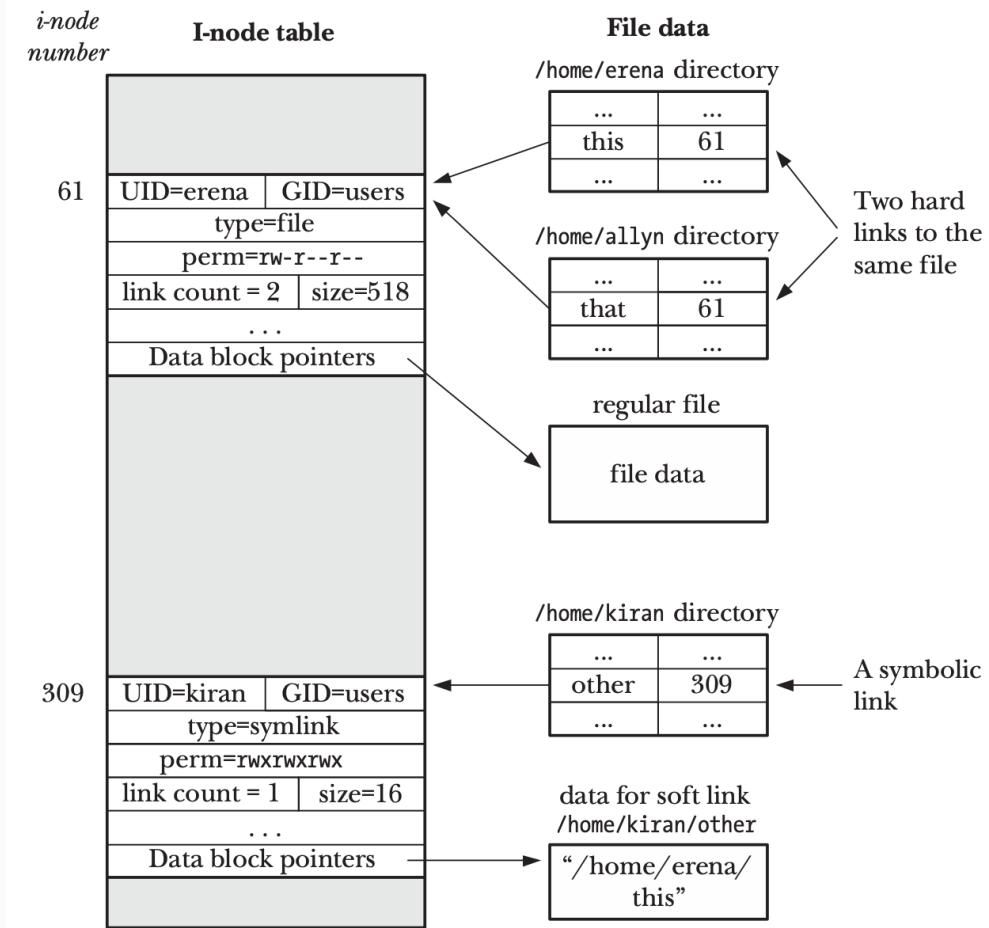
**Nota:** La System Call per rimuovere un file in Linux è **unlink**.

Di fatto rimuovere un file, significa decrementare il suo *reference count* (e *eventualmente rimuovere l'inode*)

*Non si possono creare Hard Link a cartelle.* Concettualmente sbagliato; ci sarebbe un pastroccio!

## Differenza tra Hard Link e Soft Link

**Figura.** (*Differenza tra Hard Link e Soft Link*)



**Osservazioni.** Non posso creare degli *hard link* per file su *dischi diversi*, dato che avrei degli *inode diversi* per *dischi diversi*.

## System Call per Interagire con Directory

### System call per Directory: Leggere

#### 1. Lettura di informazioni su una directory o un file:

```
#include <sys/stat.h>
int stat(const char *restrict pathname,
         struct stat * statbuf);
```

Ritorna informazioni su **pathname**, le colloca nella struttura puntata da **statbuf**

- Passaggio di variable per riferimento che é di fatto un valore di ritorno
- Ritorna 0 in caso di successo, -1 in caso di errore
- La struttura **statbuf** contiene i *metadati* dell'inode

## Definizione. (*struct stat*)

La **struct stat** ritornata contiene i seguenti campi:

```
struct stat {  
    dev_t    st_dev;      /* ID of device containing file */  
    ino_t    st_ino;      /* Inode number */  
    mode_t   st_mode;     /* File type and mode */  
    nlink_t  st_nlink;    /* Number of hard links */  
    uid_t    st_uid;      /* User ID of owner */  
    gid_t    st_gid;      /* Group ID of owner */  
    dev_t    st_rdev;     /* Device ID (if special file) */  
    off_t    st_size;     /* Total size, in bytes */  
    blksize_t st_blksize; /* Block size for filesystem I/O */  
    blkcnt_t st_blocks;   /* Number of 512B blocks allocated */  
  
    struct timespec st_atim; /* Time of last access */  
    struct timespec st_mtim; /* Time of last modification */  
    struct timespec st_ctim; /* Time of last status change */  
};
```

## Definizione. (*mode\_t*)

Il campo **mode\_t** indica *se si tratta di file* (o qualsiasi altra cosa) in forma di una **bit mask**.

Si possono usare le seguenti macro per testare facilmente **mode\_t**

- **S\_ISREG(m)**: True se file regolare
- **S\_ISDIR(m)**: True se file directory
- **S\_ISLNK(m)**: True se file un Symbolic Link

---

## System call per Directory: Esempio

Si scriva un programma che riceve un path da riga di comando e stampa se esso é file, directory o link simbolico. *Importante per l'esame!*

```
#include <stdio.h>
#include <stdlib.h> // Necessario per exit
#include <sys/stat.h> // Necessario per stat
int main (int argc, char * argv[])
{
    struct stat buf;
    if (argc!=2){
        printf("Specifica un path\n");
        return 1;
    }

    if (stat(argv[1], &buf) < 0) {
        printf ("Impossibile leggere le informazioni sul file\n");
        exit (1); /* Termina subito il programma con codice 1 */
    }

    if (S_ISREG(buf.st_mode))
        printf("%s: file\n", argv[1]);
    else if (S_ISDIR(buf.st_mode))
        printf("%s: directory\n", argv[1]);
    else if (S_ISLNK(buf.st_mode))
        printf("%s: link simbolico\n", argv[1]);
    else
        printf("%s: altro tipo di path\n", argv[1]);

    return 0;
}
```

## System call per Directory: Creazione e Rimozione

### 2. Creazione di directory

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

### 3. Rimozione di directory

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

- Rimuove *solo se* la cartella è vuota.

**mode** ha stesso ruolo che nella **open**

Valore di ritorno 0 in caso di successo –1 in caso di errore

---

## System call per Directory: Listare

Per *listare il contenuto di una directory* si possono usare le System Call **open** e **getdents** e la **struct linux\_dirent**.

1. Si apre una directory come fosse un file

```
int fd = open("path", O_RDONLY | O_DIRECTORY);
```

2. Si leggono batch di **struct linux\_dirent**

```
int nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);
```

Tutto ciò è difficile, *poco pratico e non portabile*. Infatti, manca esiste un wrapper in C per invocarla. Per noi sarà inutile, e useremo direttamente le *funzioni di libreria*.

Si usano sempre le *funzioni di libreria POSIX* per leggere il contenuto di una cartella

---

## Funzioni di Libreria per Directory

### Listare File con Funzione di Libreria

Per listare il *contenuto di una cartella*, si usano le funzioni di libreria

```
#include <sys/types.h> // per definire tutti i tipi derivati
#include <dirent.h> // per dirent
DIR * opendir(const char *name);
struct dirent *readdir(DIR *dirp);
// faccio quello che voglio
int closedir(DIR *dirp);
```

Funzionano **solo** su sistemi POSIX.

Su Windows si usano **FindFirstFile()** e **FindNextFile()**.

1. **Apertura:** una cartella, prima di essere letta, va aperta con **opendir**.

Essa ritorna un puntatore a **DIR \*** se l'apertura va a buon fine, altrimenti **NULL**

Un **DIR \*** è l'equivalente di **FILE \*** per le directory

```
DIR * d;  
d = opendir("/path/");  
if (d!=NULL)  
...  
...
```

### Struttura "dirent".

Una **struct dirent** rappresenta un elemento di una directory.

Contiene i campi:

```
struct dirent  
{  
    ino_t      d_ino;    /* inode number */  
    char       d_name[256]; /* filename */  
    ... // Da qui in poi è irrilevante per noi  
};
```

2. **Listare il contenuto:** si usa la funzione **readdir** che ritorna **una struct dirent \***

- Opera in maniera sequenziale
  - A ogni invocazione ritorna l'elemento successivo
- Va invocata finche non ritorna **NULL** (ovvero ho finito tutti i file da elencare)

```
struct dirent * entry;  
while ((entry = readdir(d)) != NULL)  
    printf(" %s\n", entry->d_name);
```

- L'ordine degli elementi ritornati *dipende dal FS* e di solito *non ha alcun significato*

3. **Chiusura cartelle.** Infine bisogna chiudere una cartella per rilasciare le risorse associate.

```
int r = closedir(d);
```

Ritorna 0 se l'operazione va a buon fine, altrimenti –1. Ad esempio, se provo a chiudere una cartella *più di una volta*, allora finisce male.

---

## Funzioni di libreria per Directory: Esempio

Si scriva un programma che riceve una directory da riga di comando e ne lista il contenuto. Importante per l'esercitazione e per l'*esame*!

```
#include <stdio.h>
#include <stdlib.h> // Necessario per exit
#include <sys/stat.h> // Necessario per stat
#include <dirent.h> // Necessario per struct dirent *
int main (int argc, char * argv[])
{
    struct stat buf;
    struct dirent *dirp;
    DIR *dp;

    if (argc!=2){
        printf("Specifica un path\n"); exit (1);
    if (stat(argv[1], &buf) < 0) {
        printf ("Impossibile leggere le informazioni sul file\n"); exit (1);
    if (!S_ISDIR(buf.st_mode)){
        printf("%s deve essere una directory\n", argv[1]); exit(1);
    if ( (dp = opendir(argv[1])) == NULL) {
        printf("%s impossibile da aprire\n", argv[1]); exit (1);

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

    closedir(dp);

    return 0;
}
```

---

## Comandi Bash per Link, Directory

Sono programmi pre-installati che facilitano l'uso delle System Call per compiti comuni.  
Ne facciamo un ripasso

### Comandi per link:

SHELL

```
ln target link_name
```

Crea un link al path **link\_name** verso un path esistente **target**

- Di default crea un hard link
- L'opzione **-s**
- Si rimuovono i link con **rm**
- Si possono anche usare i comandi più grezzi **link** e **unlink**

### Comandi per stat:

SHELL

```
stat path
```

Invoca la System Call **stat()** sul **path** specificato

- Stampa in formato human-readable tutto ciò che c'è dentro la **struct stat** risultante (e poco altro)
- Esempio:

```
$ stat file.tex
  File: file.tex
  Size: 0          Blocks: 0          IO Block: 4194304 regular empty file
Device: 31h/49d Inode: 1099555504104  Links: 1
Access: (0644/-rw-r--r--) Uid: ( 5012/trevisan)  Gid: ( 5000/det_user)
Access: 2022-01-07 11:00:28.001143767 +0100
Modify: 2022-01-07 11:01:29.860547368 +0100
Change: 2022-01-07 11:01:29.860547368 +0100
 Birth: -
```

### Comandi per directory:

- **ls directory**: lista il contenuto
- **ll directory**: alias per **ls -lh directory**
- **mkdir directory** e **rmdir directory**: crea o rimuove una directory
- **find ...**: cerca all'interno di una cartella

- **tree directory**: stampa l'albero di file e cartelle contenuti
  - **du directory**: ottiene la dimensione della cartella e di tutto ciò che vi è contenuto
- 

## Domande

E' possibile creare link a cartelle

- **Di tipo Hard Link**
- **Di tipo Soft Link**
- **Sempre**
- **Mai**

Risposta: "*Di tipo Soft Link*"

Un Hard Link può riferirsi a un file inesistente

- **Vero**
- **Falso**

Risposta: "*Falso*"

Un Soft Link può riferirsi a un file inesistente

- **Vero**
- **Falso**

Risposta: "*Vero*"

Quale System Call permette di conoscere lo user ID del proprietario di una directory?

- **open**
- **opendir**
- **readdir**
- **stat**

Risposta: "*stat*"

La directory **dir** contiene i file **f.txt** e **g.txt**. Il seguente codice:

```
dp = opendir("dir1");
while ( (dirp = readdir(dp)) != NULL){
    if (strcmp(dirp->d_name, "dir/f.txt") == 0 )
        printf("%s\n", "Found\n");
```

Stampa:

- **"Found" una volta**
- **"Found" due volte**
- **Niente**

Risposta: *Niente*

## u4-s5-raid

---

Osservazione preliminare per RAID e FDS: le problematiche dei sistemi professionali.

# Problematiche dei Sistemi professionali

## I Sistemi Professionali.

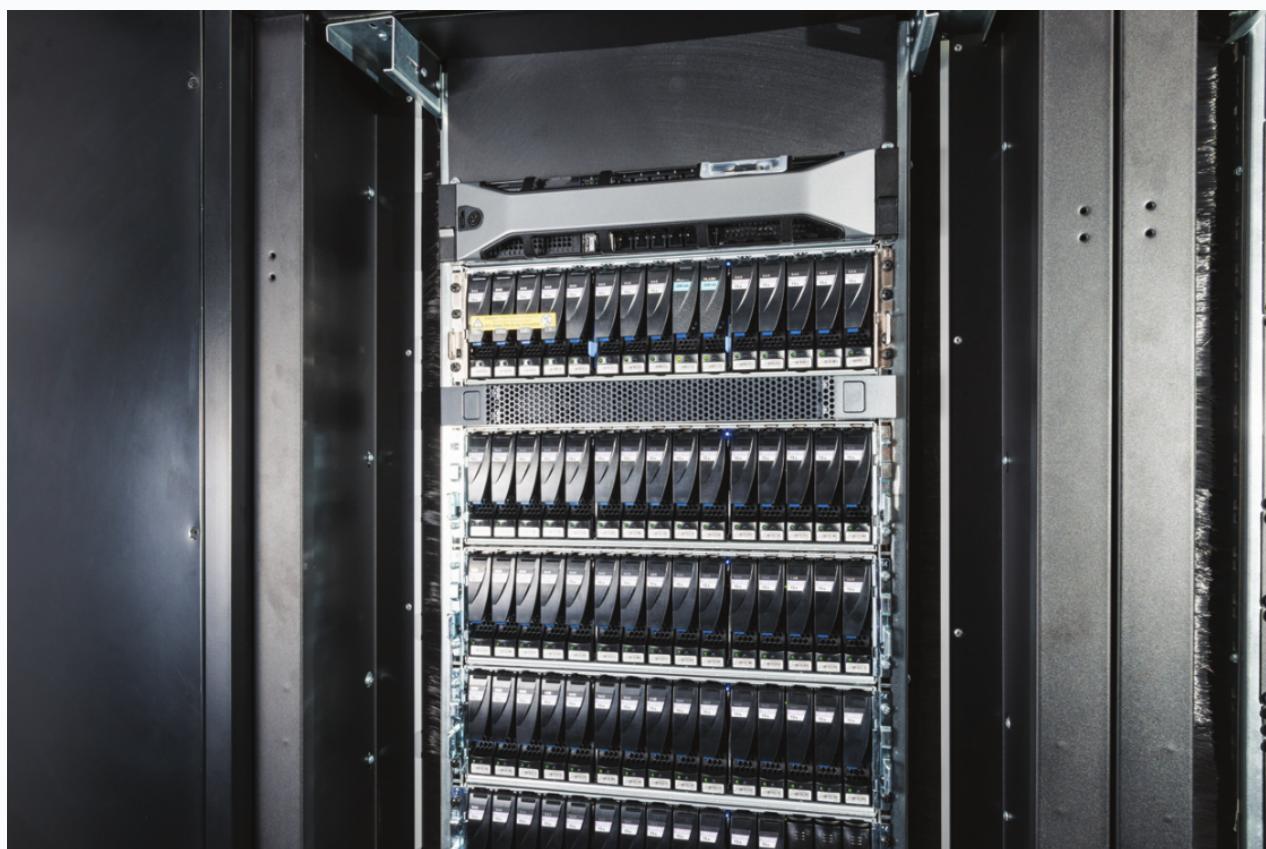
Nei sistemi di grandi dimensioni, una singola macchina ha tanti dischi (possiamo averne fino a 100!)

- Fino a 100 dischi su una stessa macchina
- Premettono di conservare *enormi quantità di dati*

Spesso ci sono server dedicati allo *storage*

- I calcolatori accedono *via rete* ai dati dello storage
- Tramite *protocolli di rete*

**FIGURA.** (*Esempio di rack storage*)



Allora seguono i seguenti *problem*i, da *gestire*.

## Problemi.

1. Sono necessarie tecniche per gestire i *guasti* (*failure*)

- Un disco ha l'1% di probabilità di rompersi *ogni mese*
  - Dato reale per i dischi magnetici
- Se ho 100 dischi, ho in media *un guasto al mese*; nel senso probabilistico, dopo un certo periodo di tempo la *probabilità di avere almeno un guasto* tende a

salire verso 1 ([vedere la Scimmia di Borel](#)).

- Non è pensabile *perdere dati* in sistemi professionali!
- Voglio gestire gli errori in una *maniera automatica*

## 2. Sono auspicabili tecniche per *aumentare le prestazioni*

- Se 100 dischi vengono opportunamente usati in parallelo, possono moltiplicare  $\times 100$  la *velocità* del sistema

-

Adesso vedremo *due soluzioni* per gestire le problematiche: da un lato avremo il *RAID*, e dall'altra i *FDS*.

---

*Definizione di RAID, brevissimo excursus storico. Concetto essenziale di RAID: lo striping. Livelli di RAID: 0, 1, 4, 5, 6. Osservazioni conclusive su RAID.*

---

## Definizione di RAID

Le tecniche *RAID* ("*redundant array of independent disks*") hanno lo scopo di affrontare i problemi di *prestazioni e affidabilità*

- Proposto nel 1988 da David A. Patterson (e altri) nel paper *A Case for Redundant Arrays of Inexpensive Disks (RAID)* (questo era infatti il nome storico)
- Famiglia di metodi per organizzare dati su *batterie di dischi*
- Molto comunemente utilizzato, sia in ambienti professionali che personali.

## Concetti Essenziali di Raid

### Definizione. (*striping*)

Si basa su *striping*, ovvero distribuire i dati su  $N$  dischi. Possiamo farlo in due modi:

- A livello di *bit*: il disco  $i$ -esimo contiene i bit  $n \mid n \bmod N = i$  (ad esempio i bit pari vanno su disco 1, i bit dispari sull'altro disco)
- A livello di *blocco*: il disco  $i$ -esimo contiene i blocchi  $n \mid n \bmod N = i$  (la più *comune*)  
Ciò migliora le prestazioni, permettendo *lettura parallela*

### Definizione. (*codice di parità*)

Eventualmente con l'aggiunta di *codici di parità*

- Sono dei codici "*backup*" che servono per rilevare errori effettuando somme. In grado di ripristinare i dati, prendendo il complementare.
- Per essere *fault-tolerant*
- Non si perdono i dati in caso di guasti di un disco

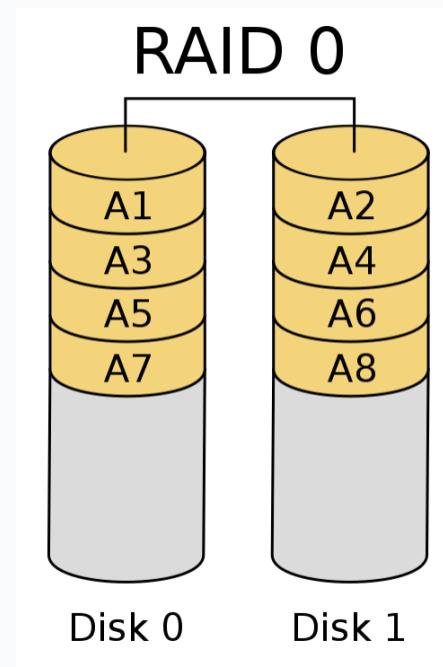
# Livelli di RAID

Ci sono *diverse configurazioni o schemi di dischi possibili*, detti "*livelli*". Differiscono da:

- A seconda che offrano aumenti di *prestazione o affidabilità*
- *Numero minimo di dischi richiesto*
- *Robustezza a guasti* multipli
- Alcune tecniche sono intuitive, altre meno.

## RAID 0. (*Selezionamento*)

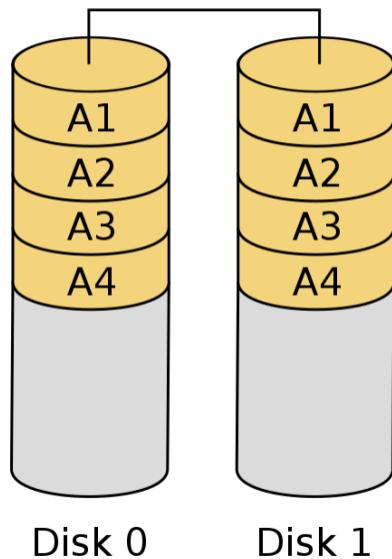
- **Idea:** I dati sono *divisi* tra i dischi tramite *striping* (a livello di blocco di solito)
- **Minimo numero di dischi:**  $\geq 2$
- **Vantaggi:** Alta velocità sia di lettura che di scrittura grazie ad accessi paralleli. Come vedremo, questo è l'*unico vantaggio* di RAID 0.
- **Svantaggi:** Decresce affidabilità del sistema; con un guasto, ho perso tutti i dati!
- **Casi d'uso:** Traffici di rete ad alta velocità (come 10Gb/s)



## RAID 1. (*Mirroring*)

- **Idea:** I dati sono *replicati* su più dischi. L'opposto di *RAID 0*.
- **Minimo numero di dischi:**  $\geq 2$
- **Vantaggi:** Con  $N$  dischi, resiste a  $N - 1$  guasti. Alte prestazioni di lettura.
- **Svantaggi:** Bassa *velocità di scrittura* limitata dal disco più lento. Prendi infatti il *minimo della velocità* tra i dischi.
- **Casi d'uso:** Server dove è necessaria una buona resistenza a guasti, oppure una alta velocità di lettura.

# RAID 1

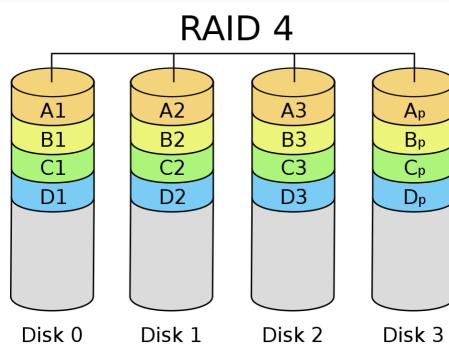


## RAID 2, 3.

- Ignorate, dato che sono assolutamente obsolete. Per un approfondimento vedere la pagina Wikipedia ([Link](#))

## RAID 4. (*Disco di parità*)

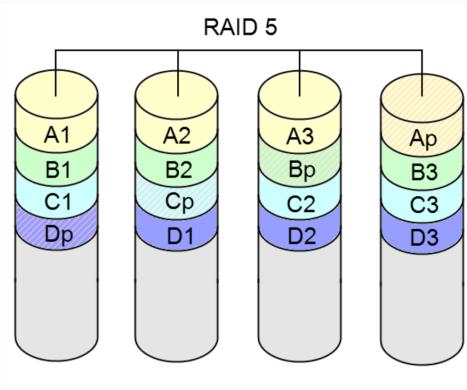
- **Idea:** Il disco  $N$  memorizza la parità dei dati sugli altri  $N - 1$  dischi
- **Minimo numero di dischi:**  $\geq 3$ . Due (o  $n + 1$ ) di dato più parità
- **Vantaggi:** Resiste a un guasto e permette letture parallele (quindi alte prestazioni di lettura).
- **Svantaggi:** Scrittura lenta. Necessario calcolare e scrivere parità. Inoltre, sollecitando sempre l'ultimo disco per la scrittura della parità, rischio sia di avere il "bottleneck" (letteralmente ingorghi) e di incrementare la probabilità del guasto.
- **Casi d'uso:** Non si usa tanto. Si userà una sua variante più furba, la RAID 5.



## RAID 5. (*Parità distribuita*)

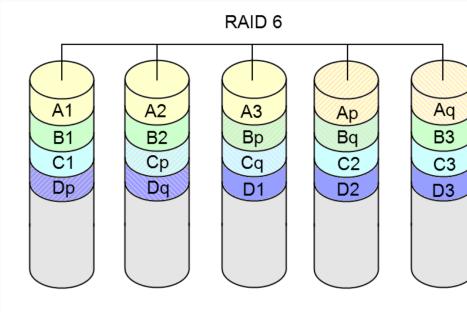
- **Idea:** Come RAID 4, ma codici di parità distribuiti su tutti i dischi equalmente
- **Minimo numero di dischi:**  $\geq 3$
- **Vantaggi:** Sostanzialmente ci sono solo vantaggi rispetto a RAID 3: resiste a un guasto. Scritture più veloci di RAID 4: non è necessario accedere sempre a disco di parità, rimuovendo eventuali bottleneck.

- **Svantaggi:** Scrittura comunque lenta (a causa di parità)
- **Casi d'uso:** Molto usato in sistemi reali



## RAID 6 (*Doppia parità distribuita*)

- **Idea:** Codici di parità memorizzati due volte. Tra tutti i dischi. Ovvero, per ogni riga *dedico due dischi per la parità*.
- **Minimo numero di dischi:**  $\geq 4$
- **Vantaggi:** Resiste a *due guasti*
- **Svantaggi:** Scrittura molto lenta (a causa di doppia parità)
- **Casi d'uso:** Molto usato in sistemi reali, in particolare dove *abbiamo dati sensibili*, quindi abbiamo bisogno di una garanzia maggiore.



# RAID: Conclusioni

## Riassunto RAID.

Gli schemi RAID permettono di migliorare *prestazioni e affidabilità* quando si hanno molti dischi su una stessa macchina

## Limiti della RAID.

1. Non proteggono da un *failure completo della macchina*: eventualmente alcuni livelli proteggono le *failure dei dischi*. Come ad esempio abbiamo
  - *Temporaneo*: manca la corrente
  - *Permanente*: si rompe la scheda madre
- Ciò non è accettabile per servizi *mission-critical* (esempio: le *ASL*)
2. Le tecniche RAID *non scalano*:

- C'è un *massimo numero di dischi collegabili* a una macchina
- Il *BUS PCI ha un limite* (di circa 36 Gb/s)

Per sistemi *molti grandi* (quindi ancora più grossi) si usano *File System distribuiti*

---

*Soluzione FDS: Introduzione (collegamento via rete), approcci per la FDS (NFS, NBD). Definizione di File System Distribuito, software orchestratore. Tecnologie attuali di FDS: HDFS e CEPH.*

---

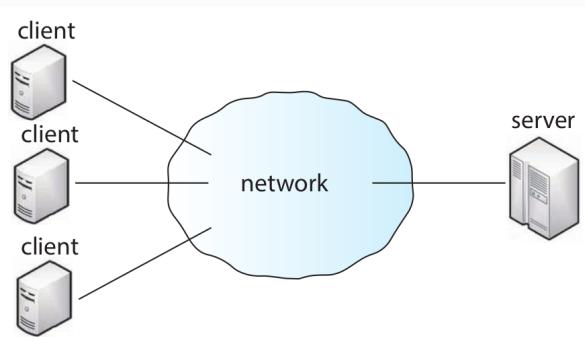
## File System Distribuiti: Introduzione

**Problema.** (*File System Distribuiti*)

E' possibile usare un FS che *si trova su un'altra macchina*

- Tipicamente un *server dedicato allo storage*
- Si utilizzano *protocolli dedicati*
  - **Network File System (NFS)**: il più *usato e flessibile*
  - **Samba**: Microsoft (anche se di dubbia reputazione)
  - **File Transfer Protocol (FTP)**: obsoleto, purtroppo in certi casi ancora utilizzata
  - **Fiber Channel**: per grandi *Storage Area Network* (in declino, oggi si preferisce usare i file server normali)

**Figura.** (*L'idea del FSD*)

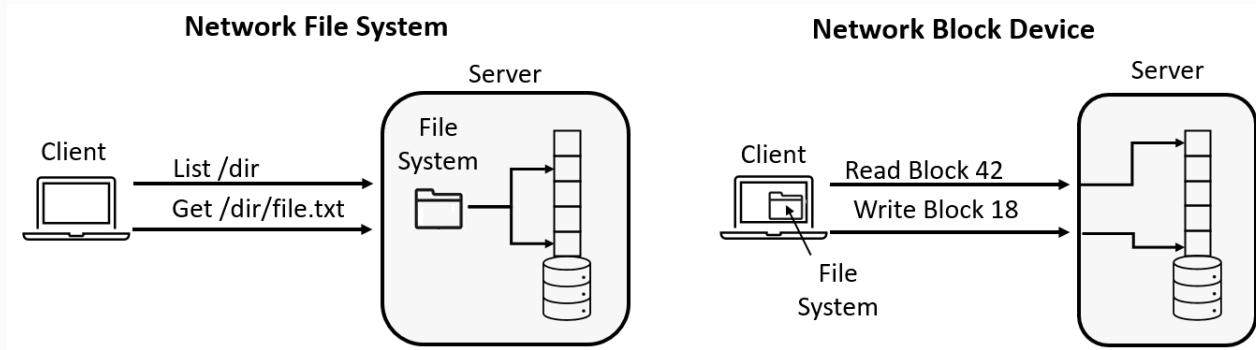


## Approcci per i File System Distribuiti

Abbiamo *due approcci* per i *file system distribuiti*, che rispondono alla domanda: "*Dove poniamo il File System*"? Quindi questi approcci sono concettualmente diversi:

- *File System di Rete* (o Network File System): FS gira sul server
  - *Dispositivi a Blocchi di Rete* (o Network Block Device): FS gira sul client
- Vedremo che ognuna ha i suoi vantaggi e svantaggi.

**Figura.** (L'idea del NFS e NBD)



### Network File System

- **Vantaggi:** Possono collegarsi più *client in parallelo*, possiamo svolgere *operazioni di alto livello*..
- **Svantaggi:** Carico troppo sul *file system del server*, soprattutto quando ho più *clienti in parallelo*. Problemi di scalabilità.

### Network Block Device

- **Vantaggi:** Possiamo svolgere *operazioni di basso livello*, come ad esempio decidere quale file system usare. Necessario in certi casi, come ad esempio usare il database *MYSQL*.
- **Svantaggi:** Può girarci *solo ed esclusivamente solo* un client.

---

## File System Distribuiti: Definizione

**Definizione.** (*File System Distribuito, software orchestratore*)

Un *File System Distribuito* è un file system che risiede *su più dischi su macchine diverse*

- E' necessario un software *orchestratore*
- Per far sì che l'utilizzatore ne fruisca come un unico FS
  - Quindi è *molto complesso*, dato che deve dare l'*illusione* di essere un *unico FS*.

Un FS distribuito:

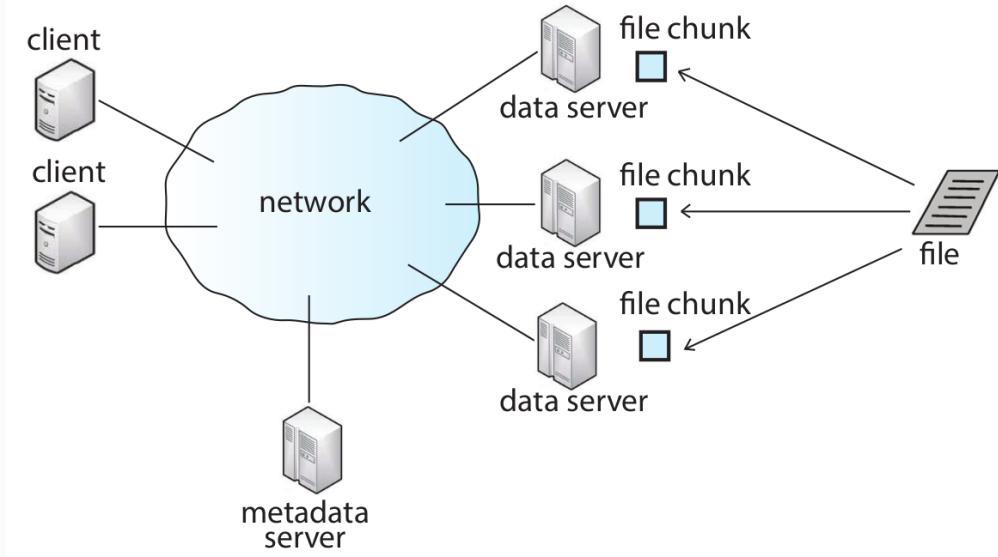
- E' visto da utilizzatori come un unico FS *grande e affidabile*
- Vi si accede tipicamente come disco di rete (è un File System di Rete)

**Modello.** (*Client-Server*)

Basati su modello *client-server*

- Client consulta il *metadata server* per listare directory e ottenere *informazioni sui file*
- Client accede al *contenuto da uno o più data server*

**Figura.** (L'idea)



## Tecnologie per FS distribuiti

I FS distribuiti si installano con *software di orchestrazione* dedicati

- Organizzano i dati nei vari dischi e nodi
  - Replicano i dati per aumentare le prestazioni
  - Recuperano i dati quando un utilizzatore vi accede
- Adesso ne studiamo alcuni.

### 1. Hadoop Distributed File System (HDFS)

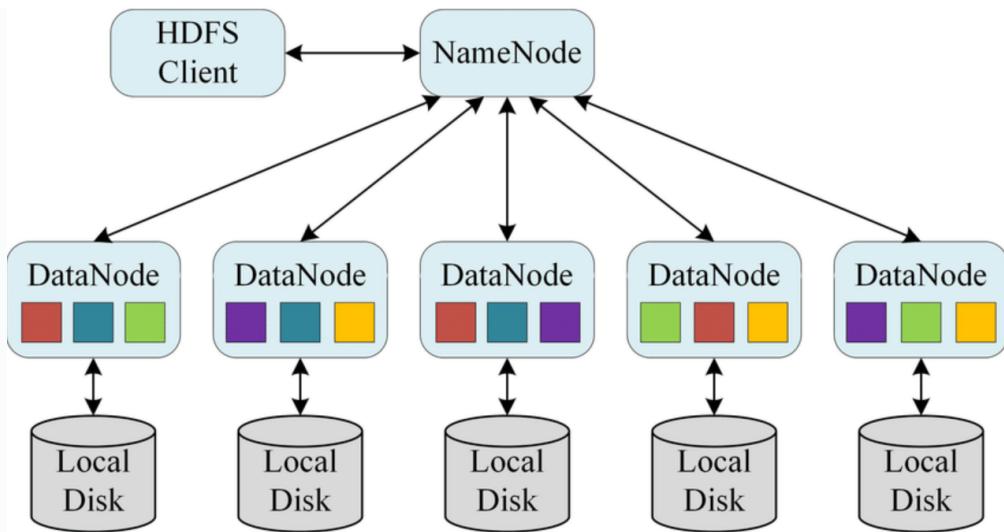
Parte della suite *Hadoop per Big Data*. E' un FS distribuito

- Si installa su un *cluster* (insieme) di server/nodi
- I *Name Node* hanno l'indice dei *file*
- I *Data Node* memorizzano il contenuto dei *file*
- Tutto viene replicato  $N$  volte

**Vantaggi:** Funziona benissimo, infatti viene utilizzato quasi da tutti (banche, eccetera...)

**Svantaggi:** Problemi di scalabilità: infatti in realtà HDFS è in declino e sta vivendo di "*rendita aziendale*", ovvero viene usata solo perché le aziende non possono o non vogliono cambiare la tecnologia per FSD.

**Figura.** (\*Schema HDFS)

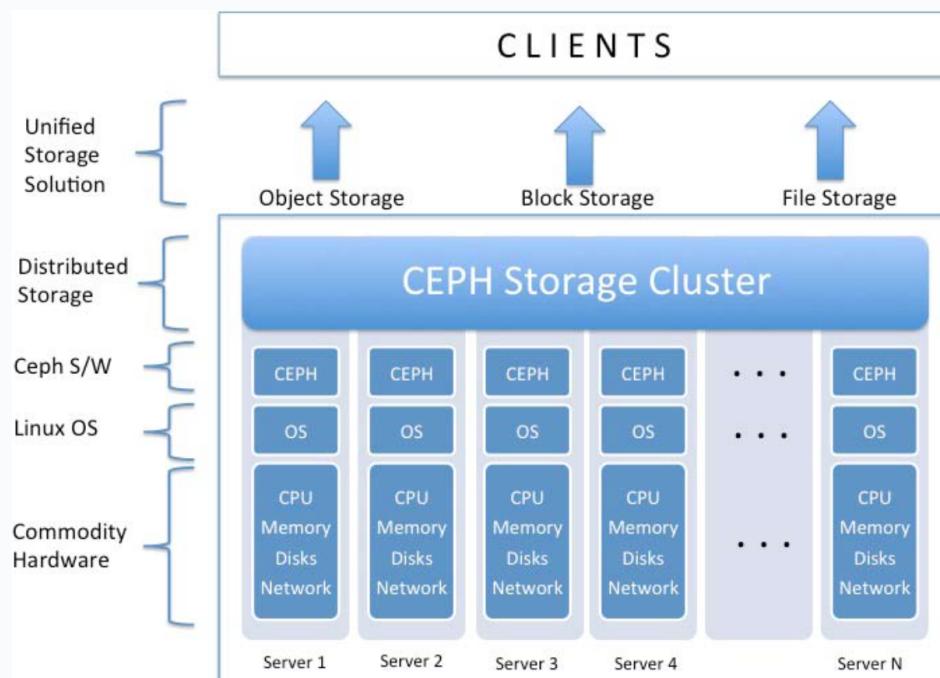


## 2. CEPH

*Concettualmente simile a HDFS.* Si usa su cluster di nodi. Implementa nuovi concetti più moderni, tra cui:

- **FS distribuito:** i client accedono a *file e cartelle*
- **Dispositivo a blocchi:** i client vedono *disco grezzo a blocchi*
- **Object storage:** i client accedono a *bucket generici* identificati da ID
- **Vantaggi:** Molto flessibile, infatti permette di avere sia *Newtork FS* che *Network BD*. Inoltre, abbiamo già implementato un *database molto efficiente*. La tecnologia **CEPH** è "in rising".

**Figura. (CEPH)**



## Domande

Un sistema di dischi basato su RAID è sempre più affidabile di un disco singolo?

- Si
- No

In un sistema RAID 0, quali sono le conseguenze in caso di fallimento di un disco?

- I dati vengono persi
- E' possibile recuperare i dati

In un sistema RAID 1, quali sono le conseguenze in caso di fallimento di un disco?

- I dati vengono persi
- E' possibile recuperare i dati

E' possibile creare un sistema RAID 6 con 3 dischi?

- Si
- No

Qual è il sistema di accesso tipico a un FS Distribuito?

- Bus PCI
- Rete
- USB