

# Sistemi Operativi

Laurea in Ingegneria Informatica

Università di Roma Tor Vergata



Docente: Francesco Quaglia

## Virtual file system

1. Nozioni preliminari
2. Architettura di base e funzioni
3. Gestione dei file
4. Utenze e gruppi
5. Gestione dei dispositivi fisici
6. Virtual file system in sistemi operativi attuali (UNIX/Windows)

# Virtual file system – concetti basici

- È costituito da tutti i moduli di livello kernel che supportano operazioni di I/O
- Queste avvengono secondo uno schema omogeneo (stesse system call) indipendentemente da quale sia l'oggetto di I/O coinvolto nell'operazione
- Si basano su modelli di riferimento quali:
  - ✓ stream I/O
  - ✓ block I/O
- System call ad-hoc esistono quindi solo in relazione all'istansiazione dei vari oggetti di I/O (non alle vere e proprie operazioni su di essi)

# Overview

Istanziamento/eliminazione  
di oggetti di I/O (**interfacce  
specifiche per tipologie di  
oggetti**)

Files  
Pipes  
FIFOs  
Mailslots  
Sockets  
Char devices  
Block devices

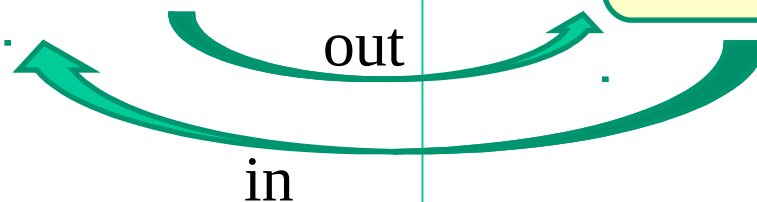
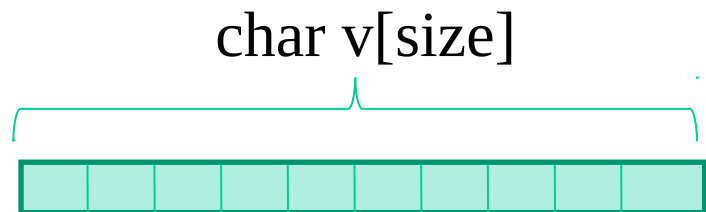
UNIX and/or  
Windows

Reali operazioni di I/O  
(**interfaccia comune**)

Lecture  
Scritture  
Ripozionamenti  
Eliminazione di contenuti (se non volatili)

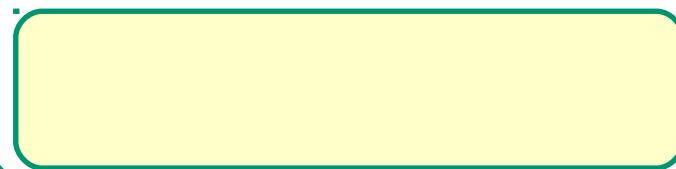
# Stream/block I/O model

## User space memory



## Kernel space memory (RAM storage)

Struttura dati che implementa lo specifico oggetto di I/O (inclusi metadati)



Backend hardware device, if any, where the I/O object data are flushed or taken from, e.g.

- ✓ hard disks
- ✓ network interfaces

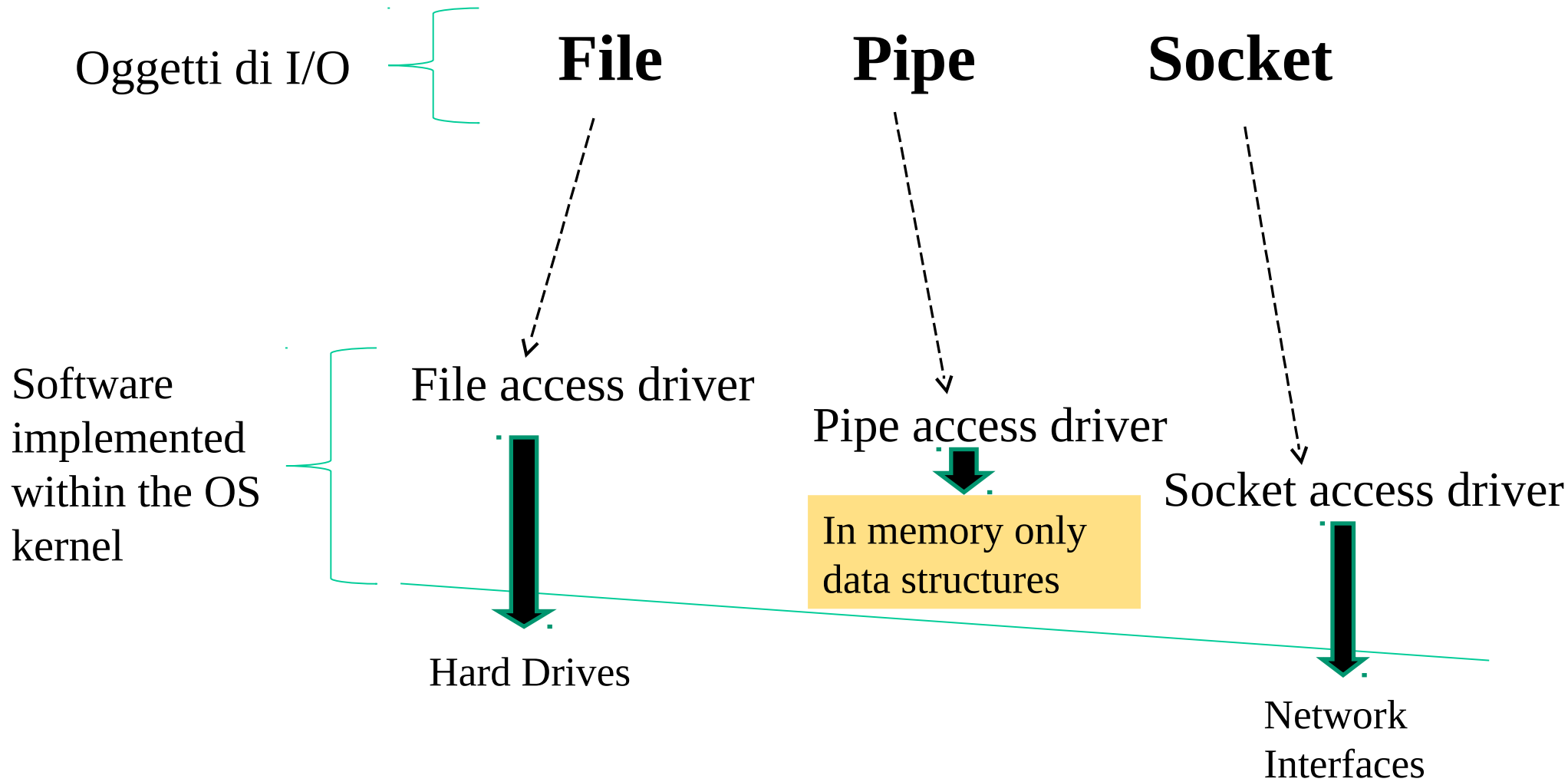
Stream I/O può portare a letture di **frazioni arbitrarie di dati** scritti in precedenza

Block I/O porta a letture di **unità di dati** scritte in precedenza

# Drivers

- Tipi di oggetti di I/O differenti sono generalmente rappresentati tramite strutture dati differenti
- Inoltre non tutti i tipi di oggetti di I/O hanno una rappresentazione di backend (volatile o non) su dispositivi hardware
- Inoltre le eventuali rappresentazioni di backend possono afferire a dispositivi hardware differenti
- Tutta questa eterogeneità è risolta in modo del tutto trasparente al codice applicativo tramite il concetto di driver
- Il driver è l'insieme di moduli software di livello kernel per eseguire le reali operazioni afferenti ad un qualsiasi oggetto di I/O
- Ogni tipologia di oggetti ha la sua tipologia di driver

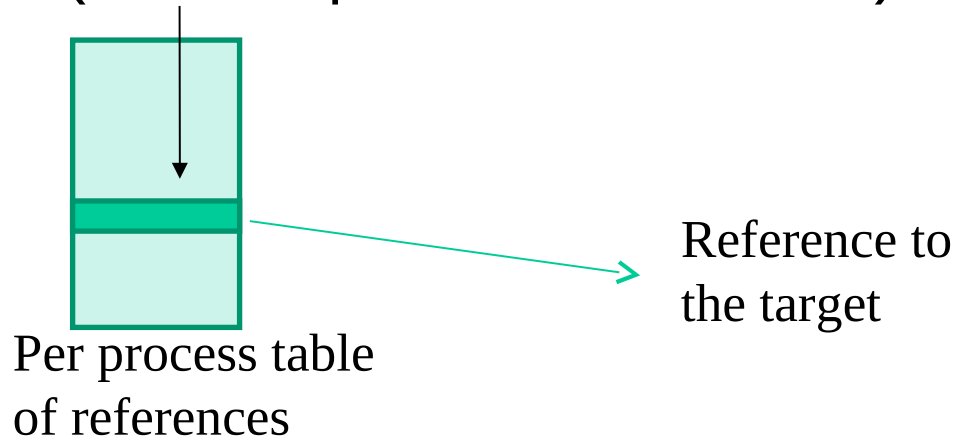
# Uno schema



# Canali di I/O

- Sono identificatori logici per eseguire operazioni di I/O sugli oggetti
- Ovvero chiavi di accesso all'istanza di oggetto di I/O
- Il setup del canale di I/O richiede istansiazione e/o apertura dell'oggetto
- I canali di I/O portano il kernel a riconoscere l'istanza di oggetto target tramite la tabella degli oggetti accessibili al processo

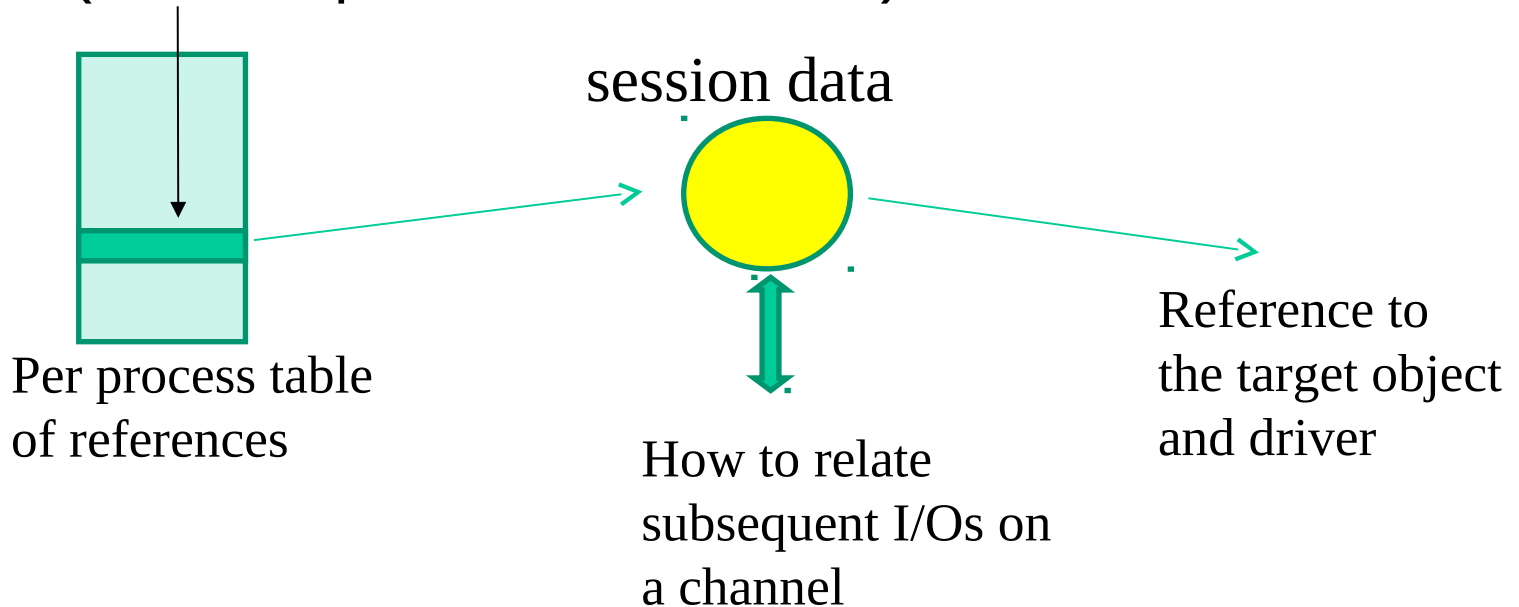
`I/O_Syscall(descriptor/handle .....`



# Sessioni di I/O

- Il setup del canale di I/O porta automaticamente al setup della così detta sessione di lavoro (sessione di I/O) sull'oggetto target
- Questa mantiene dati temporanei relativi alle operazioni che vengono eseguite sul canale di I/O

`I/O_Syscall(descriptor/handle .....`





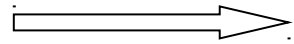
# I/O e archiviazione - il file system

## Il punto di vista del sistema operativo

- minima unità informativa archiviabile: **il file**
- informazioni in archivio (ovvero su dispositivi di memoria di massa), potranno essere contenute esclusivamente all'interno di un file

## Il punto di vista delle applicazioni

- minima unità informativa accessibile (manipolabile): **il record**



Il file system mostra alle applicazioni ogni singolo file come una semplice sequenza di records

---

## Un file system associa ad ogni file un insieme di attributi

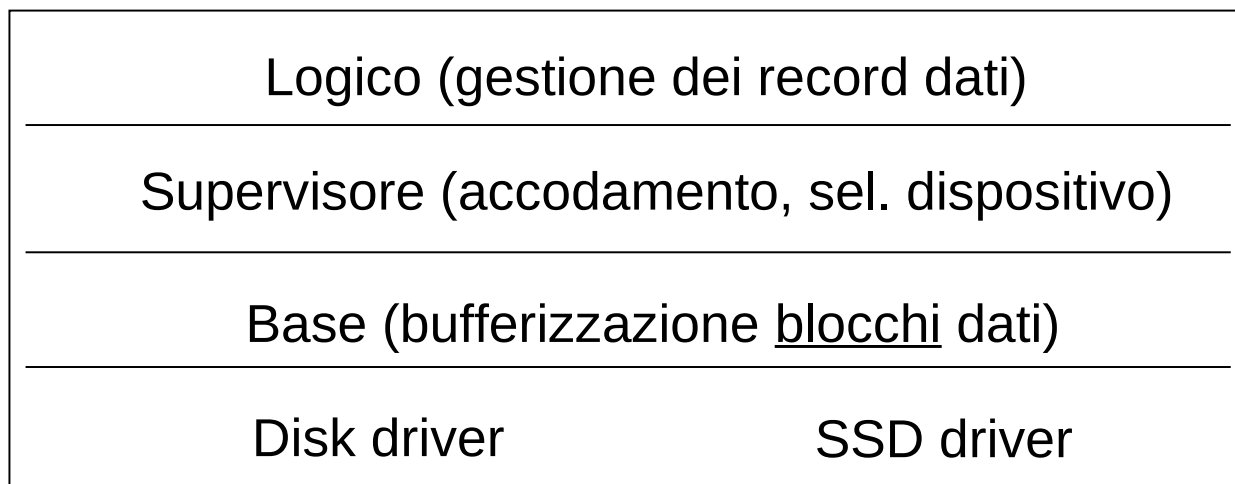
- Nome (identificazione univoca)
- Protezione (controllo sugli accessi)
- Altro, e.g. timestamp, bounds (dipendendo dallo specifico file system)

# Architettura di base di un file system

Applicazioni

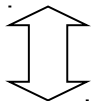
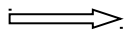
interfaccia (I/O system calls)

Livelli

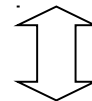


**File system drivers**

poggia su



setup/management



Disk controller

SSD controller

# Operazioni base sui file

## **Creazione**

- allocazione di un “record di sistema” (RS) per il mantenimento di informazioni relative al file (e.g. attributi) durante il suo tempo di vita

## **Scrittura/Lettura (di record)**

- aggiornamento di un indice (puntatore) di scrittura/lettura valido per sessione

## **Apertura (su file esistenti)**

- inizializzazione dell'indice di scrittura/lettura per la sessione corrente

## **Chiusura**

- rilascio dell'indice di scrittura/lettura

## **Riposizionamento**

- aggiornameno dell'indice di scrittura/lettura

## **Eliminazione**

- deallocazione di RS e rilascio di memoria (blocchi dati) sul dispositivo

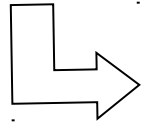
## **Troncamento**

- rilascio di memoria (blocchi dati) sul dispositivo

# Indici di scrittura/lettura

- L'indice di scrittura/lettura **NON** fa parte di RS (accessi concorrenti su punti del file scorrelati)

- L'indice di scrittura/lettura **PUÒ** essere condiviso da più processi

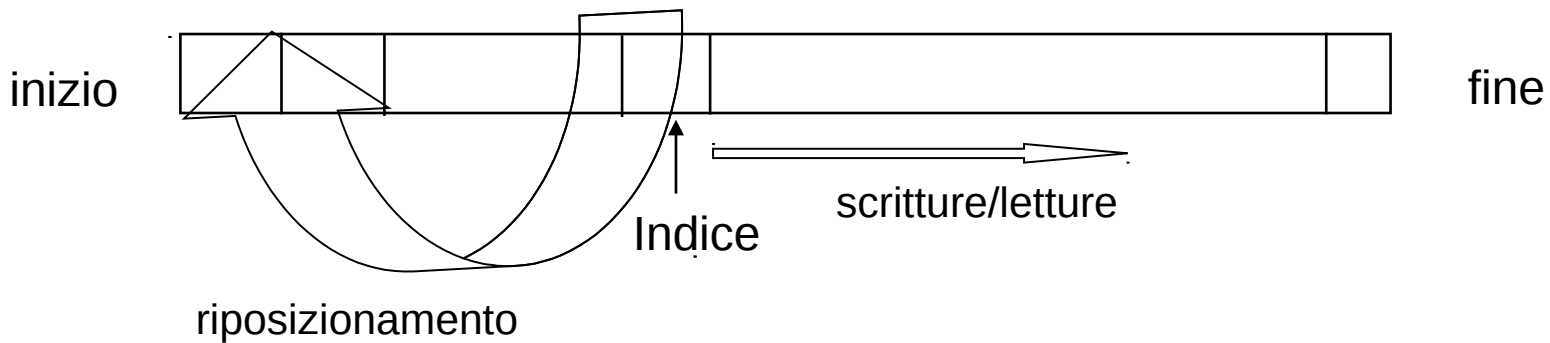


- ✓ quindi **NON** fa parte della singola immagine di processo mantenuta dal sistema operativo
- ✓ fa tipicamente parte dell'immagine di sessione

- Le modalità di aggiornamento dell'indice di scrittura/lettura **in riposizionamento** dipendono dai metodi di accesso ai record di un file supportati dallo specifico file system

# Metodo di accesso sequenziale

- i records vengono acceduti sequenzialmente
- l'indice di scrittura/lettura è incrementato di una unità per ogni record acceduto
- il riposizionamento dell'indice può avvenire solo all'inizio del file



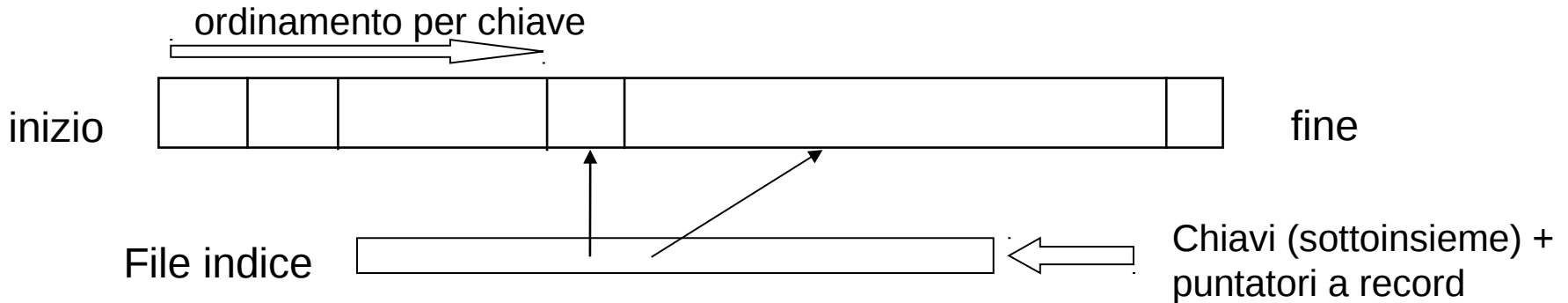
## Tipico di:

- **File sequenziali**, caratterizzati da record di taglia e struttura fissa
- **File a mucchio**, caratterizzati da record di taglia e struttura variabile (ogni record mantiene informazioni esplicite su taglia e struttura)

# Metodo di accesso sequenziale indicizzato

## Tipico di:

- **File sequenziali indicizzati**, caratterizzati da record di taglia e struttura fissa, ordinati in base ad un campo chiave



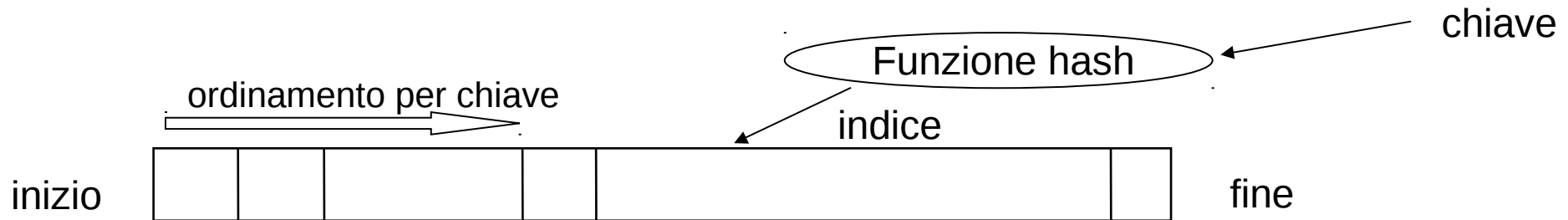
- esiste un file sequenziale di indici associato a ciascun file di dati
- i record sono ordinati per “chiave”
- tramite il file di indici ci si può posizionare in punti specifici del file di dati (ovvero in punti con valori specifici del campo chiave)
- i records vengono acceduti sequenzialmente una volta posizionati sui punti stabiliti
- l'indice di scrittura/lettura è manipolato (incrementato) di conseguenza
- il riposizionamento dell'indice può avvenire solo all'inizio del file

# Metodo di accesso diretto

- il riposizionamento dell'indice può avvenire in un qualsiasi punto del file
- si può accedere direttamente all' $i$ -esimo record (senza necessariamente accedere ai precedenti)
- dopo un accesso all' $i$ -esimo record, l'indice di scrittura/lettura assume il valore  $i+1$

## Tipico di:

- **File diretti**, caratterizzati da record di taglia e struttura fissa
- **File hash**, caratterizzati da record di taglia e struttura fissa con ordinamento per chiave



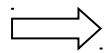
# Struttura di directory

- La directory è un file ``speciale``
- Essa contiene informazioni per poter accedere a file veri e propri, contenenti record di dati
- Il modo con cui le informazioni vengono mantenute nelle directory (ovvero nei file associati alle directory) determinano la così detta **struttura di directory**

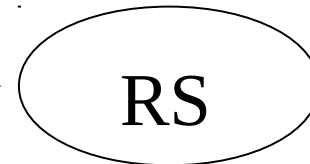
## Tipica struttura di directory

- nomi dei file contenuti nella directory
- informazioni di identificazione dei RS associati ai file

File associato  
alla directory



Nome del file	





# File vs blocchi di dispositivo (i)

- Ciascun file è allocato sul dispositivo di memoria di massa come un insieme di blocchi (chiamati anche blocchi logici) non necessariamente contigui
  1. organizzazione fissa
    - record di taglia fissa - possibilità di frammentazione interna
  2. organizzazione variabile
    - record di taglia variabile con possibilità che un record sia suddiviso o non tra più blocchi

# File vs blocchi di dispositivo (ii)

- Le unità di allocazione possono essere costituite da più blocchi (per motivi di efficienza)
- RS tiene traccia di quanti e quali blocchi sono allocati per un dato file
- Il file system tiene conto degli spazi liberi sul dispositivo di memoria di massa tramite apposite strutture:
  - A) Lista Libera: tiene traccia di unità di allocazione libere
  - B) Bit Map: dedica un bit ad ogni unità di allocazione, per indicare se essa è libera o meno

# Lista libera e bit-map – esempi per Hard Disks

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

Track	Sector											
	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

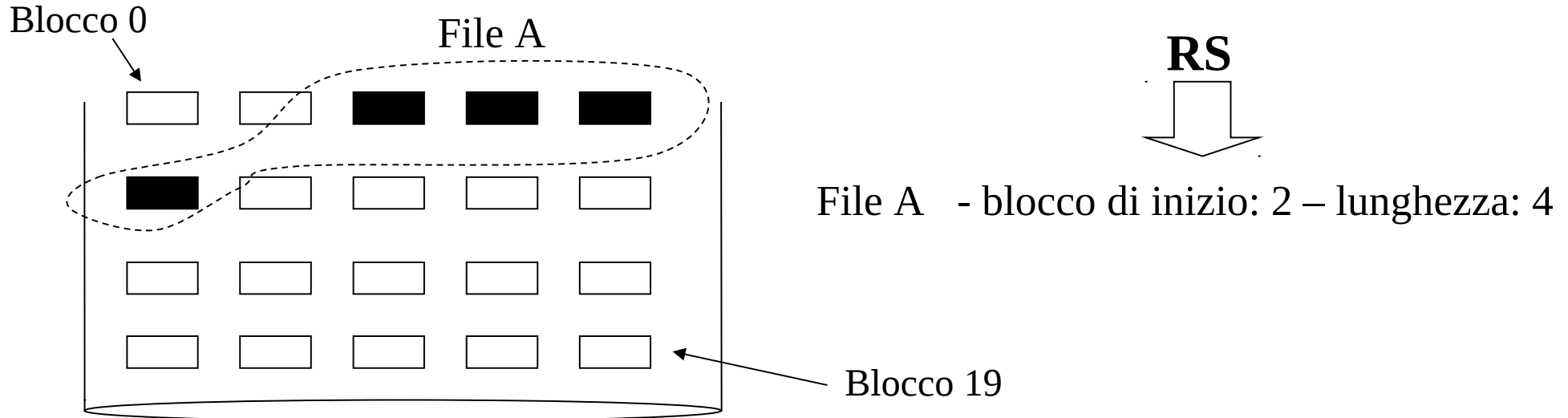
(b)

(a) Lista Libera

(b) Bit Map

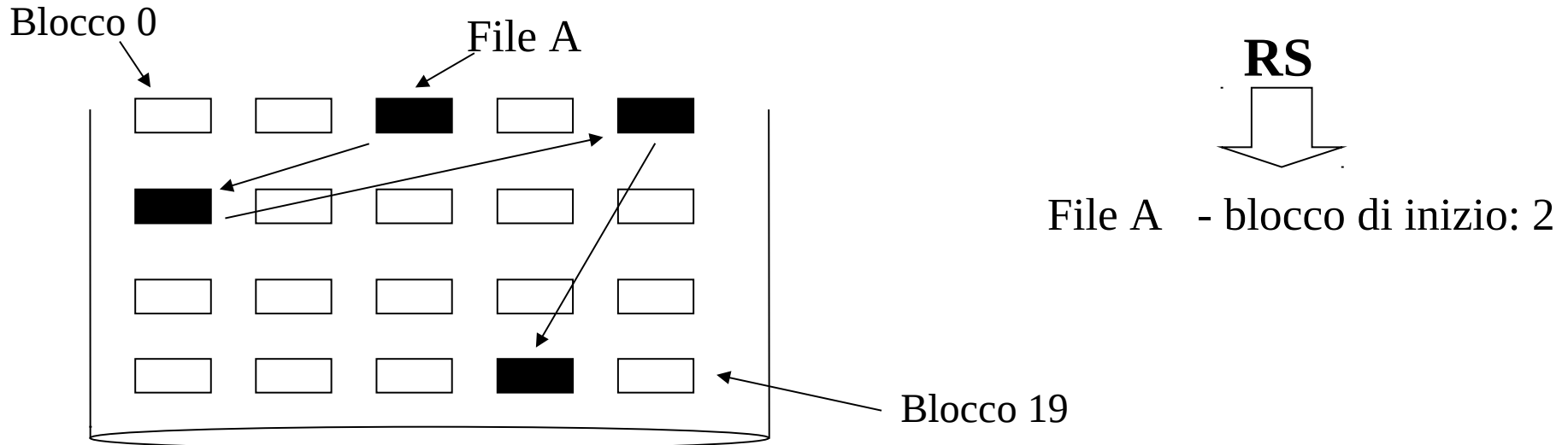
# Allocazione di file contigua

- un insieme di blocchi contigui è allocato per un file all'atto della creazione
- la taglia massima dipende dal numero di blocchi allocati
- l'occupazione reale è sempre pari al numero di blocchi allocati (anche se essi non contengono record del file)
- RS mantiene informazioni sul primo blocco e sul numero di blocchi
- ricompattazione per affrontare il problema della frammentazione esterna



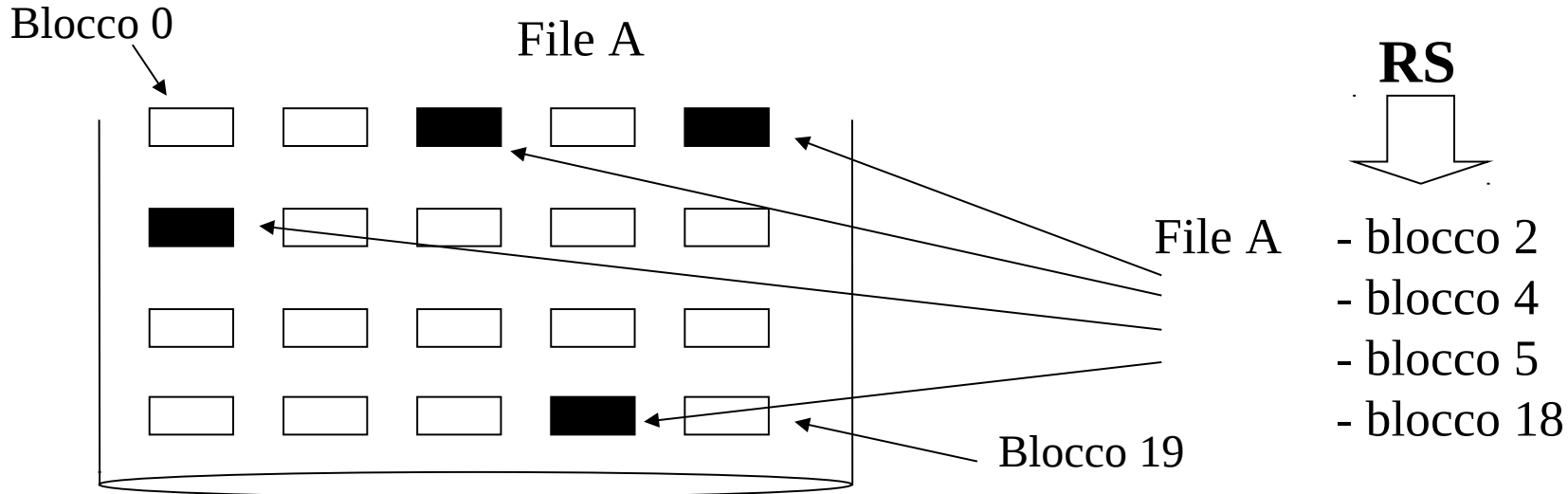
# Allocazione di file a catena

- i blocchi di un file sono collegati come in una lista
- l'occupazione reale e' sempre pari al numero di blocchi realmente nella lista
- RS mantiene informazioni sul primo blocco
- accesso potenzialmente costoso
- ricompattazione utile per diminuire il costo di accesso

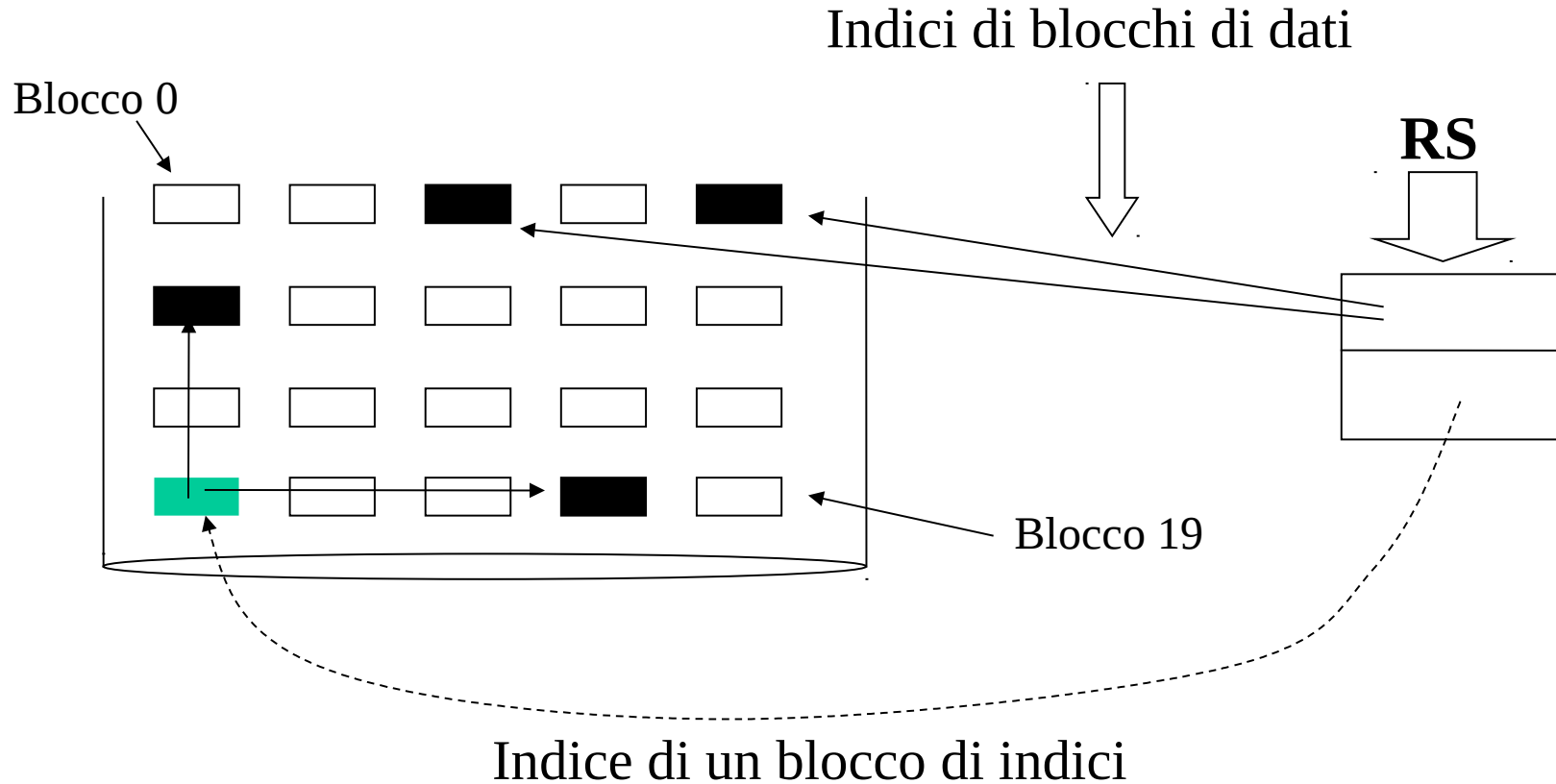


# Allocazione di file indicizzata

- i blocchi di un file sono rintracciati tramite un indice
- l'occupazione reale è sempre pari al numero di blocchi realmente allocati per record del file
- RS mantiene informazioni sugli indici dei blocchi (maggiore occupazione di spazio per RS rispetto ad altri schemi)



# Indicizzazione a livelli multipli



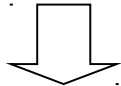
**Indici di blocchi di dati possono anche non essere presenti in RS**

# Cache dei dispositivi

- il sistema operativo mantiene una regione di memoria che funge da buffer temporaneo (**buffer cache**) per i dati acceduti in I/O **e che siano riaccessibili**
- hit nel buffer cache evita interazione con gli hard-drive (diminuzione della latenza e del carico sugli hard-drive)
- efficienza legata alla località delle applicazioni (**lettura anticipata/ scrittura ritardata**)

## Strategia di sostituzione dei blocchi

- Least-Recently-Used: viene mantenuta una lista di gestione a stack
- Least-Frequently-Used: si mantiene un contatore di riferimenti al blocco indicante il numero di accessi da quando è stato caricato

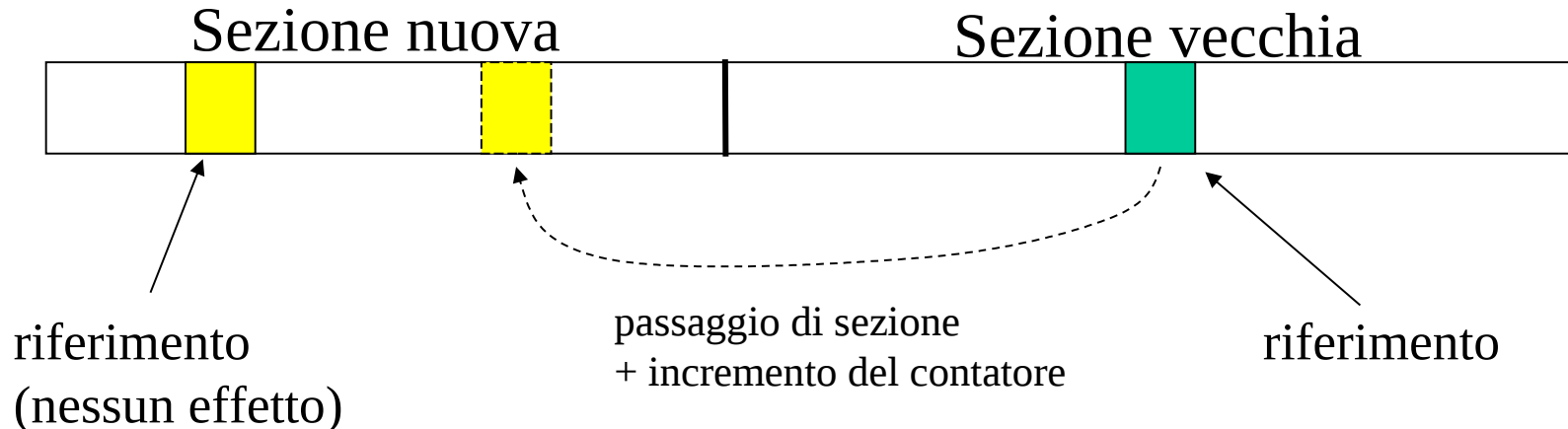


Problemi sulla variazione di località



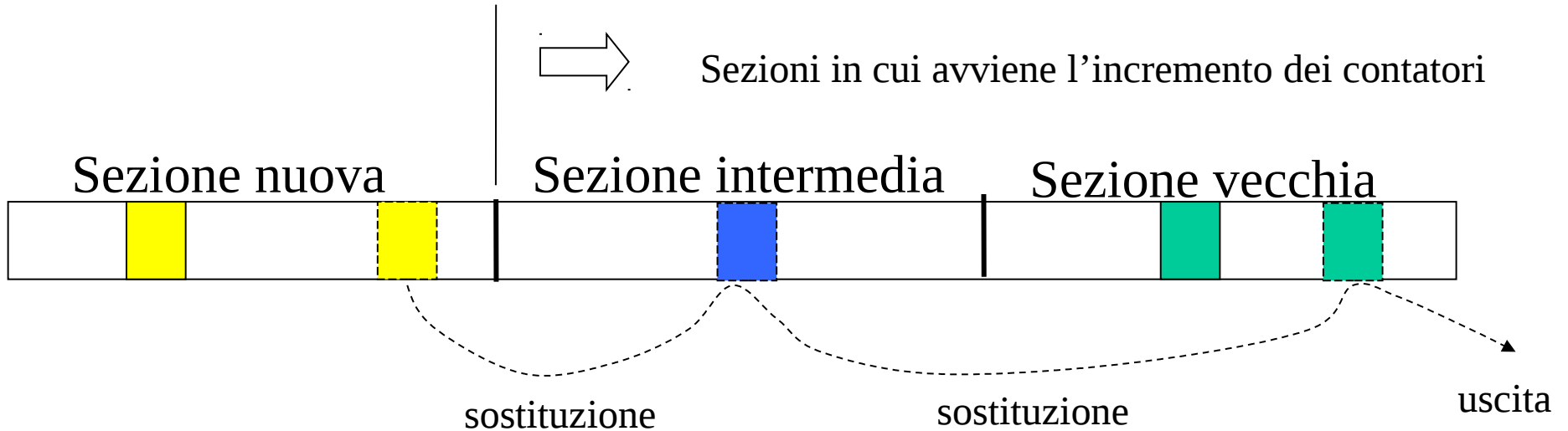
# Buffer cache a due sezioni

- ogni volta che un blocco è riferito, il suo contatore di riferimenti è incrementato ed il blocco è portato nella **sezione nuova**
- per i blocchi nella sezione nuova il contatore di riferimenti non viene incrementato
- per la sostituzione dalla sezione nuova si sceglie il blocco con il numero di riferimenti minore
- la stessa politica è usata per la sostituzione nella **sezione vecchia**



# Buffer cache a tre sezioni

- esiste una sezione intermedia
- i blocchi della sezione intermedia vengono passati nella sezione vecchia per effetto della politica di sostituzione
- un blocco della sezione nuova difficilmente verrà escluso dal buffer cache in caso sia riferito in breve tempo dall'uscita dalla sezione nuova



# I/O e swapping

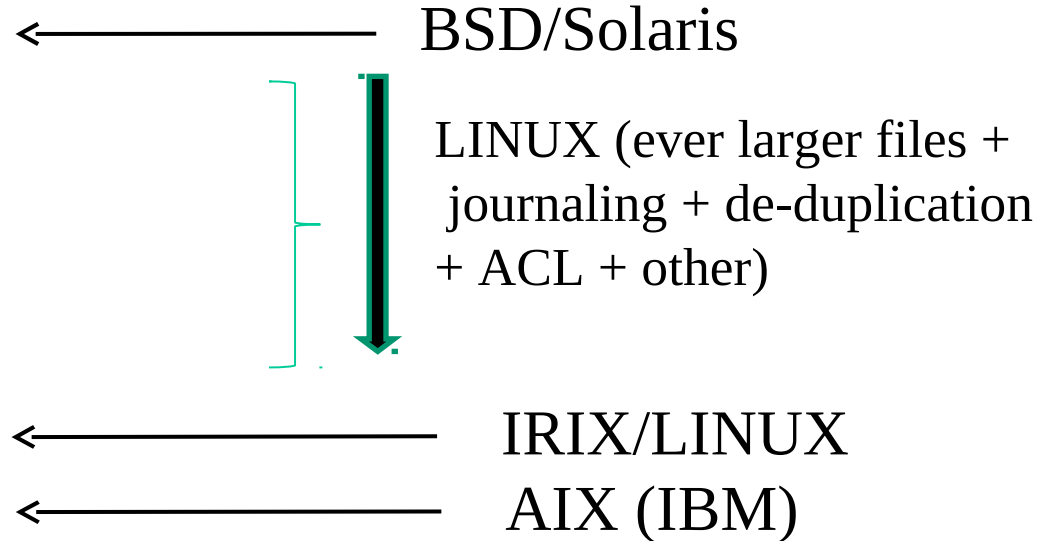
## Collocamento dell'area di swap

- file system
  1. Funzioni di gestione del file system vengono usate anche per l'area di swap
  2. Possibilità di gestione inefficiente (tradeoff spazio-tempo)
  3. Taglia dell'area non predefinita
- partizione privata
  1. Esiste un gestore dell'area di swap
  2. La disposizione sul dispositivo può essere tale da ottimizzare l'accesso in termini di velocità
  2. Ammessa la possibilità di elevata frammentazione interna
  3. Taglia dell'area predefinita

# UNIX file systems

- Berkeley Fast File System also known as UFS (Unix File System)

- Ext2
- Ext3
- Ext4
- Btrfs (Better File System)
- XFS
- JFS (Journal FS)

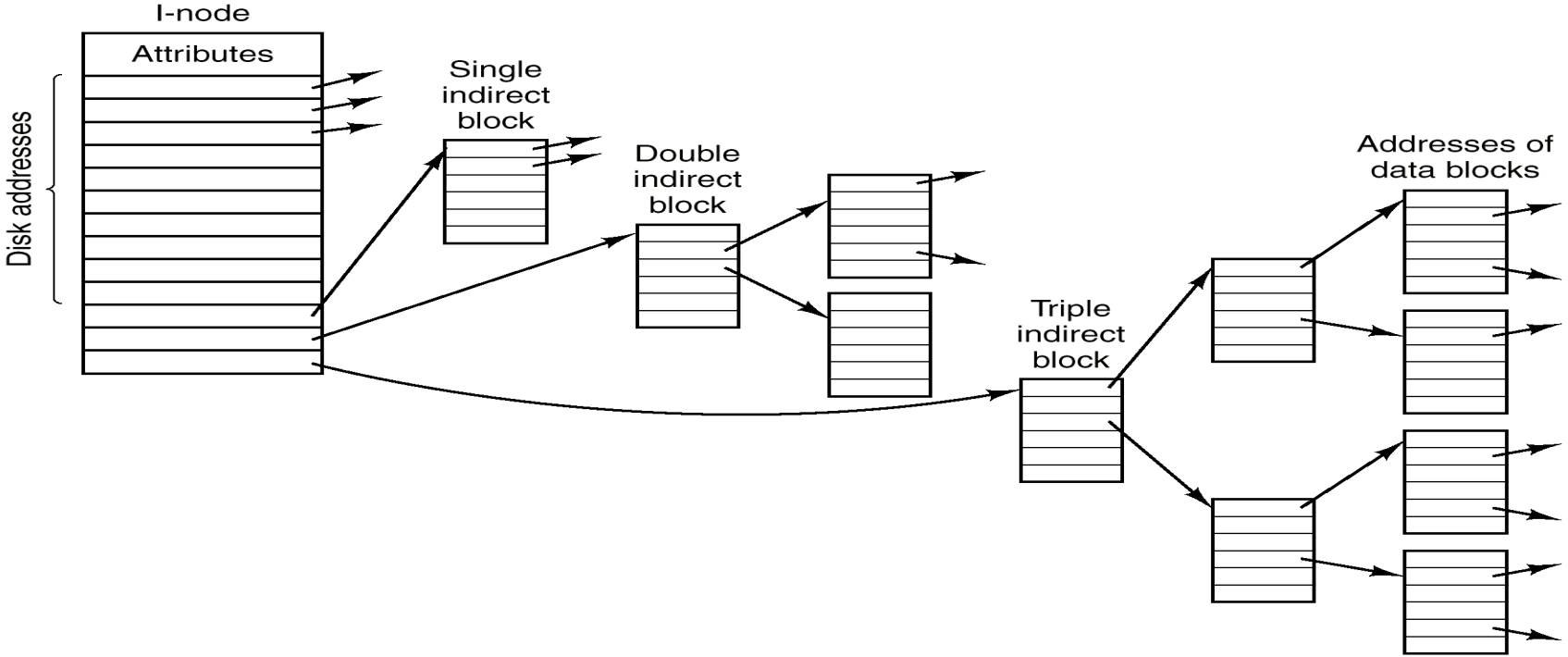


# Caratteristiche di base dei file system UNIX

- Ogni file è trattato dal file system come una semplice sequenza di bytes (stream)
- Metodo di accesso diretto
- L'RS viene denominato **i-node**
- Esiste un vettore di i-nodes di dimensione fissata
- Organizzazione gerarchica dell'archivio
  - 1) la directory associa ad ogni file il numero di i-node corrispondente
  - 2) struttura di directory-entry: numero di i-node – spiazamento per la entry – successiva – lunghezza del nome – nome
- Bit map per blocchi dati e per i-nodes



# Classica struttura di un i-node



# Taglia massima di un file - un esempio

## Parametri (dipendenti dalla versione di file system ed hard drive)

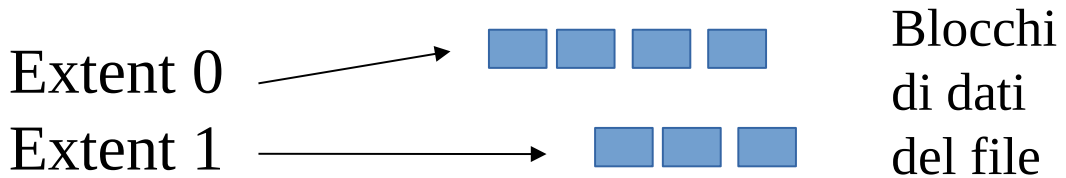
- Blocchi su disco da 512 byte
- Indirizzi su disco 4 byte
- In un blocco:  $512/4 = 128$  indirizzi
  - ind. 1-10            10    blocchi dati
  - ind. 11            128    blocchi dati
  - ind. 12             $128^2$     blocchi dati
  - ind. 13             $128^3$     blocchi dati

$$\begin{aligned}\text{Maxfile} &= 10 + 128 + 128^2 + 128^3 = \\ &= 2.113.664 \text{ blocchi} * 512 \text{ bytes} = \\ &= 1.082.195.968 \approx 1\text{Gbyte.}\end{aligned}$$

# Ext4 - extents

- le extents sono un modo per indicizzare blocchi multipli contigui (nel file e su dispositivo) con un unica struttura indice – la taglia attuale è fino a 128 MB
- un i-node ha quindi una serie di indici che possono essere
  - Diretti a sequenze di blocchi dati (ultimo livello) – questa è una extent
  - Indiretti a blocchi di indici (livelli superiori)
- La struttura di indicizzazione indiretta di fatto diventa un albero

## Un esempio



Le extent possono essere in un i-node o in un blocco intermedio indicizzato



# Attributi basici

{-,d,b,c,p}

**tipologia di file**: normale, directory, block-device, character-device, pipe

---

UID (2/4 byte)

GID (2/4 byte)

**identificatori** del proprietario e del suo gruppo

---

rwx rwx rwx

**permessi di accesso** per proprietario, gruppo, altri (codifica ottale)

---

SUID (1 bit)

SGID (1 bit)

**specifica di identificazione dinamica**  
per chi utilizza il file

---

Sticky (1 bit)

**per le directory** rimuove la possibilità di cancellare files se non si è l'owner

# Associazione userid-username

- Questa associazione non è una specifica presente nei metadati di gestione dei file a livello del file system
- Essa è solo una sovrastruttura utilizzata per comodità
- Tale associazione viene quindi realizzata tramite informazioni di mapping che associano userid e username, le quali vengono mantenute all'interno di specifici file
- Uno di questi è il file delle utenze `/etc/passwd`
- La stessa cosa è vera per l'identificativo del gruppo ed il nome del gruppo (si veda il file `/etc/group`)

# User-id/group-id e processi

- Ad ogni processo viene associato un insieme di identificatori numerici
- Questi identificatori sono relativi ad utenze e gruppi
- Questi identificatori vengono controllati quando si cerca di accedere a file o lanciare programmi tramite incrocio con i permessi di accesso
- Servizi base di gestione degli identificatori utente:

`uid_t getuid()` ← Accessibile a tutte le utenze

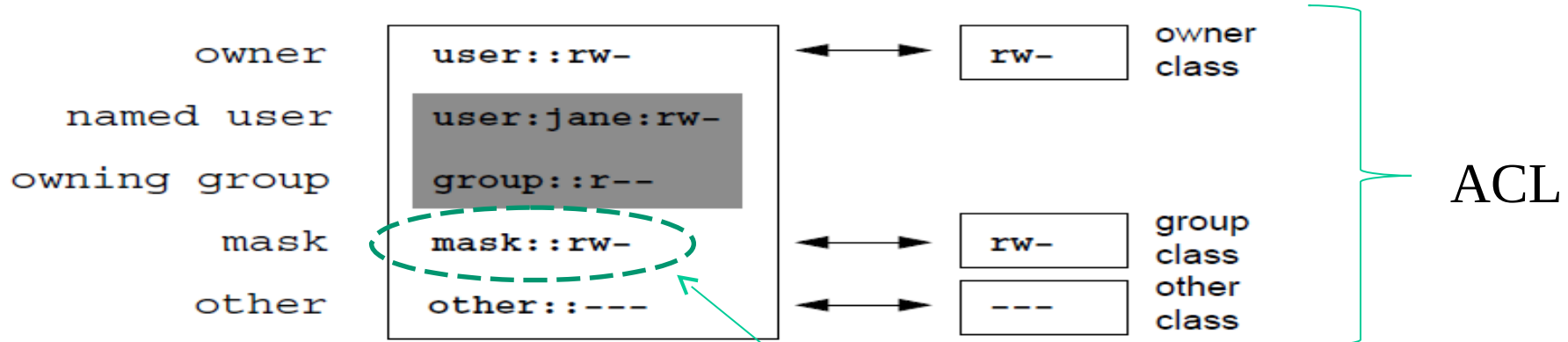
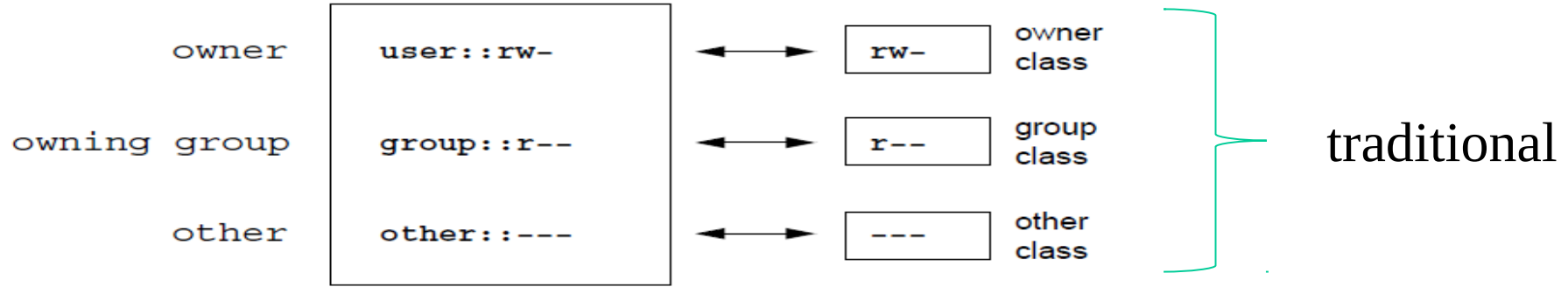
`uid_t geteuid()` ← Accessibile a tutte le utenze

`int setuid(uid_t)` ← Accessibile a “euid” root

# ACL (Access Control List)

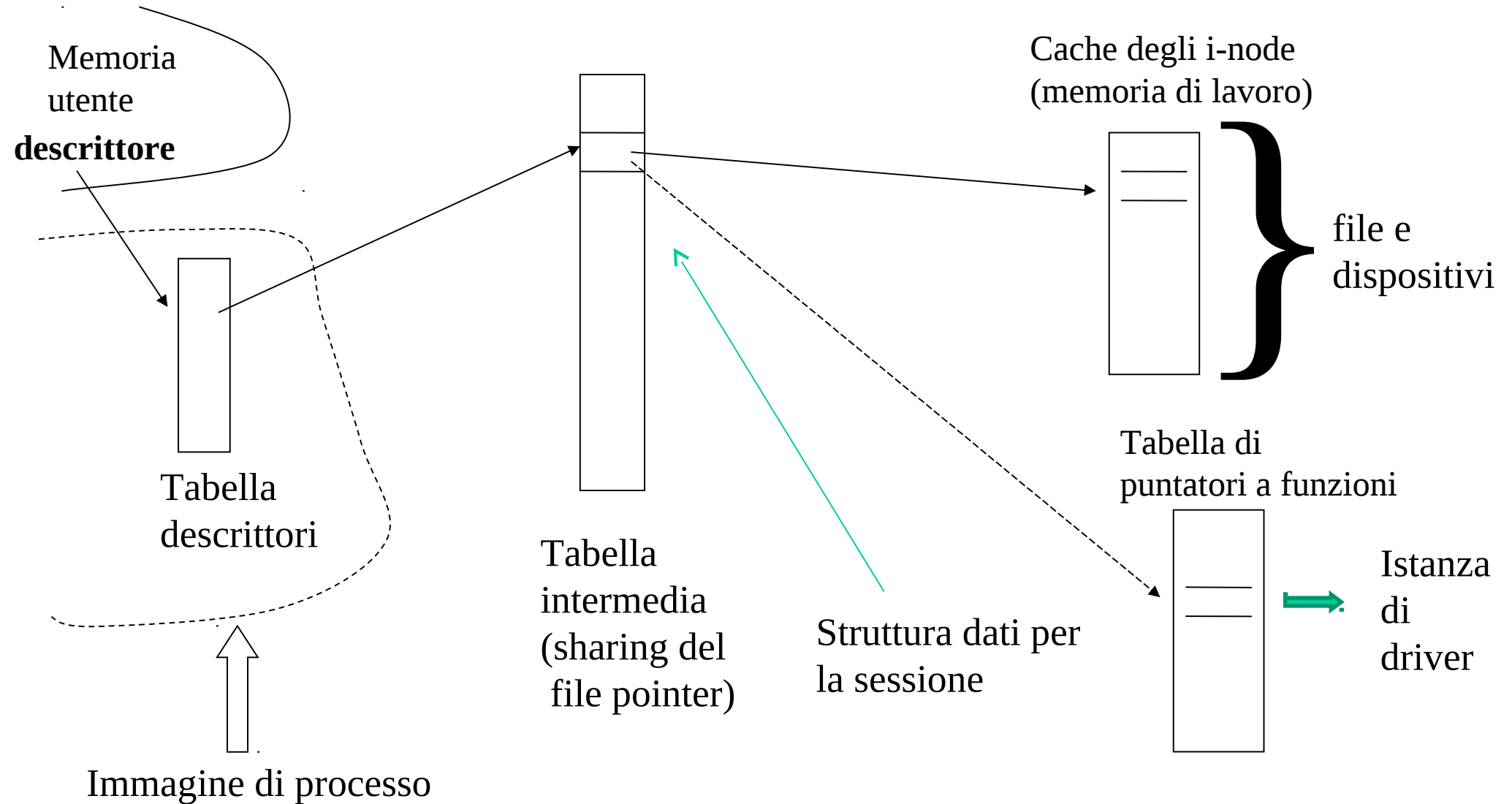
- Permette di avere i-nodes in cui specifichiamo a grana molto piú fine i permessi di accesso e gestione dei file
- Una ACL è realizzata tramite un **file shadow** (associato al file originale)
- Quindi anche tramite un **i-node shadow**
- L'i-node originale in tal caso ha un indice che identifica l'i-node shadow
- Con ACL si può per ogni file specificare i permessi di accesso per ogni singolo utente o gruppo del sistema
- I comandi basici di shell per gestire ACL sono **getfacl** e **setfacl**

# ACL example



Denominatore comune su  
named-users/groups

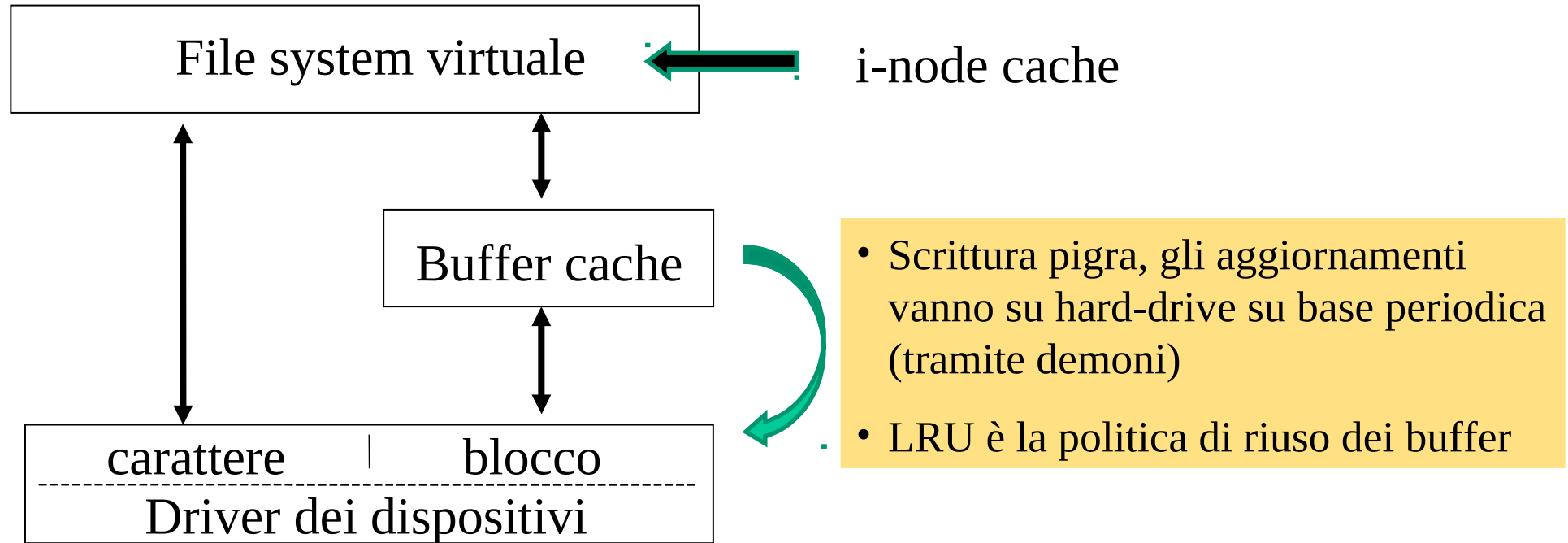
# UNIX virtual file system - architettura (I)



# UNIX virtual file system - architettura (II)

- i dispositivi vengono gestiti come file
- ogni i-node del file system virtuale viene associato o ad un file o ad un dispositivo
- le funzioni realmente eseguite su richiesta delle applicazioni dipendono dall'entità associata all'i-node relativo al descrittore per il quale viene invocata la funzione
- gli i-node associati ai file sono riportati su hard-drive allo spegnimento (shutdown) se non prima
- gli i-node associati ai dispositivi possono essere rimossi allo spegnimento (i-node dinamici)
- la cache degli i-node serve per gestire i-node (dinamici e non) e per rendere l'accesso ai file più efficiente
- esiste un **buffer-cache** per tenere temporaneamente i blocchi di dati relativi ai file in memoria di lavoro

# Architettura di I/O in sistemi UNIX





# Creazione di file

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

**Descrizione** invoca la creazione un file

**Argomenti**

- 1) \*file\_name: puntatore alla stringa di caratteri che definisce il nome del file da creare
- 2) mode: specifica i permessi di accesso al file da creare (codifica ottale)

**Restituzione** -1 in caso di fallimento; un descrittore di file altrimenti

## Esempio

```
void main() {  
    if(creat("pippo", 0666) == -1) {  
        printf("Errore nella chiamata creat\n");  
        exit(EXIT_FAILURE);  
    }  
}
```

# Apertura/chiusura di file

```
int open(char *file_name, int option_flags [, int mode])
```

**Descrizione** invoca la creazione un file

**Argomenti**

- 1) \*file\_name: puntatore alla stringa di caratteri che definisce il nome del file da aprire
- 2) option\_flags: specifica la modalità di apertura (read, write etc.)
- 3) mode: specifica i permessi di accesso al file in caso di creazione contestuale all'apertura

**Restituzione** -1 in caso di fallimento, altrimenti un descrittore per l'accesso al file

```
int close(int descriptor)
```

**Descrizione** invoca la chiusura di un file

**Argomenti** descriptor: descrittore del file da chiudere

**Restituzione** -1 in caso di fallimento

# Valori per “option\_flags”

O\_RDONLY: apertura del file in sola lettura

O\_WRONLY: apertura del file in sola scrittura

O\_RDWR: apertura in lettura e scrittura

O\_APPEND: apertura del file con puntatore alla fine del file; ogni scrittura sul file sarà effettuata a partire dalla fine del file

O\_CREAT : crea il file con modalità d'accesso specificate da *mode* solo se esso stesso non esiste

O\_TRUNC : elimina il contenuto del file se esso è già esistente

O\_EXCL : (exclusive) serve a garantire che il file sia stato effettivamente creato dalla chiamata corrente

- 
- definiti in **fcntl.h**
  - combinabili tramite l'operatore “|” (OR binario)

# Operazioni di lettura/scrittura

```
ssize_t read(int descriptor, char *buffer, size_t size)
```

**Descrizione** invoca la lettura da un file

**Argomenti**

- 1) descriptor: descrittore relativo al file da cui leggere
- 2) buffer: puntatore al buffer dove memorizzare i byte letti
- 3) size: quantita' di byte da leggere

**Restituzione** -1 in caso di fallimento, altrimenti il numero di byte realmente letti

```
ssize_t write(int descriptor, char *buffer, size_t size)
```

**Descrizione** invoca la scrittura su un file

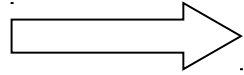
**Argomenti**

- 1) descriptor: descrittore relativo al file su cui scrivere
- 2) buffer: puntatore al buffer dove prendere i byte da scrivere
- 3) size: quantita' di byte da scrivere

**Restituzione** -1 in caso di fallimento, altrimenti il numero di byte realmente scritti

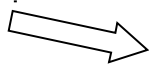
# Descrittori “speciali” ed eredità di descrittori

0 standard input  
1 standard output  
2 standard error



- associati a specifici oggetti di I/O ed utilizzati da molte funzioni di libreria standard (e.g. `scanf()`/`printf()`)
- i relativi stream possono essere chiusi

---

Tutti i descrittori vengono ereditati da un processo figlio  
generato da una `fork()`  **sharing del file pointer**

---

Tutti i descrittori (inclusi 0, 1 e 2) restano validi quando avviene una sostituzione di codice tramite una chiamata `execX()` eccetto che nel caso in cui si specifichi operatività `close-on-exec` (tramite la system-call `fcntl()` o il flag `O_CLOEXEC` in apertura del file)

# Riposizionamento del file pointer

```
off_t lseek(int descriptor, off_t offset, int option)
```

**Descrizione** invoca il riposizionamento del file pointer

**Argomenti**

- 1) descriptor: descrittore relativo al file su cui riposizionarsi
- 2) offset: quantità di caratteri di cui spostare il file pointer
- 3) option: opzione di spostamento (da inizio, da posizione corrente, da fine – valori relativi: 0, 1, 2)

**Restituzione** -1 in caso di fallimento, altrimenti il nuovo valore del file pointer

## Esempi

```
lseek(fd, 10, 0); /* Spostamento di 10 byte dall'inizio di fd */  
lseek(fd, 20, 1); /* Spostamento di 20 byte in avanti dalla posizione corrente */  
lseek(fd, -10, 1); /* Spostamento di 10 byte all'indietro dalla posizione corrente */  
lseek(fd, -10, 2); /* Spostamento di 10 byte all'indietro dalla fine del file */  
lseek(fd, -10, 0); /* Fallisce e il valore del file pointer resta uguale */
```

# Duplicazione di descrittori e redirezione

```
int dup(int descriptor)
```

**Descrizione** invoca la duplicazione di un descrittore

**Argomenti** descriptor: descrittore da duplicare

**Restituzione** -1 in caso di fallimento, altrimenti un nuovo descrittore

Il descrittore restituito è il primo libero nella tabella dei descrittori del processo

## Esempio

```
#include ...
#define FNAME "info.txt"
#define STDIN 0

int main() {
    int fd;
    fd = open(FNAME,O_RDONLY); /* Apro il file in lettura */
    close(STDIN);             /* Chiudo lo standard input */
    dup(fd);                  /* Duplico il descrittore di file*/
    execlp("more", "more", 0); /* Eseguo 'more' con input redirezionato */
}
```

# Più recenti standard

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

```
int dup(int oldfd)
```

```
int dup2(int oldfd, int newfd)
```



Specifica esplicita della posizione della  
“file-descriptor table” ove duplicare il canale originale



# Controllo di canale di I/O

## NAME [top](#)

`fcntl` - manipulate file descriptor

## SYNOPSIS [top](#)

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

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

## DESCRIPTION [top](#)

`fcntl()` performs one of the operations described below on the open file descriptor *fd*. The operation is determined by *cmd*.

`fcntl()` can take an optional third argument. Whether or not this argument is required is determined by *cmd*. The required argument type is indicated in parentheses after each *cmd* name (in most cases, the required type is *int*, and we identify the argument using the name *arg*), or *void* is specified if the argument is not required.

# Hard links e rimozione

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

ritorna -1 in caso di fallimento

---

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

ritorna -1 in caso di fallimento

Inseriscono/eliminano una directory-entry in un file speciale rappresentante una directory

# Symbolic links

- Sono file il cui contenuto identifica una stringa che definisce un pathname
- Possono esistere indipendentemente dal target

## SYNOPSIS

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

```
int symlink(const char *oldpath, const char *newpath)
```

# Gestione delle directory

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

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

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

---

Creazione/rimozione/apertura  
di directory

## Lettura di dir-entry

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

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

```
struct dirent {  
    ino_t      d_ino;    /* Inode number */  
    off_t      d_off;    /* Not an offset; see below */  
    unsigned short d_reclen; /* Length of this record */  
    unsigned char d_type; /* Type of file; not supported  
                           by all filesystem types */  
    char        d_name[256]; /* Null-terminated filename */  
};
```

# Gestione dei permessi d'accesso

## NAME

chmod, fchmod - change permissions of a file

## SYNOPSIS

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

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

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

```
int fchmod(int fildes, mode_t mode);
```

## DESCRIPTION

The mode of the file given by path or referenced by fields is changed.

---

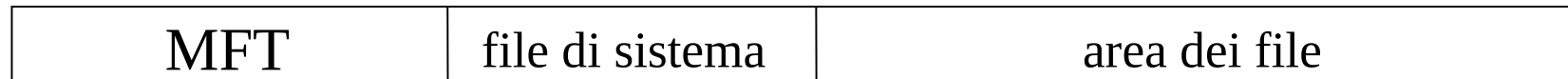
varianti **chown** gestiscono il cambio di proprietà

# Comandi shell basici per l'interazione con il File System

<b>link</b>	makes hard links
<b>ln</b>	makes symbolic links
<b>dumpe2fs</b>	dump file system info
<b>mkfs</b>	installs a file system on a device
<b>stat</b>	list lower level file system information

# Windows File System - NTFS

- Hard-drive divisi in volumi (partizioni), e organizzati in cluster fino a 64K byte
- per ogni volume si ha una **MFT (Master File Table)**
- ad ogni file corrisponde almeno un elemento nella MFT
- l'elemento contiene:
  - nome del file: fino a 255 char Unicode
  - informazioni sulla sicurezza
  - nome DOS del file: 8+3 caratteri
  - i dati del file, o puntatori per il loro accesso – organizzazione stile extent



Bitmap dei cluster + logfile (recupero in caso di crash)

# Alcuni attributi

Informazioni  
standard

---

**attributi di accesso, timestamp**

Lista di attributi

---

**dove localizzare nella MFT attributi  
che non entrano in un singolo record**

Descrittore di  
sicurezza

---

**permessi di accesso** per proprietario  
ed altri utenti

Nome

---

**identificazione nel sistema**

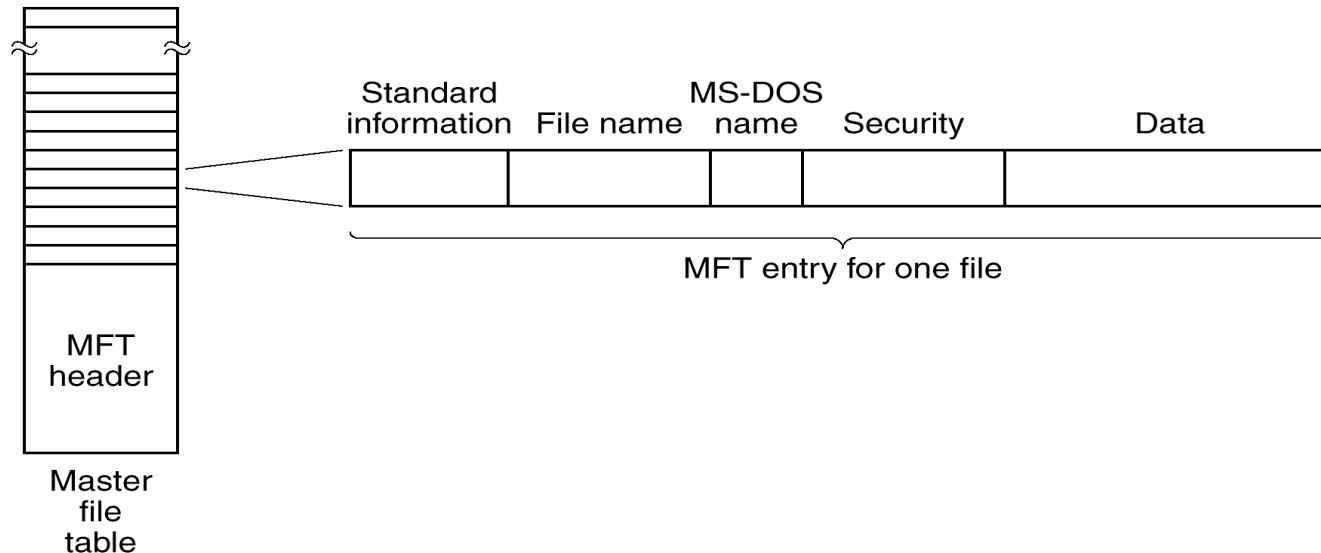
Dati

**visti tipicamente come attributo  
o come indici per l'accesso**



# MFT - Master File Table

- per piccoli file i dati sono direttamente nella parte dati nell'elemento della MFT (file immediati)
- per file grandi la parte dati dell'elemento della MFT contiene gli indirizzi di cluster o di gruppi di cluster
- se un elemento della MFT non basta si aggregano altri



# Architettura di I/O in sistemi Windows

## Gestore di I/O

Driver del  
file system

Gestore della  
cache

Driver di  
dispositivi

- scrittura pigra: gli aggiornamenti vanno su hard-drive su base periodica
- sostituzione cache LRU
- swapping tramite thread di sistema (su file)

# Creazione/apertura di file

```
HANDLE CreateFile(LPCTSTR lpFileName,  
                DWORD dwDesiredAccess,  
                DWORD dwShareMode,  
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
                DWORD dwCreationDisposition,  
                DWORD dwFlagsAndAttributes,  
                HANDLE hTemplateFile )
```

## Descrizione

- invoca la creazione di un file

## Restituzione

- un handle al nuovo file in caso di successo

# Parametri

- `lpFileName`: puntatore alla stringa di caratteri che definisce il nome del file da creare
- `dwDesiredAccess`: specifica la modalità di accesso al file da creare (`GENERIC_READ`, `GENERIC_WRITE`)
- `dwShareMode`: specifica se e quando il file può essere nuovamente aperto prima di essere stato chiuso (`FILE_SHARE_READ`, `FILE_SHARE_WRITE`)
- `lpSecurityAttributes`: specifica il descrittore della sicurezza del file
- `dwCreationDisposition`: specifica l'azione da fare se il file esiste e quella da fare se il file non esiste (`CREATE_NEW`, `CREATE_ALWAYS`, `OPEN_EXISTING`, `TRUNCATE_EXISTING`)
- `dwFlagsAndAttributes`: specifica varie caratteristiche del file (`FILE_ATTRIBUTES_NORMAL`, ... `HIDDEN`, ... `DELETE_ON_CLOSE`)
- `hTemplateFile`: specifica un handle ad un file di template

# Codifiche sui pathname

CreateFileA (...)

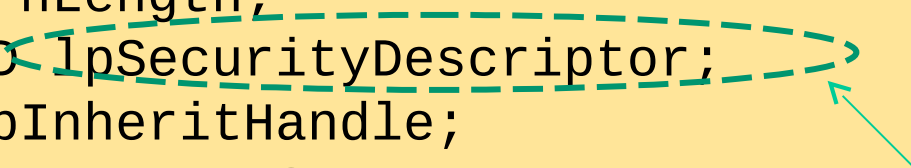
ASCII

CreateFileW(...)

UNICODE

# Richiami su SECURITY\_ATTRIBUTES

```
typedef struct _SECURITY_ATTRIBUTES {  
    DWORD nLength;  
    LPVOID lpSecurityDescriptor;  
    BOOL bInheritHandle;  
} SECURITY_ATTRIBUTES
```



Implementa la  
logica ACL

## Descrizione

- struttura dati che specifica permessi

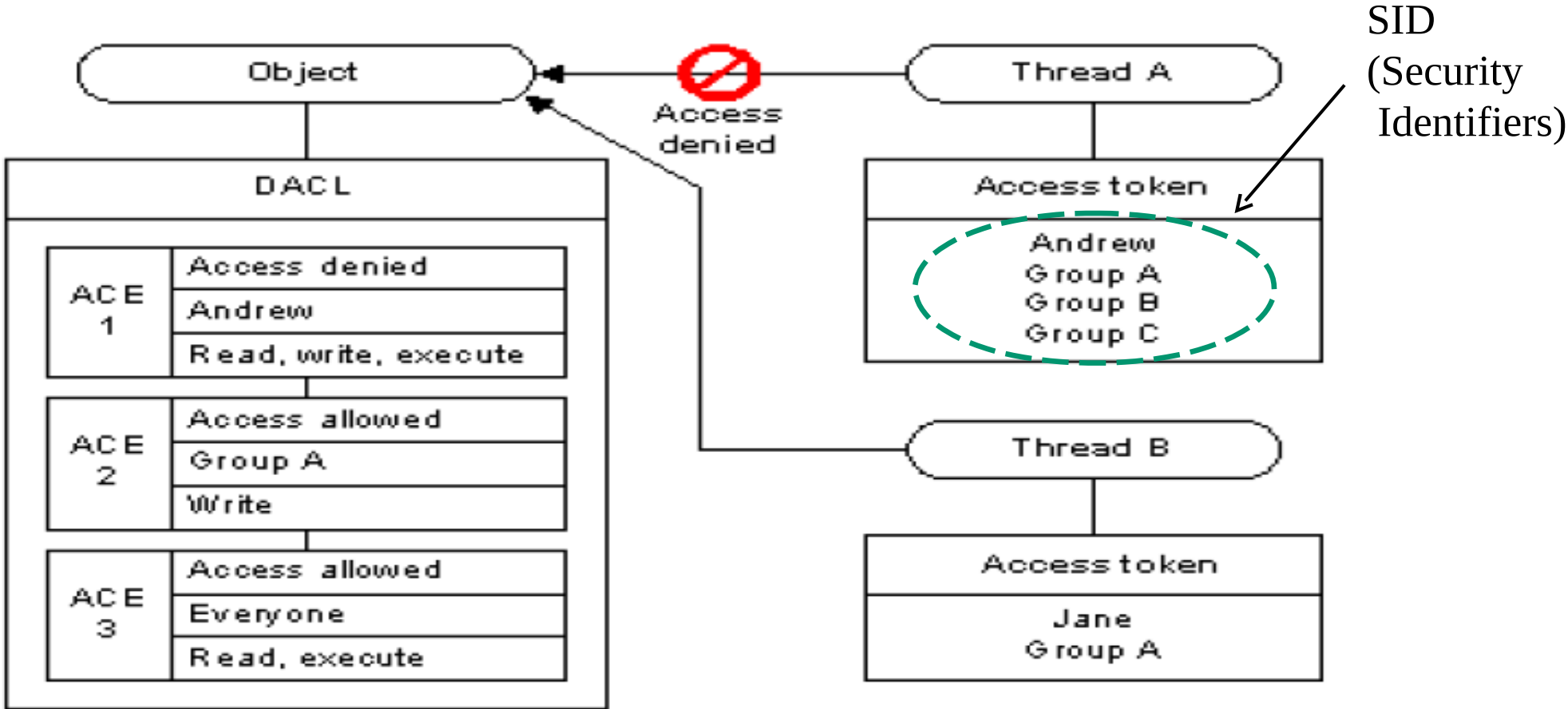
## Campi

- nLength: va settato SEMPRE alla dimensione della struttura
- lpSecurityDescriptor: puntatore a una struttura SECURITY\_DESCRIPTOR
- bInheritHandle: se uguale a TRUE un nuovo processo può ereditare l'handle a cui fa riferimento questa struttura

# ACL (Access Control List)

- specifica la descrizione della sicurezza di oggetti, quindi anche di file
- è formata da una lista di ACE (Access Control Entry)
- nel caso di generazione di oggetti in cui il descrittore di sicurezza non è specificato, ACL viene popolata a partire dall'*access token* del processo chiamante
- in tal caso si ha una ACL di default
- ACL è formata da DACL (Discretionary ACL) e SACL (System ACL)
- DACL specifica i permessi
- SACL specifica azioni di log da eseguire in base agli accessi
- sono entrambe parti di un security descriptor

# Un esempio





# Struttura di un Security Identifier (SID)

- Esso contiene una parte iniziale che indica il formato del SID
- Poi viene specificata la “authority entity per il rilascio del SID”
- Poi viene specificato il dominio di rilascio del SID
- Poi viene specificato il codice relativo l’entità (user/group) relativa al SID
- Un esempio (in rappresentazione stringa):

S-1-5-domain-500    SID dell’utente Administrator



SECURITY\_NT\_AUTHORITY

# Principali permessi gestibili da una ACE

- Modify
- Read & Execute
- Read
- Write



Tipici per file e cartelle

# Schema di utilizzo - riepilogo

- Ogni processo ha dei Security Identifier (come utente e come gruppo)
- I SID costituiscono l'access token che può direttamente garantire speciali privilegi (ad esempio l'accesso a qualsiasi file)
- Quando un processo cerca di accedere a un oggetto viene utilizzato l'access token per controllare se il processo ha diritti incondizionati sull'oggetto
- Altrimenti il kernel “scandisce” l'ACL controllando le singole ACE
- La prima ACE che specificamente garantisce o nega l'accesso richiesto è decisiva nell'esito dell'accesso

# Discovery dei SID

- Tramite registry attraverso il file

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\  
CurrentVersion\ProfileList
```

- Tramite (power)shell command

```
wmic useraccount get name,sid
```

```
whoami /user
```

# Manipolazione di ACL

GetNamedSecurityInfo

GetSecurityInfo

SetNamedSecurityInfo

SetSecurityInfo



WinAPI basiche

A queste Win-API vengono associate una serie di ulteriori funzioni di libreria dello standard WinAPI per la manipolazione di security descriptors in user space

# Un esempio

The **GetSecurityInfo** function retrieves a copy of the *security descriptor* for an object specified by a handle.

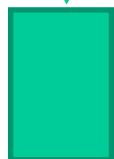
## Syntax

C++

```
DWORD WINAPI GetSecurityInfo(  
    _In_      HANDLE          handle,  
    _In_      SE_OBJECT_TYPE  ObjectType,  
    _In_      SECURITY_INFORMATION SecurityInfo,  
    _Out_opt_ PSID            *ppsidOwner,  
    _Out_opt_ PSID            *ppsidGroup,  
    _Out_opt_ PACL            *ppDacl,  
    _Out_opt_ PACL            *ppSacl,  
    _Out_opt_ PSECURITY_DESCRIPTOR *ppSecurityDescriptor  
);
```

field pointers  
into the descriptor

need free  
after usage



# Tipi di oggetto

```
typedef enum _SE_OBJECT_TYPE {
    SE_UNKNOWN_OBJECT_TYPE      = 0,
    SE_FILE_OBJECT,
    SE_SERVICE,
    SE_PRINTER,
    SE_REGISTRY_KEY,
    SE_LMSHARE,
    SE_KERNEL_OBJECT,
    SE_WINDOW_OBJECT,
    SE_DS_OBJECT,
    SE_DS_OBJECT_ALL,
    SE_PROVIDER_DEFINED_OBJECT,
    SE_WMIGUID_OBJECT,
    SE_REGISTRY_WOW64_32KEY
} SE_OBJECT_TYPE;
```

# Associazione SID/users/groups

## LookupAccountSid function

The **LookupAccountSid** function accepts a *security identifier* (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found.

### Syntax

C++

```
BOOL WINAPI LookupAccountSid(  
    _In_opt_ LPCTSTR lpSystemName,  
    _In_ PSID lpSid,  
    _Out_opt_ LPTSTR lpName,  
    _Inout_ LPDWORD cchName,  
    _Out_opt_ LPTSTR lpReferencedDomainName,  
    _Inout_ LPDWORD cchReferencedDomainName,  
    _Out_ PSID_NAME_USE peUse  
);
```

actual buffers

account type

buffer sizes



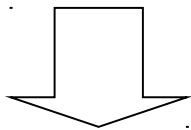
# Costruzione di un security descriptor - schema

Servizi WinAPI da usare e sequenza di uso

1. InitializeSecurityDescriptor ← Reset della struttura dati
2. SetSecurityDescriptorOwner
3. SetSecurityDescriptorGroup } Scrittura dei SID nella struttura dati
4. InitializeAcl ← Inizializzazione di un buffer ACL
5. AddAccessDeniedAce.... } Inserimento di ACE in ACL
6. AddAccessAllowedAce...
7. SetSecurityDescriptorDacl ← Collegamento di ACL al security descriptor

# Chiusura di file

BOOL CloseHandle(HANDLE hObject)



chiude un oggetto

## **Descrizione**

- invoca la chiusura di un oggetto (ad esempio un file)

## **Restituzione**

- 0 in caso di fallimento

# Letture da file

```
BOOL ReadFile(HANDLE hFile,  
              LPVOID lpBuffer,  
              DWORD nNumberOfBytesToRead,  
              LPDWORD lpNumberOfBytesRead,  
              LPOVERLAPPED lpOverlapped)
```

## Descrizione

- invoca la lettura di una certa quantità di byte da un file

## Restituzione

- 0 in caso di fallimento

# Parametri

- `hFile`: handle valido al file da cui si vuole leggere
- `lpBuffer`: puntatore all'area di memoria nella quale i caratteri letti devono essere bufferizzati
- `tnNumberOfBytesToRead`: definisce il numero di caratteri (byte) che si vogliono leggere
- `lpNumberOfBytesRead`: puntatore a un intero positivo corrispondente al numero di caratteri effettivamente letti in caso di successo
- `lpOverlapped`: puntatore a una struttura `OVERLAPPED` da usarsi per I/O asincrono

# Scrittura su file

```
BOOL WriteFile(HANDLE hFile,  
              LPCVOID lpBuffer,  
              DWORD nNumberOfBytesToWrite,  
              LPDWORD lpNumberOfBytesWritten,  
              LPOVERLAPPED lpOverlapped)
```

## Descrizione

- invoca la scrittura di una certa quantità di byte su un file

## Restituzione

- 0 in caso di fallimento

# Parametri

- `hFile`: handle valido al file su cui si vuole scrivere
- `lpBuffer`: puntatore all'area di memoria che contiene i caratteri da scrivere
- `tnNumberOfBytesToWrite`: definisce il numero di caratteri (byte) che si vogliono scrivere
- `lpNumberOfBytesWritten`: puntatore a un intero positivo corrispondente al numero di caratteri effettivamente scritti in caso di successo
- `lpOverlapped`: puntatore a una struttura `OVERLAPPED` da usarsi per I/O asincrono

# Cancellazione di file

BOOL DeleteFile(LPCTSTR lpFileName)

## Descrizione

- invoca la cancellazione di un file

## Parametri

- lpFileName: puntatore alla stringa di caratteri che definisce il nome del file che si vuole rimuovere

## Restituzione

- un valore diverso da zero in caso di successo, 0 in caso di fallimento

# Riposizionamento del file pointer

```
DWORD SetFilePointer(HANDLE hFile,  
                    LONG lDistanceToMove,  
                    PLONG lpDistanceToMoveHigh,  
                    DWORD dwMoveMethod)
```

## Descrizione

- invoca il riposizionamento del file pointer

## Restituzione

- `INVALID_SET_FILE_POINTER` in caso di fallimento, i 32 bit meno significativi del nuovo valore del file pointer (valutato in caratteri dall'inizio del file) in caso di successo



# Parametri

- `hFile`: handle di file che identifica il canale di input/output associato al file per il quale si vuole modificare il file pointer
- `lDistanceToMove`: i 32 bit meno significativi di un valore intero con segno indicante il numero di caratteri di cui viene spostato il file pointer
- `lpDistanceToMoveHigh`: (opzionale) puntatore a un long contenente i 32 bit più significativi del valore in `lDistanceToMove`
- `dwMoveMethod`: tipo di spostamento da effettuare (`FILE_BEGIN`, `FILE_CURRENT`, `FILE_END`)

# Gestione degli standard handle

```
HANDLE WINAPI GetStdHandle( _In_ DWORD nStdHandle );
```

```
BOOL WINAPI SetStdHandle( _In_ DWORD nStdHandle, _In_ HANDLE hHandle );
```

Value	Meaning
<b>STD_INPUT_HANDLE</b> (DWORD)-10	The standard input device.
<b>STD_OUTPUT_HANDLE</b> (DWORD)-11	The standard output device.
<b>STD_ERROR_HANDLE</b> (DWORD)-12	The standard error device.

# Gestione delle directory

```
BOOL WINAPI CreateDirectory(  
    __in LPCTSTR lpPathName,  
    __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes );
```

```
BOOL WINAPI RemoveDirectory(  
    __in LPCTSTR lpPathName );
```

```
HANDLE FindFirstFile(  
    [in] LPCSTR lpFileName,  
    [out] LPWIN32_FIND_DATA lpFindFileData  
);
```

```
BOOL FindNextFile(  
    [in] HANDLE hFindFile,  
    [out] LPWIN32_FIND_DATA lpFindFileData  
);
```



ASCII vs UNICODE

# NTFS hard links

```
BOOL WINAPI CreateHardLink( LPCTSTR lpFileName,  
                            LPCTSTR lpExistingFileName,  
                            LPSECURITY_ATTRIBUTES lpSecurityAttributes );
```

*lpFileName* [in]

- The name of the new file.
- This parameter cannot specify the name of a directory.

*lpExistingFileName* [in]

- The name of the existing file.
- This parameter cannot specify the name of a directory.

*lpSecurityAttributes*

- Reserved; must be NULL.

**NOTE (taken from MSDN online manual):**

Because hard links are only directory entries for a file, many changes to that file are instantly visible to applications that access it through the hard links that reference it. However, the directory entry size and attribute information is updated only for the link through which the change was made.

The security descriptor belongs to the file to which a hard link points. The link itself is only a directory entry, and does not have a security descriptor. Therefore, when you change the security descriptor of a hard link, you change the security descriptor of the underlying file, and all hard links that point to the file allow the newly specified access. You cannot give a file different security descriptors on a per-hard-link basis.

# NTFS symbolic links (.lnk shortcuts)

BOOLEAN WINAPI CreateSymbolicLink(

\_\_in LPTSTR lpSymlinkFileName,

\_\_in LPTSTR lpTargetFileName,

\_\_in DWORD dwFlags )

*lpSymlinkFileName* [in]

- The symbolic link to be created.

*lpTargetFileName* [in]

- The name of the target for the symbolic link to be created.
- If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link

*dwFlags* [in]

- Indicates whether the link target, *lpTargetFileName*, is a directory.

See also the following  
prompt-command:

**mklink**

# I/O scheduling

- Definisce la pianificazione tramite la quale dispositivi di I/O vengono realmente attivati per le loro operazioni
- In taluni casi I/O scheduling è semplicemente attuato secondo una politica FCFS
- Questo è **tipico dei terminali**, poichè utenti interattivi non sarebbero soddisfatti di osservare in output (o di fornire in input) dati fuori ordine (a causa di un riordino delle reali operazioni sul dispositivo)
- Ci sono però contesti in cui politiche più articolate di FCFS diventano fondamentali per l'efficienza dell'intera architettura di I/O, e quindi del virtual file system
- Questo è il caso degli hard-disk a rotazione, tutt'ora componenti centrali in architetture moderne



# Dispositivi di memoria di massa

## Dischi magnetici a rotazioni (Hard Disks)

- sono dispositivi elettro-meccanici
- sono tutt'ora i dispositivi con i migliori vantaggi in termini di capacità vs costo
- sono basati su tracce (che ospitano blocchi) e una testina che attraversa tracce per riposizionarsi
- ogni blocco corrisponde ad un “blocco logico” destinato per uno specifico file

## Solid State Drives (SSD)

- sono dispositivi elettronici della classe Non Volatile Memory (NVM)
- sono la tecnologia emergente, grazie alle più elevate velocità rispetto agli Hard Disks
- ogni blocco contiene un insieme di “blocchi logici” destinati ad ospitare file

# Dischi magnetici a rotazione

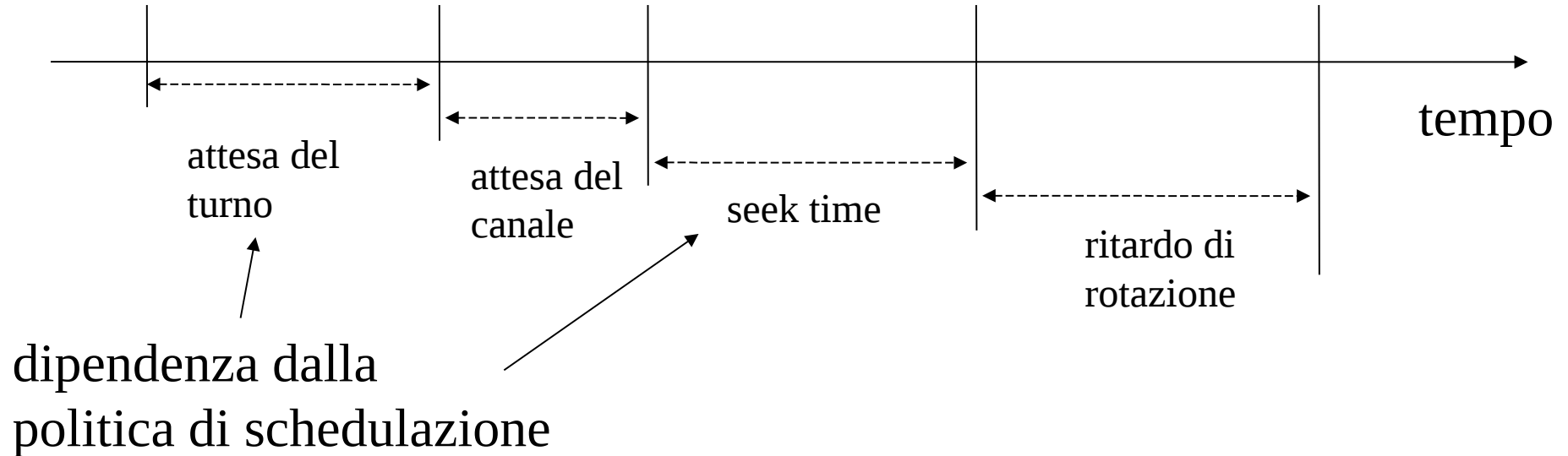
- Ogni blocco è leggibile e scrivibile ad ogni istante di tempo
- Non ci sono fasi di vita di un blocco ove date operazioni (ad esempio una scrittura) non possono essere effettuate
- L'usura del dispositivo è essenzialmente legata all'usura delle parti meccaniche
- Non ci sono relazioni dirette tra le operazioni di lettura scrittura dei blocchi e l'usura (ovvero l'usura è primariamente legata al tempo di vita del dispositivo meccanico)
- Le relazioni sono indirette, dovute per esempio al fatto che per posizionare la testina su un blocco questa dovrà essere soggetta ad uno spostamento

# Schedulazione dei dischi magnetici a rotazione

## Parametri

- tempo di ricerca della traccia (seek time)
- ritardo di rotazione per l'accesso al settore sulla traccia

## Punto di vista del processo



# Valutazione dei parametri

## Seek time

- $n$  = tracce da attraversare
- $m$  = costante dipendente dal dispositivo
- $s$  = tempo di avvio

$$\Rightarrow T_{seek} = m \times n + s$$

## Ritardo di rotazione

- $b$  = bytes da trasferire
- $N$  = numero di bytes per traccia
- $r$  = velocità di rotazione (rev. per min.)

$$\Rightarrow T_{rotazione} = \frac{b}{r \times N}$$

---

## Valori classici

- $s = 20/3/..$  msec.
- $m = 0.3/0.1/..$  msec.
- $r = 300/3600/.../7200$  rpm (floppy vs hard disk)

**fattore critico**

# Scheduling FCFS (First Come First Served)

- le richieste di I/O vengono servite nell'ordine di arrivo
- non produce starvation
- non minimizza il seek time

## Un esempio

Traccia iniziale = 100

Sequenza delle tracce accedute = 55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 184

**distanze** 45 - 3 - 19 - 21 - 72 - 70 - 10 - 112 - 146

lunghezza media di ricerca

$$\frac{\sum_{i=1}^{|insiemedist|} dist_i}{|insiemedist|} = 55.3$$

# Scheduling SSTF (Shortest Service Time First)

- si dà priorità alla richiesta di I/O che produce il minor movimento della testina
- non minimizza il tempo di attesa medio
- può provocare starvation

## Un esempio

Traccia iniziale = 100

Insieme delle tracce coinvolte = 55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 184

Riordino in base al seek time = 90 - 58 - 55 - 39 - 38 - 18 - 150 - 160 - 184

**distanze** 10 - 32 - 3 - 16 - 1 - 20 - 132 - 10 - 24

lunghezza media di ricerca

$$\frac{\sum_{i=1}^{|insemedist|} dist_i}{|insemedist|} = 27.5$$

# Scheduling SCAN (elevator algorithm)

- il seek avviene in una data direzione fino al termine delle tracce o fino a che non ci sono più richieste in quella direzione
- sfavorevole all'area attraversata più di recente (ridotto sfruttamento della località)
- la versione C-SCAN utilizza scansione in una sola direzione (fairness nel servizio delle richieste)

## Un esempio

Traccia iniziale = 100

Direzione iniziale = crescente

Insieme delle tracce coinvolte = 55 – 58 – 39 – 18 – 90 – 160 – 150 – 38 – 184

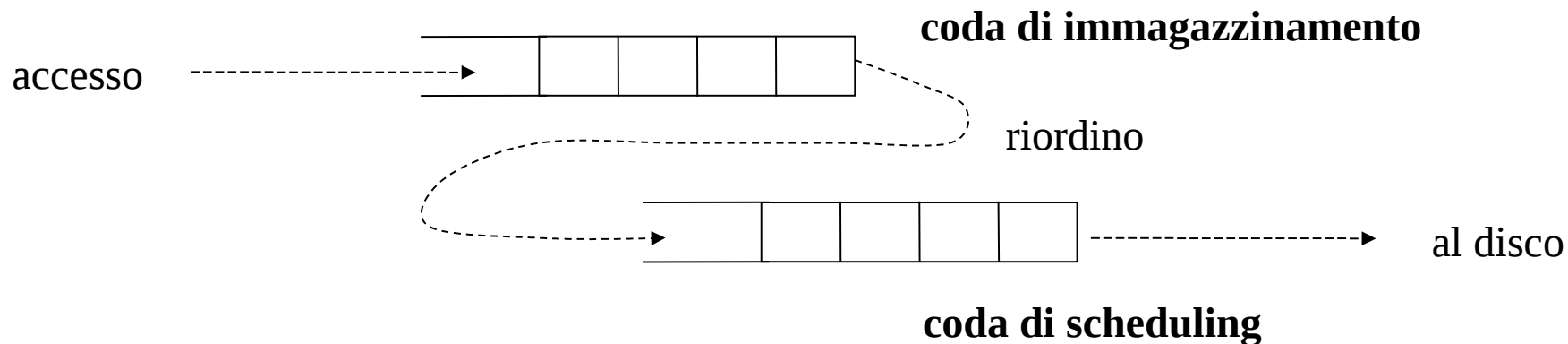
Riordino in base alla direzione di seek = 150 – 160 – 184 – 90 – 58 – 55 – 39 – 38 – 18

**distanze**     50 – 10 – 24 – 94 – 32 – 3 – 16 – 1 – 20

$$\text{lunghezza media di ricerca} = \frac{\sum_{i=1}^{|\text{insiemedist}|} \text{dist}_i}{|\text{insiemedist}|} = 27.8$$

# Scheduling FSCAN (ad esempio LINUX)

- SSTF, SCAN e C-SCAN possono mantenere la testina bloccata in situazioni patologiche di accesso ad una data traccia
- FSCAN usa due code distinte per la gestione delle richieste
- la schedulazione avviene su una coda
- l'immagazzinamento delle richieste di I/O per la prossima schedulazione avviene sull'altra coda
- nuove richieste non vengono considerate nella sequenza di scheduling già determinata





# Dispositivi SSD

- Ogni blocco è leggibile e scrivibile
- Il blocco può essere scritto solo dopo esser stato “cancellato”
- Ad ogni cancellazione il blocco si “deteriora”
- Il numero massimo di cancellazioni prima che il blocco non sia più utilizzabile è dell'ordine di 100000 (detti anche cicli di cancellazione)
- C'è uno sbilanciamento nella velocità delle letture e delle scritture (dovuto alla cancellazione)
- Il controller del dispositivo SSD tipicamente implementa algoritmi di gestione trasparente del “derouting” delle scritture per ammortizzare costi ed effetti negativi della cancellazione

# Dettagli

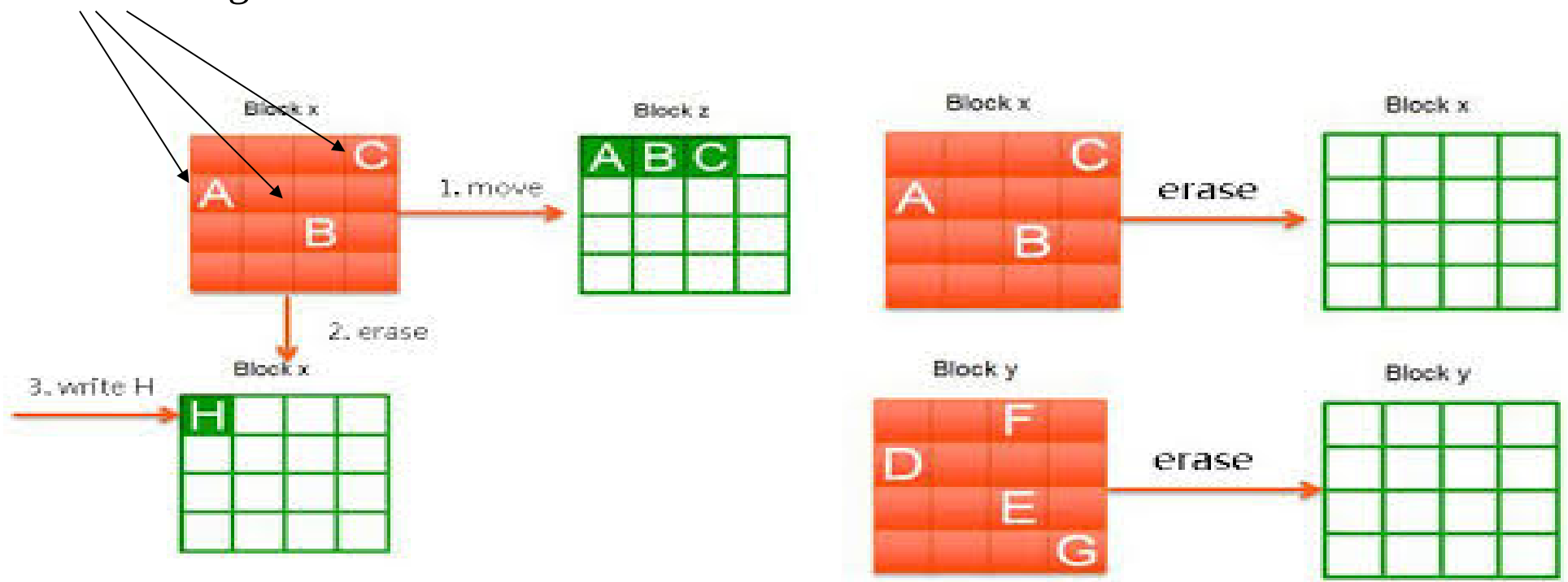
- I blocchi fisici possono contenere più di un blocco logico (detto anche pagina)
- Le scritture portano i blocchi logici ad essere scritti su blocchi fisici cancellati in precedenza (quindi in posizioni differenti del dispositivo)
- La cancellazione viene effettuata tramite algoritmi implementati all'interno del dispositivo (non via software), chiamati garbage collection
- Quando un blocco viene “garbage-collected” i suoi blocchi logici ancora validi vengono riscritti su un nuovo blocco fisico
- Si usa una tecnica detta “over-provisioning” per mantenere almeno una percentuale (ad esempio il 20%) di blocchi fisici liberi per le nuove scritture e per le riscritture di blocchi logici validi

# Scheduling SSD e identificazione dei blocchi

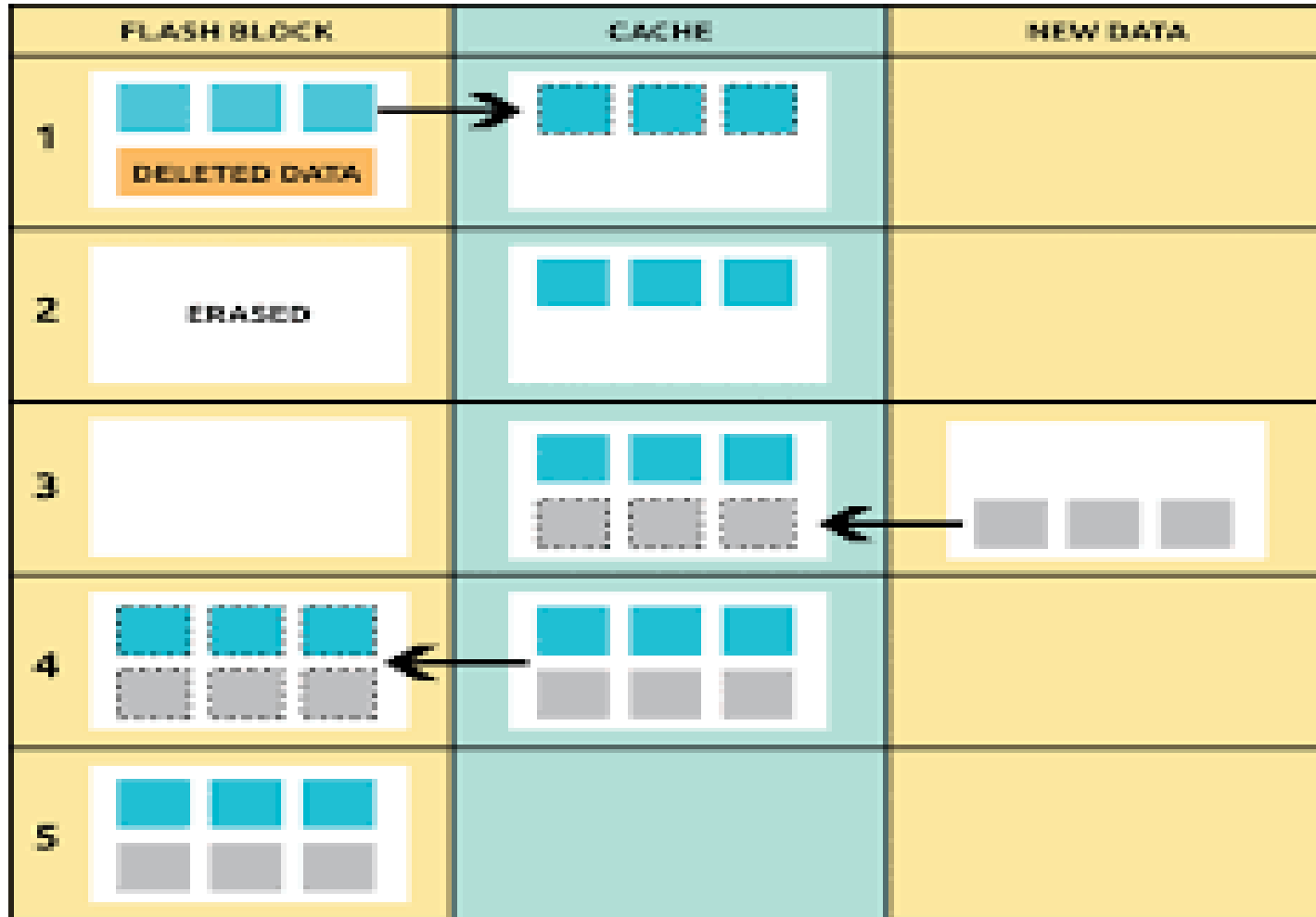
- Il sistema operativo tipicamente interagisce con l'SSD secondo regola FIFO
- Viene comunque adottata una sorta di “fusion” delle richieste di scrittura proprio per ottimizzare la durata del dispositivo
- Come detto, su un SSD abbiamo blocchi fisici che contengono più blocchi logici
- Il file system usa i blocchi logici, ed i loro identificatori
- Il mapping di dove si trovi realmente un blocco logico, rispetto al blocco fisico ospitante, è mantenuto da una tabella di lookup interna al dispositivo SSD
- Ogni elemento della tabella è una tupla <blocco-logico, blocco-fisico, pagina> dove il blocco-logico è l'identificatore di blocco come usato a livello del file system del sistema operativo

# Uno schema basico

Blocchi logici



# Garbage collection e ricompattazione



# Semantica della consistenza sul file system

## Semantica UNIX

- ogni scrittura di dati sul file system è direttamente visibile ad ogni lettura che sia eseguita successivamente (riletture da buffer cache)
- supportata anche da Windows NT/2000/....
- solo approssimata da NFS (Network File System)

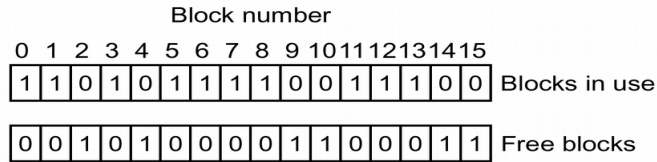
## Semantica della sessione

- ogni scrittura di dati sul file system è visibile solo dopo che il file su cui si è scritto è stato chiuso (buffer cache di processo)
- supportata dal sistema operativo AIX (file system ANDREW)

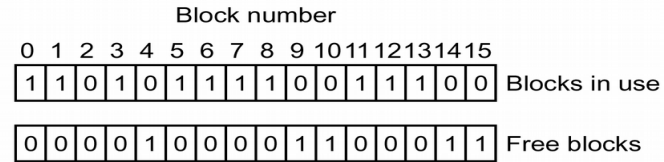
# Problematiche di consistenza nei file system

- crash di sistema possono portare il file system in uno stato inconsistente a causa di
  1. operazioni di I/O ancora pendenti nel buffer cache
  2. aggiornamenti della struttura del file system ancora non permanenti
- possibilità di ricostruire il file system con apposite utility
  - *fsck* in UNIX
  - *scandisk* in Windows
- vengono analizzate le strutture del file system (MFT, I-nodes) e si ricava per ogni file system:
  - Blocchi in uso: a quanti e quali file appartengono (si spera uno eccetto che in scenari di deduplicazione)
  - Blocchi liberi: quante volte compaiono nella lista libera (si spera una o nessuna)

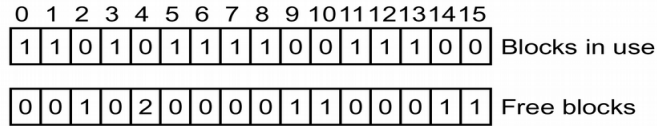
# Ricostruzione dei file system



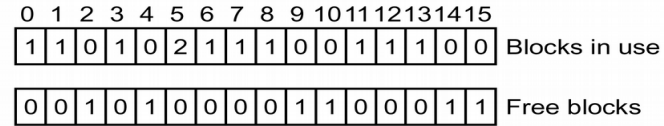
(a)



(b)



(c)



(d)

**a)** Situazione consistente

**b)** *Il blocco 2 non è in nessun file né nella lista libera:*  
aggiungilo alla lista libera

**c)** *Il blocco 4 compare due volte nella lista libera:* togli  
un'occorrenza

**d)** *Il blocco 5 compare in due file:* duplica il blocco e  
sostituiscilo in uno dei file (rimedio parziale!!!)



# Sincronizzazione del file system UNIX

## NAME

`fsync`, `fdatasync` - synchronize a file's complete in-core state with that on disk

## SYNOPSIS

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);
```

## DESCRIPTION

`fsync` copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage. It also updates metadata stat information. It does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit `fsync` on the file descriptor of the directory is also needed.

`fdatasync` does the same as `fsync` but only flushes user data, not the meta data like the `mtime` or `atime`.

# Sincronizzazione del file system Windows

The **FlushFileBuffers** function flushes the buffers of a specified file and causes all buffered data to be written to a file.

```
BOOL FlushFileBuffers(  
    HANDLE hFile  
);
```

## Parameters

*hFile*

[in] A handle to an open file.

The file handle must have the GENERIC\_WRITE access right. For more information, see [File Security and Access Rights](#).

If *hFile* is a handle to a communications device, the function only flushes the transmit buffer.

If *hFile* is a handle to the server end of a named pipe, the function does not return until the client has read all buffered data from the pipe.

# stdio.h vs file system API

**FILE \*fopen(const char \*path, const char \*mode)**

open(..)  
CreateFile(..)

**int fscanf(FILE \*stream, const char \*format, ...)**

buffering +  
read(..)  
ReadFile(..)

**int fprintf(FILE \*stream, const char \*format, ...)**

buffering +  
write(..)  
WriteFile(..)

**EOF**

errno  
GetLastError()