

Sistemi Operativi

Laurea in Ingegneria Informatica

Universita' di Roma Tor Vergata

Docente: Francesco Quaglia



Virtual file system

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

Virtual file system

- E' 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 stessa
- Si basano quindi 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

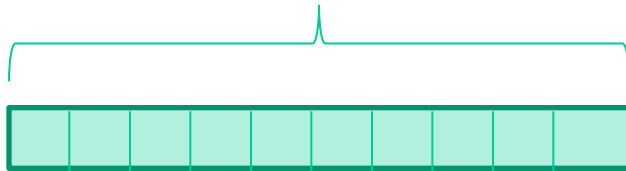
Reali operazioni di I/O
(**interfaccia comune**)

Lecture
Scritture
Ripozionamenti
Eliminazione di contenuti (se
non volatili)

Stream/block I/O model

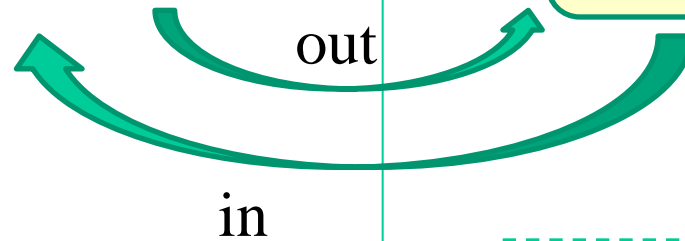
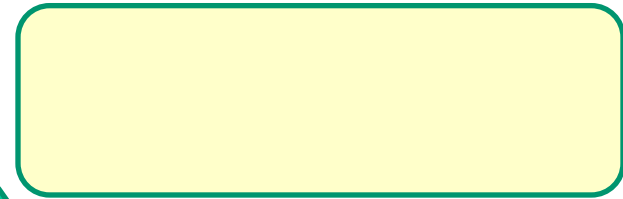
User space memory

char v[size]



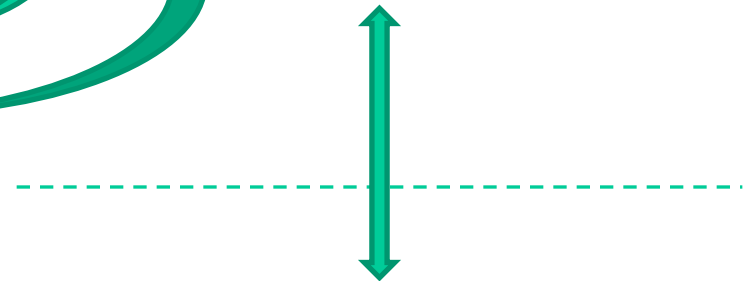
Kernel space memory (RAM storage)

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



in

out



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

- ✓ hard disks
- ✓ network interfaces

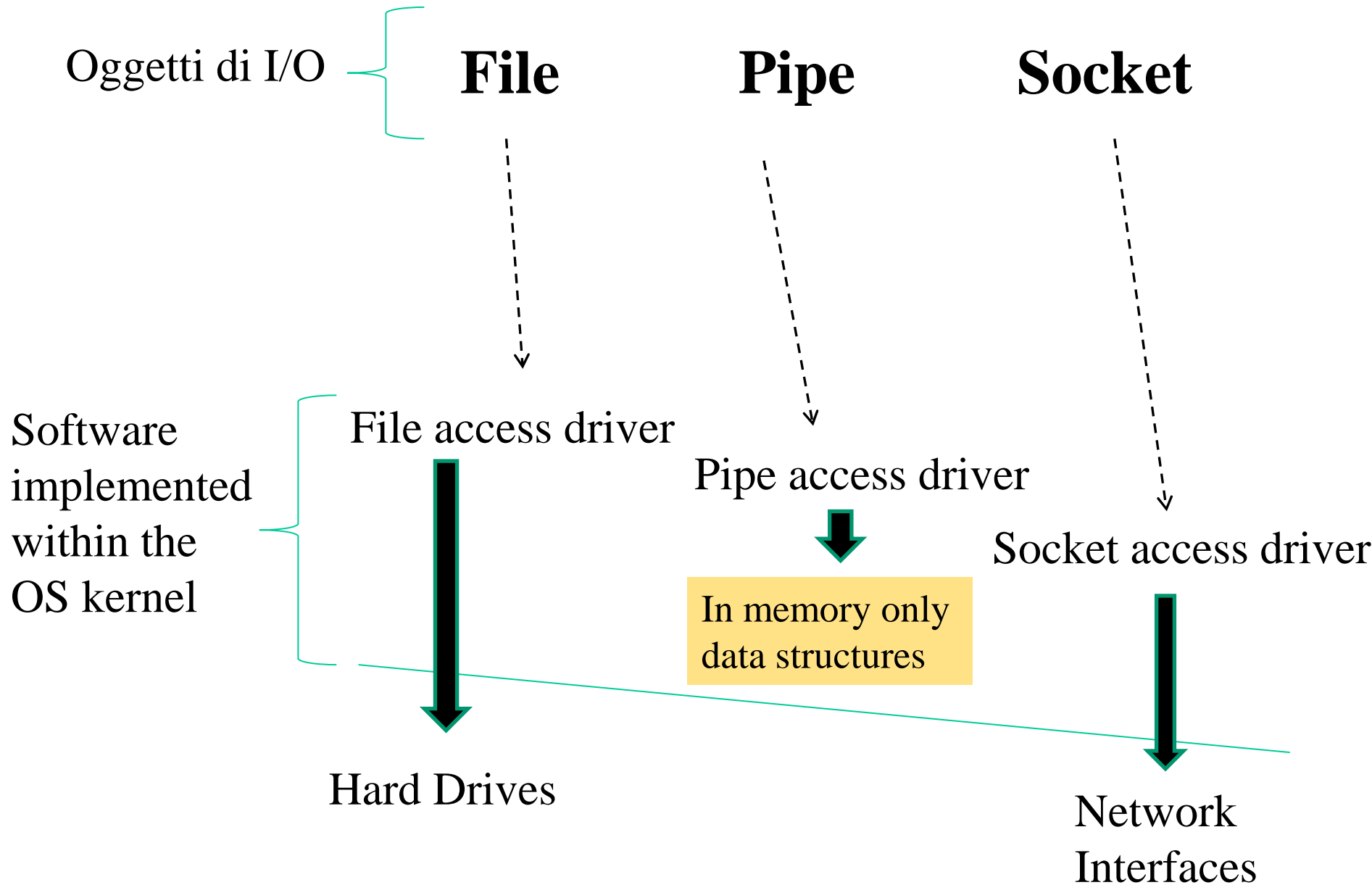
Stream I/O puo' portare a letture di **frazioni arbitrarie di dati** scritti in precedenza

Block I/O porta a letture di **unita' 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 eterogeneita' e' risolta in modo del tutto trasparente al codice applicativo tramite il concetto di driver
- Il driver e' l'insieme di moduli software di livello kernel per eseguire le operazioni afferenti ad un qualsiasi oggetto di I/O
- Ogni tipologia di oggetti ha la sua tipologia di driver

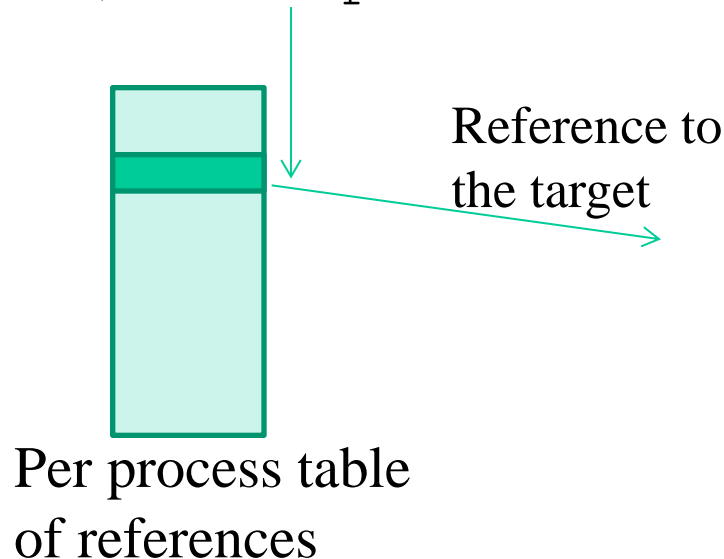
Uno schema



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 istanziiazione 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

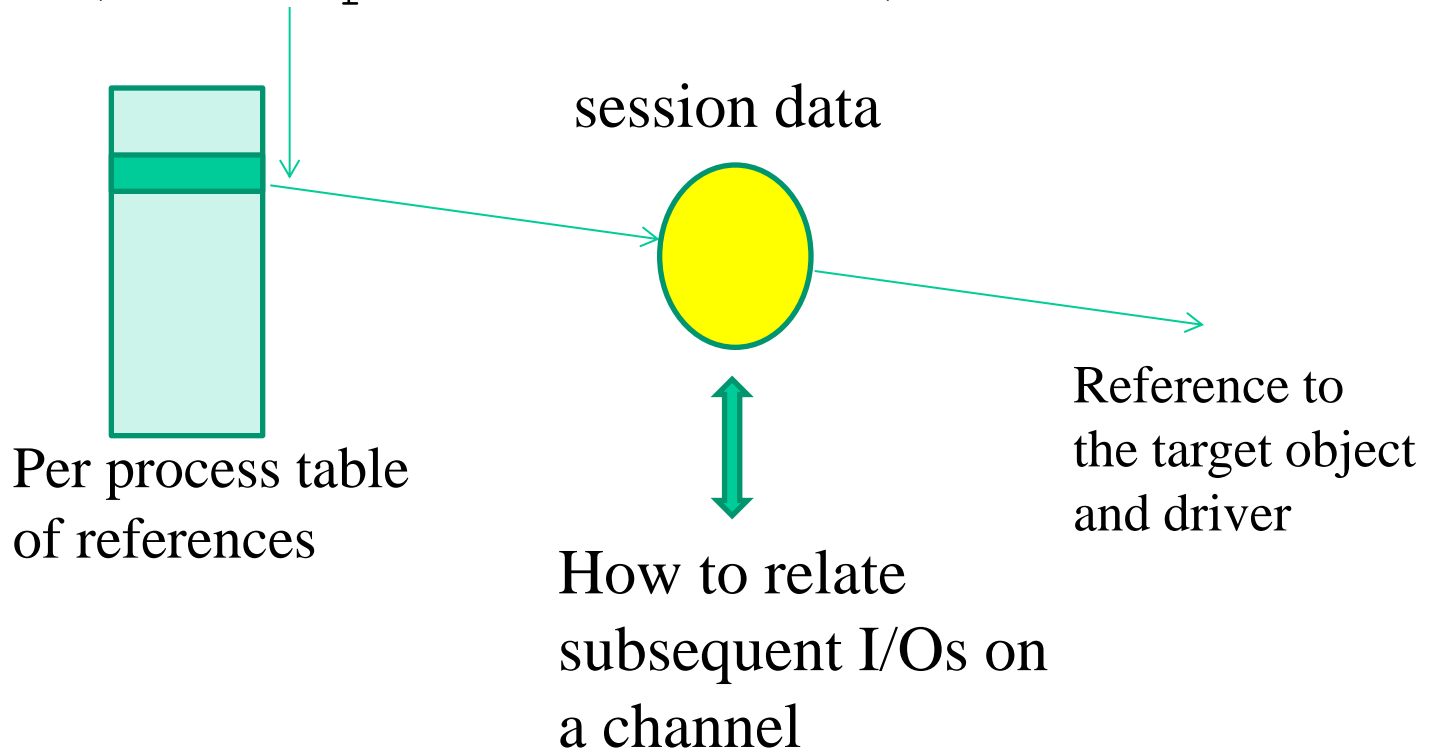
`I/O_Syscall(descriptor/handle`



Sessioni di I/O

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

`I/O_Syscall(descriptor/handle`



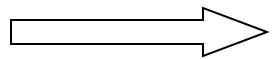
Il file system

Il punto di vista del sistema operativo

- minima unita' 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 unita' informativa accessibile (manipolabile): **il record**

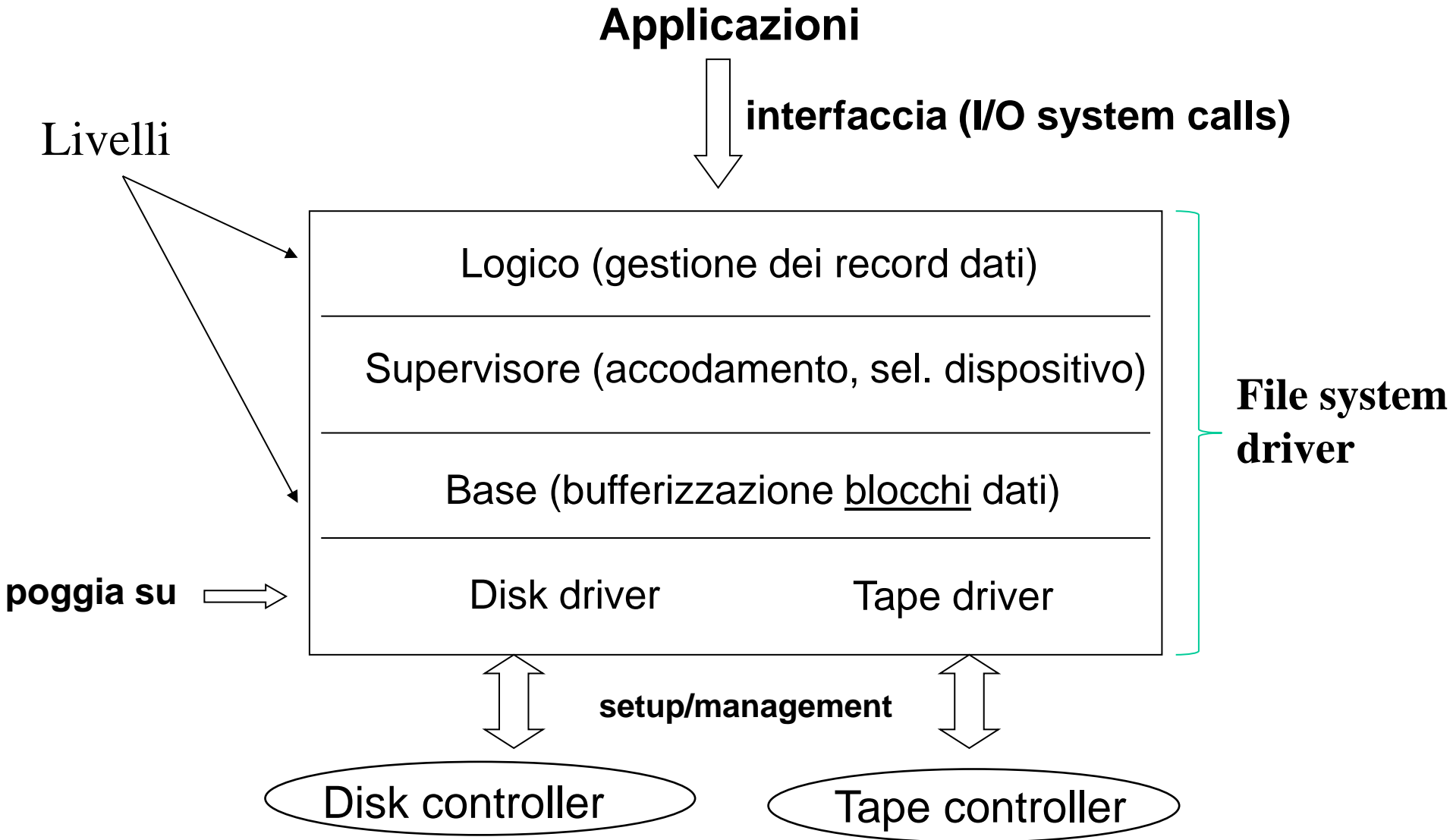


Il file system mostra alle applicazioni ogni singolo file come una semplice sequenza di records

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 sistema)

Architettura di base di un file system



Operazioni base sui file – I/O interface

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

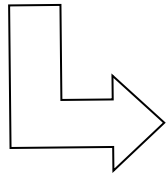
Troncamento

- 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 **PUO'** essere condiviso da piu' processi

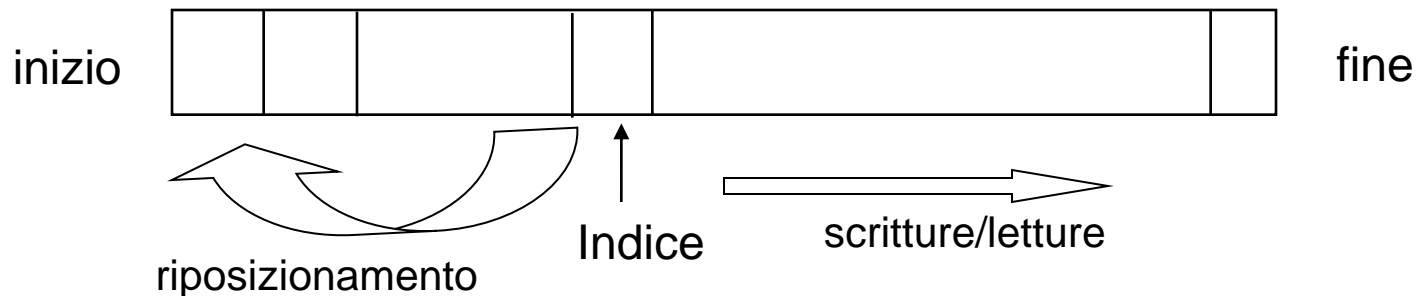


- ✓ quindi **NON** fa parte della singola immagine di processo mantenuta dal sistema operativo
- ✓ fa tipicamente parte dell'immagine di sessione

- Le modalita' 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 e' incrementato di una unita' per ogni record acceduto
- il riposizionamento dell'indice puo' avvenire solo all'inizio del file



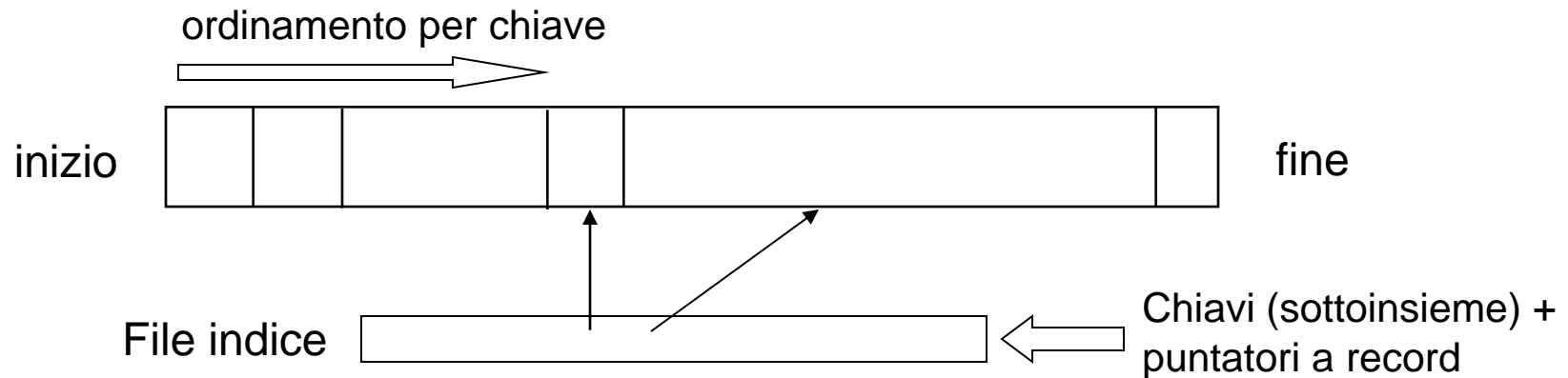
Tipico di:

- **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



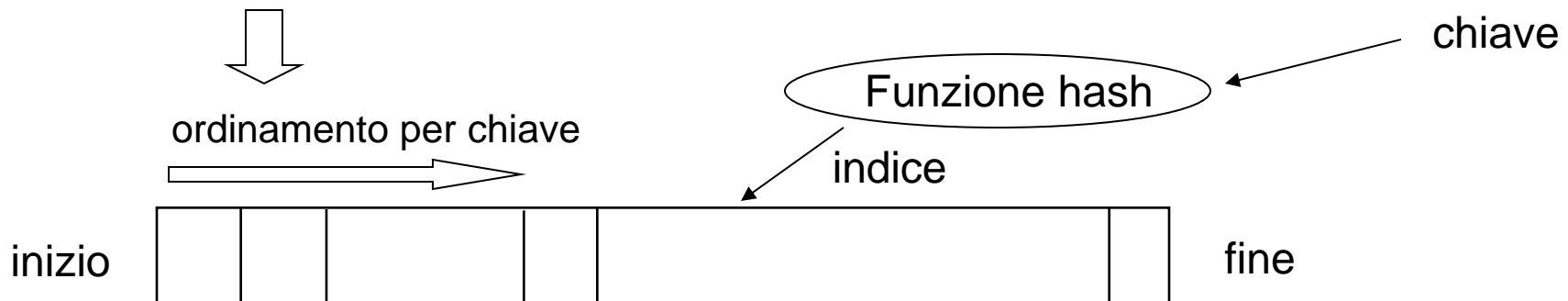
- esiste un file sequenziale di indici associato a ciascun file di dati
- i record sono ordinati per “chiave”
- tramite il file di indici ci si può posizionare in punti specifici del file di dati (ovvero in punti con valori specifici del campo chiave)
- i records vengono acceduti sequenzialmente una 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 puo' avvenire in un qualsiasi punto del file
- si puo' accedere direttamente all'i-esimo record (senza necessariamente accedere ai precedenti)
- dopo un accesso all'iesimo 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 e' 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 cosi' detta **struttura di directory**

Tipica struttura di directory

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



Blocchi di dispositivo

- Ciascun file è allocato sul dispositivo di memoria di massa come un insieme di blocchi non necessariamente contigui
 - 1) organizzazione fissa: record di taglia fissa (possibilità di frammentazione interna)
 - 2) organizzazione variabile con riporto: record di taglia variabile con possibilità che un record sia suddiviso tra più blocchi
 - 3) organizzazione variabile senza riporto: come 2) ma con possibilità di frammentazione interna
- Le unità di allocazione possono essere costituite da più blocchi (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

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)

	Sector											
Track	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

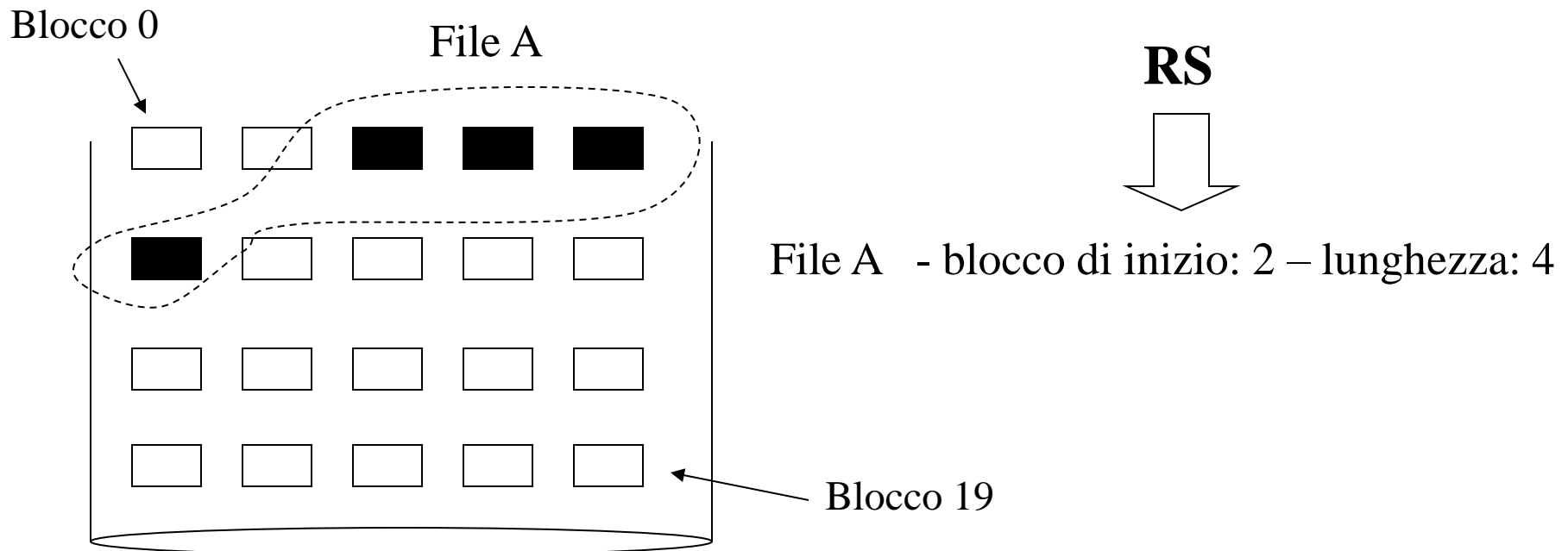
(b)

(a) Lista Libera

(b) Bit Map

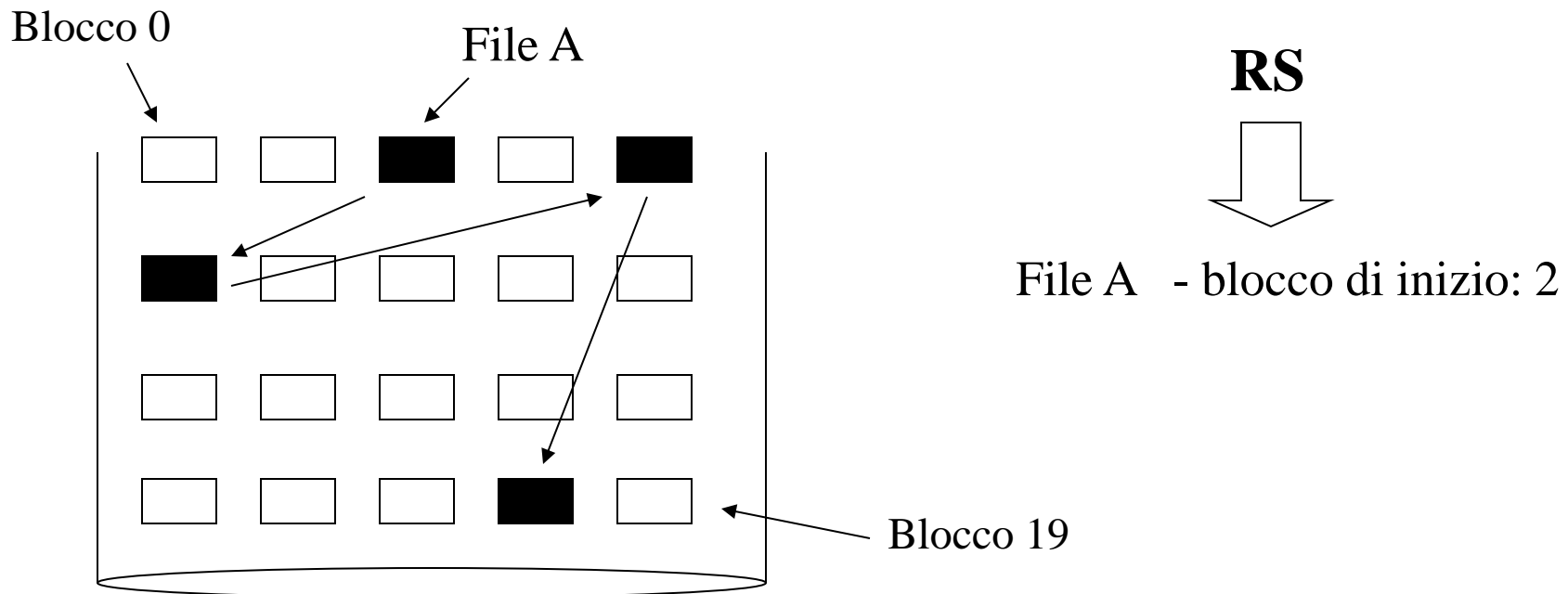
Allocazione di file contigua

- un insieme di blocchi contigui e' allocato per un file all'atto della creazione
- la taglia massima dipende dal numero di blocchi allocati
- l'occupazione reale e' 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



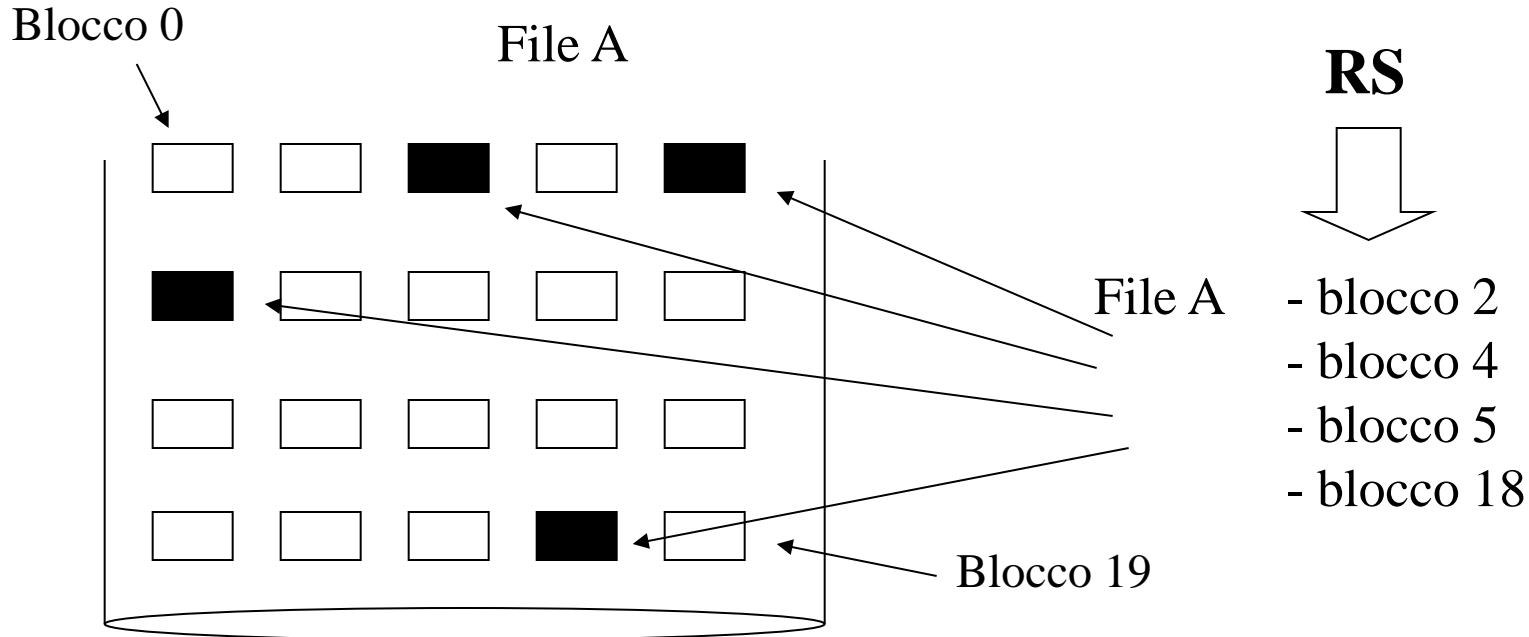
Allocazione di file a catena

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



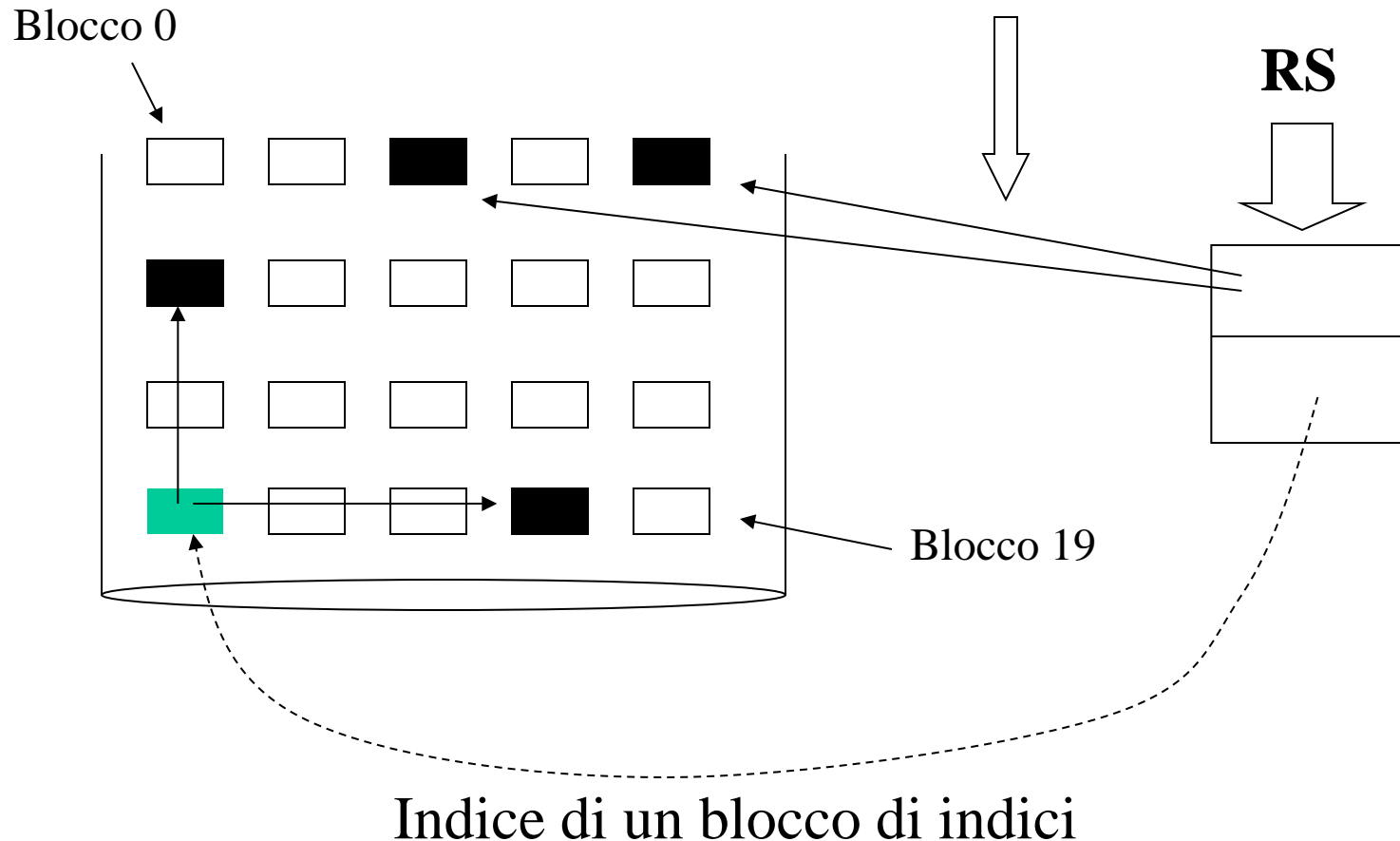
Allocazione di file indicizzata

- i blocchi di un file sono rintracciati tramite un indice
- l'occupazione reale e' sempre pari al numero di blocchi relamente 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



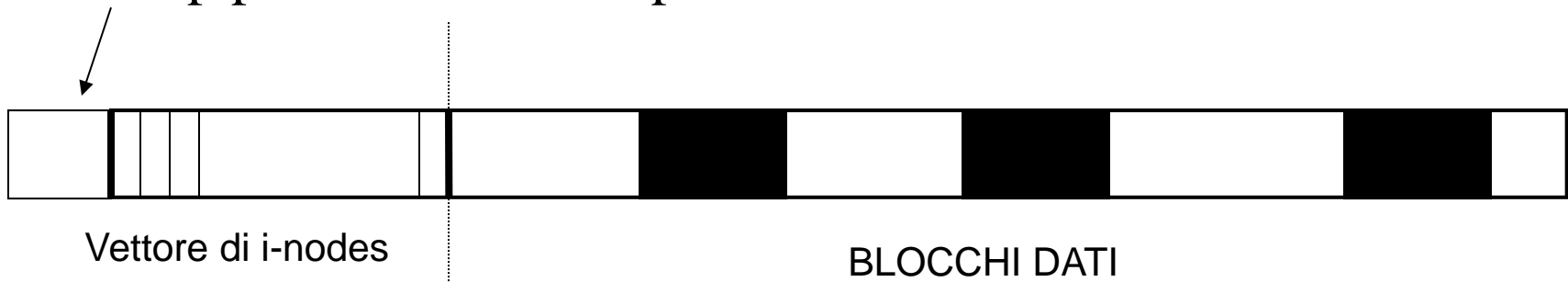
Indici di blocchi di dati possono anche non essere presenti in RS

UNIX file systems

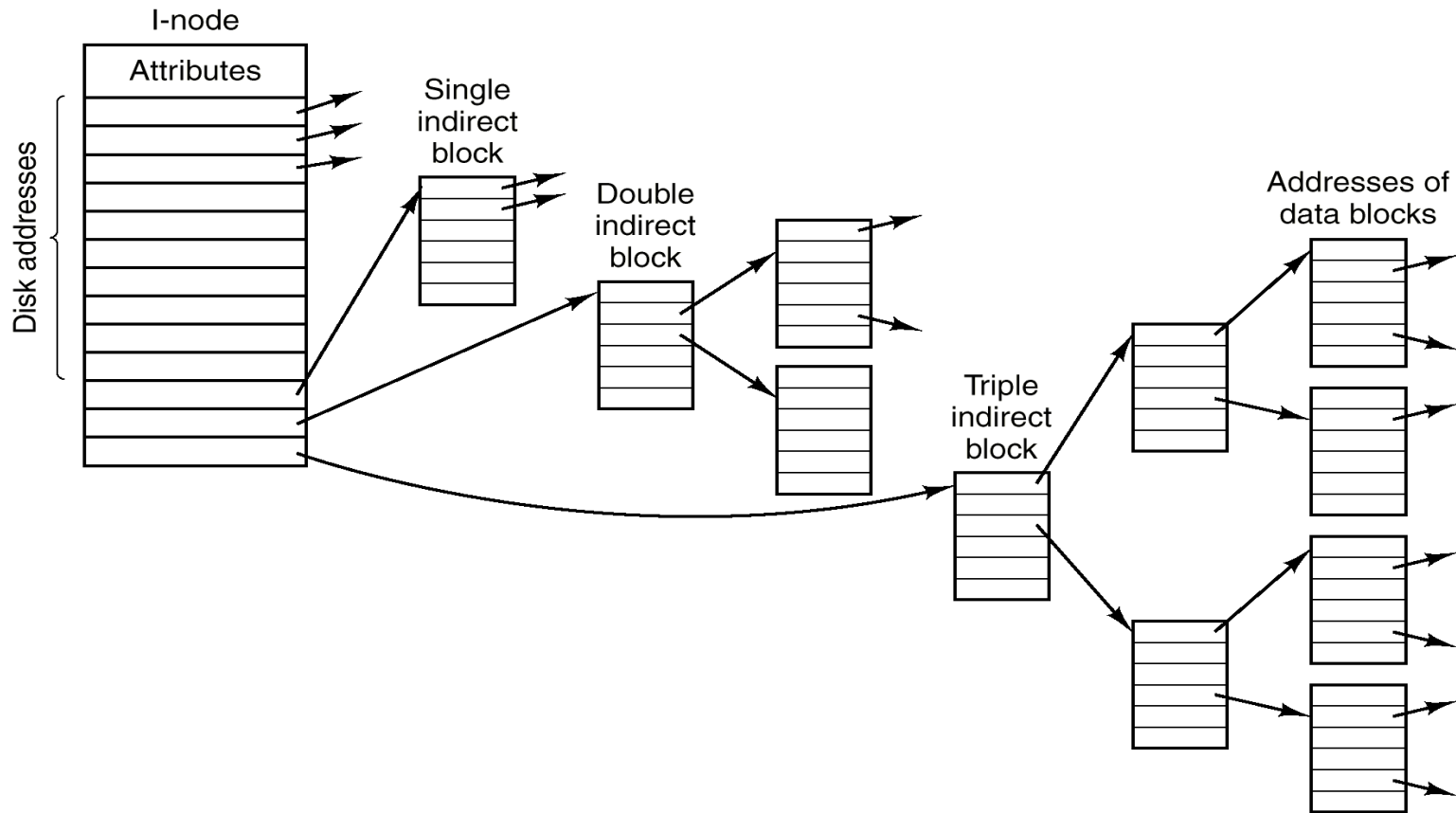
- Berkeley Fast File System also known as UFS (Unix File System) ←———— BSD
 - Ext2
 - Ext3
 - Ext4
 - Btrfs (Better File System)
 - JFS (Journal FS) ←———— AIX/IBM
-
- LINUX
(ever larger files +
journaling +
deduplication +
other)

Caratteristiche di base dei file system UNIX

- Ogni file e' 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: 4 o piu' bytes (numero di i-node) – 2 o piu' bytes (spiazzamento per la entry successiva) – 2 o piu' bytes (lunghezza del nome) – x bytes (nome)
- Bit map per blocchi dati e per i-nodes



Struttura di un i-node



Taglia massima di un file

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

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

Attributi

{-,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)

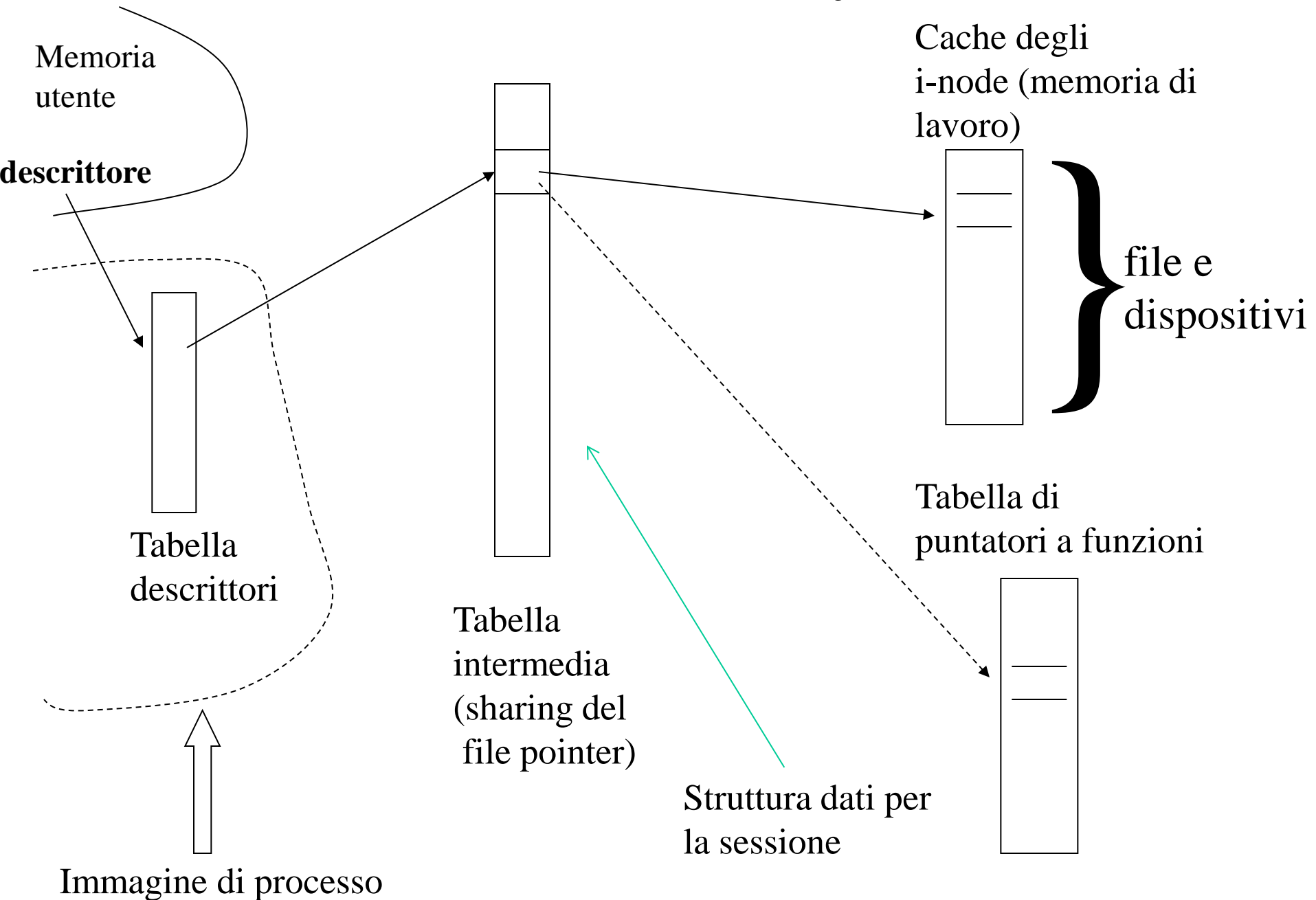
SGID (1 bit)

specifica di identificazione dinamica
per chi utilizza il file

Sticky (1 bit)

per le directory rimuove la possibilità
di cancellare files

UNIX virtual file system (I)



UNIX virtual file system (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)
- 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 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

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

Esempio

```
#include <stdio.h>

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

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 modalita' 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”

- `_RDONLY`: apertura del file in sola lettura;
- `_WRONLY`: apertura del file in sola scrittura;
- `_RDWR`: apertura in lettura e scrittura;
- `_APPEND`: apertura del file con puntatore alla fine del file; ogni scrittura sul file sara' effettuata a partire dalla fine del file;
- `_CREAT` : crea il file con modalita' d'accesso specificate da *mode* solo se esso stesso non esiste;
- `_TRUNC` : elimina il contenuto del file se esso gia' esistente.
- `_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)

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: quantità 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

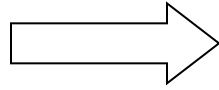
Argomenti

- 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 eredita’ 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 l'operativita' `close-on-exec` (tramite la system-call `fcntl()` o il flag `O_CLOEXEC` in apertura del file)

Un esempio di applicazione: il comando “cp”

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 1024

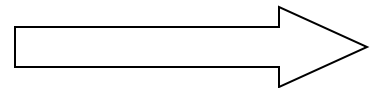
int main(int argc, char *argv[]) {
    int sd, dd, size, result; char buffer[BUFSIZE];

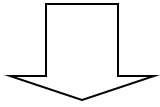
    if (argc != 3) { /* Controllo sul numero di parametri */
        printf("usage: copia source target\n");
        exit(1);
    }

    /* Apertura del file da copiare in sola lettura */
    sd=open(argv[1],O_RDONLY);
    if (sd == -1) {
        printf("source open error");
        exit(1);
    }

    /* Creazione del file destinazione */
    dd=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0660);
    if (dd == -1) {
        printf("destination open error");
        exit(1);
    }
}
```

continua





```
/* Qui iniziano le operazioni di copia */
do {
    /* Lettura di massimo di BUFSIZE caratteri */
    size=read(sd,buffer,BUFSIZE);
    if (size == -1) {
        printf("read error");
        exit(1);
    }
    /* Lettura fino ad un massimo di BUFSIZE caratteri */
    result = write(dd,buffer,size);
    if (result == -1) {
        printf("write error");

        exit(1);
    }

} while(size > 0);

close(sd);
close(dd);
}/* end main*/
```

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 redirectione

```
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 e' il primo libero nella tabella dei descrittori del processo

Esempio

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

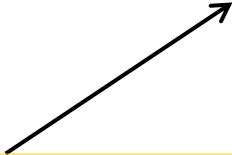
int main() {
    int fd;
    fd = open(FNAME, O_RDONLY); /* Apro il file in lettura */
    close(STDIN);               /* Chiudo lo standard input */
    dup(fd);                    /* Duplico il descrittore di file */
    execlp("more", "more", 0); /* Eseguo 'more' con input
                       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

Nuovi standard

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

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

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



Specifica esplicita della posizione della file-descripto table ove duplicare il canale originale

Controllo di canale

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.

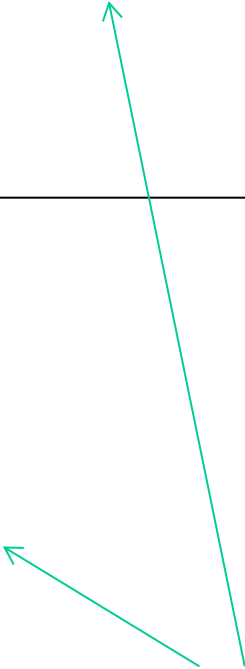
Hard links e rimozione

```
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 in file speciale rappresentate una directory

Symbolic links

- Sono file il cui contenuto identifica una stringa che definisce parte o tutto 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)
```



Creazione/rimozione di directory

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 fildes is changed.

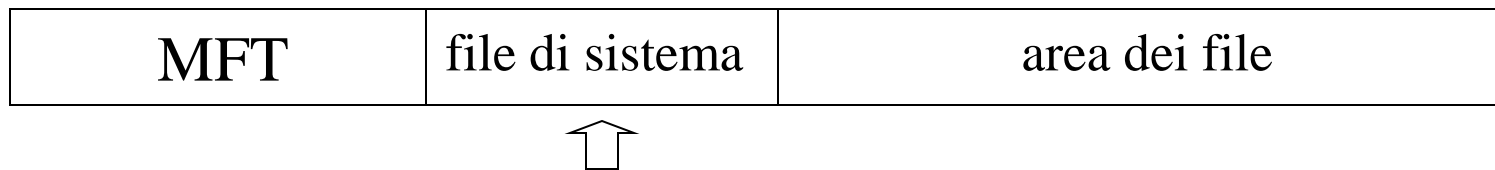
varianti **chown** gestiscono il cambio di proprietà

Comandi shell basici per l'interazione con il File System

link	makes hard links
ln	makes symbolic links
dumpe2fs	dump file system info
mkfs	installs a file system on a device

NTFS

- dischi divisi in volumi (partizioni), e organizzati in cluster da 512 a 64K byte
- per ogni volume si ha una **MFT** (**M**aster **F**ile **T**able)
- ad ogni file corrisponde un elemento nella MFT, di 1K o dimensione del cluster
- 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



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

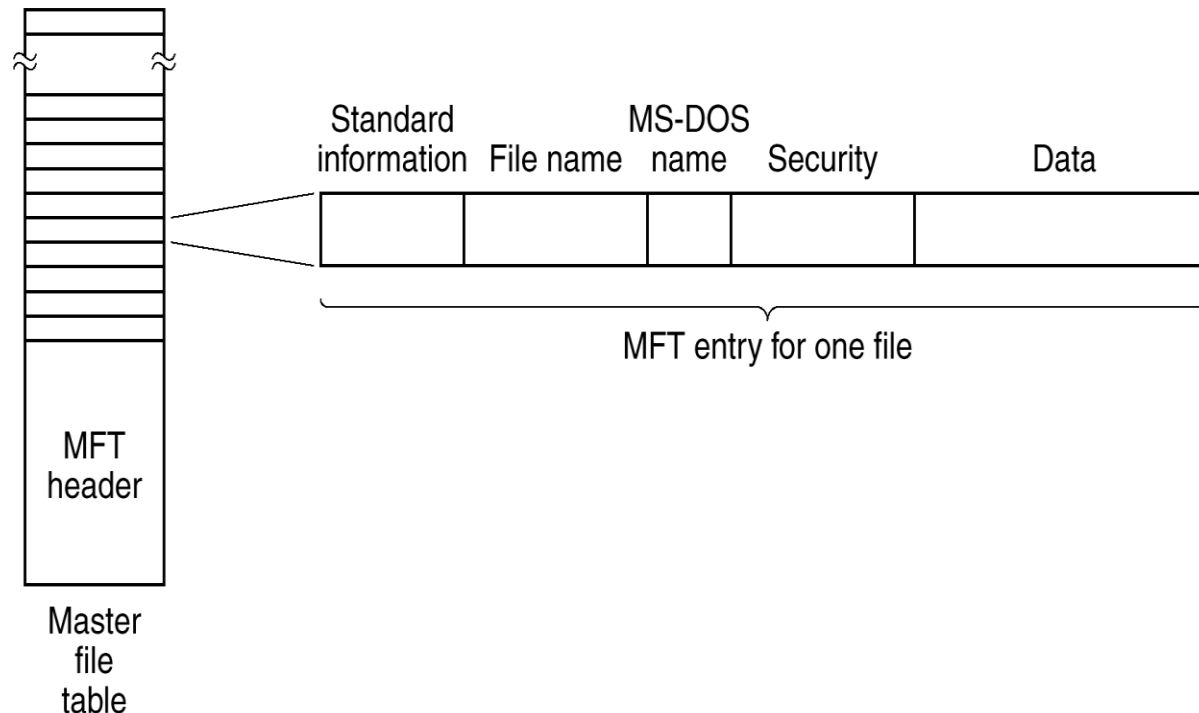
identificazione nel sistema

Dati

**visti tipicamente come attributo senza
nome (ci possono comunque essere uno
o piu' attributi dati con nome)**

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 contiene gli indirizzi di cluster o di gruppi di cluster consecutivi
- se un elemento della MFT non basta si aggrega il successivo



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 i permessi di accesso al file da creare (GENERIC_READ, GENERIC_WRITE)
- dwShareMode: specifica se e quando il file puo' 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, OPEN_ALWAYS, TRUNCATE_EXISTING)
- dwFlagsAndAttributes: specifica varie caratteristiche del file (FILE_ATTRIBUTES_NORMAL, ...HIDDEN, ...READONLY, ..DELETE_ON_CLOSE)
- hTemplateFile: specifica un handle ad un file di template

Codifiche sui pathname

CreateFileA (...) ANSI

CreateFileW(...) UNICODE

ACL (Access Control List)

- specifica la descrizione della sicurezza di oggetti, quindi anche di file
- e' formata da una lista di ACE (Access Control Entry)
- nel caso di generazione di oggetti in cui il descrittore di sicurezza non e' specificato, ACL viene popolata a partire dall'*access token* del processo chiamante
- in tal caso si ha una ACL di default
- ACL e' 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

Manipolazione di ACL

GetNamedSecurityInfo

GetSecurityInfo

SetNamedSecurityInfo

SetSecurityInfo



System call basiche

A queste system-call vengono associate una serie di 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.

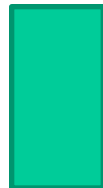
Syntax

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



Object types

```
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;
```

ACL structure

The **ACL** structure is the header of an *access control list* (ACL). A complete ACL consists of an **ACL** structure followed by an ordered list of zero or more *access control entries* (ACEs).

Syntax

C++

```
typedef struct _ACL {  
    BYTE AclRevision;  
    BYTE Sbz1;  
    WORD AclSize;  
    WORD AceCount;  
    WORD Sbz2;  
} ACL, *PACL;
```


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

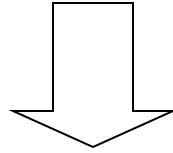


buffer sizes



Chiusura di file

BOOL CloseHandle(HANDLE hObject)



chiude un oggetto

Descrizione

- invoca la chiusura di un file

Restituzione

- 0 in caso di fallimento

Lettura da file

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

Descrizione

- invoca la lettura di una certa quantita' 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

Un esempio di applicazione: il comando “copy”

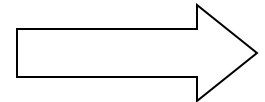
```
#include <windows.h>
#include <stdio.h>
#define BUFSIZE 1024

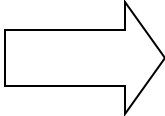
int main(int argc, char *argv[]) {
    HANDLE sd, dd; DWORD size, result; char buffer[BUFSIZE];

    if (argc != 3) { /* Controllo sul numero di argomenti */
        printf("usage: copia source target\n");
        exit(1);
    }
    /* Apertura del file da copiare in sola lettura */
    sd=CreateFile(argv[1],
                  GENERIC_READ,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

    if (sd == INVALID_HANDLE_VALUE) {
        printf("Cannot open source file\n");
        ExitProcess(1);
    }
```

continua





```
/* Creazione del file destinazione */
    dd=CreateFile(argv[2],
                  GENERIC_WRITE,
                  0,
                  NULL,
                  CREATE_ALWAYS,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

    if (dd == INVALID_HANDLE_VALUE) {
        printf("Cannot open destination file\n");
        ExitProcess(1);
    }

    do { /* Qui iniziano le operazioni di copia */
        if (ReadFile(sd,buffer,BUFSIZE,&size,NULL) == 0) {
            printf("Cannot read from source file\n");
            ExitProcess(1);
        }
        if (WriteFile(dd,buffer,size, &result, NULL) == 0){
            printf("Cannot write to destination file\n");
            ExitProcess(1);
        }
    } while(size > 0);
    CloseHandle(sd);
    CloseHandle(dd);
    return(0);
}
```


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 piu' significativi del valore in **lDistanceToMove**
- **dwMoveMethod**: tipo di spostamento da effettuare (**FILE_BEGIN**, **FILE_CURRENT**, **FILE_END**)

Duplicazione di handle: sharing del file pointer

```
BOOL DuplicateHandle(HANDLE hSourceProcessHandle,  
                    HANDLE hSourceHandle,  
                    HANDLE hTargetProcessHandle,  
                    LPHANDLE lpTargetHandle,  
                    DWORD dwDesiredAccess,  
                    BOOL bInheritHandle,  
                    DWORD dwOptions)
```

Descrizione

- invoca la duplicazione di un handle di file da un processo ad un altro

Restituzione

- 0 in caso di fallimento

Parametri

- **hSourceProcessHandle**: handle del processo di cui si vuole duplicare un handle di file
- **hSourceHandle**: handle di file che si vuole duplicare
- **hTargetProcessHandle**: handle del processo a cui si vuole passare la copia dell'handle di file
- **lpTargetHandle**: puntatore a una variabile di **hTargetProcessHandle** in cui viene copiato l'handle di file che si vuole duplicare
- **dwDesiredAccess**: richieste di accesso per l'handle che si vuole duplicare
- **bInheritHandle**: Indica se il nuovo handle può essere ereditato da processi figli
- **dwOptions**: opzioni (es. **DUPLICATE_CLOSE_SOURCE**), 0 è il default

Gestione degli standard handle

HANDLE WINAPI GetStdHandle(_In_ DWORD nStdHandle);

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

Value	Meaning
STD_INPUT_HANDLE (DWORD)-10	The standard input device.
STD_OUTPUT_HANDLE (DWORD)-11	The standard output device.
STD_ERROR_HANDLE (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 );
```

FindFirstFile function

Searches a directory for a file or subdirectory with a name that matches a specific name (or partial name if wildcards are used).

To specify additional attributes to use in a search, use the [FindFirstFileEx](#) function.

To perform this operation as a transacted operation, use the [FindFirstFileTransacted](#) function.

Syntax

C++

```
HANDLE WINAPI FindFirstFile(  
    _In_ LPCTSTR lpFileName,  
    _Out_ LPWIN32_FIND_DATA lpFindFileData  
);
```


FindNextFile function

Continues a file search from a previous call to the **FindFirstFile**, **FindFirstFileEx**, or **FindFirstFileTransacted** functions.

Syntax

C++

```
BOOL WINAPI FindNextFile(  
    _In_   HANDLE          hFindFile,  
    _Out_  LPWIN32_FIND_DATA lpFindFileData  
);
```

NTFS hard links

Syntax

```
BOOL WINAPI CreateHardLink(  
    __in LPCTSTR lpFileName,  
    __in LPCTSTR lpExistingFileName,  
    __reserved 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

Dispositivi fisici ed I/O scheduling

Tipologie

Interattivi

- terminali video
- tastiera mouse
- stampanti

Non-interattivi

- dischi
- nastri
- controller ed attuatori

Di comunicazione

- modem
- schede di rete

Diversificati in base a

- velocità di trasferimento
- complessità del controllo
- unità di trasferimento (blocco)
- rappresentazione dei dati (codifiche)
- condizioni di errore e relativo rilevamento

Tecniche per l'I/O sui dispositivi fisici

I/O programmato

- il processore rimane in attesa attiva fino al completamento dell'interazione con lo specifico dispositivo

I/O interrupt-driven

- il processore impartisce un comando al dispositivo di I/O
- torna poi ad eseguire le istruzioni successive e viene interrotto dal dispositivo quando esso ha completato il lavoro richiesto
- le istruzioni successive al comando verso il dispositivo possono o non appartenere al processo che lancia il comando

DMA

- un dispositivo DMA controlla lo scambio di dati tra memoria e dispositivi di I/O
- il dispositivo DMA interrompe il processore nel momento in cui operazioni di I/O precedentemente richieste sono completate

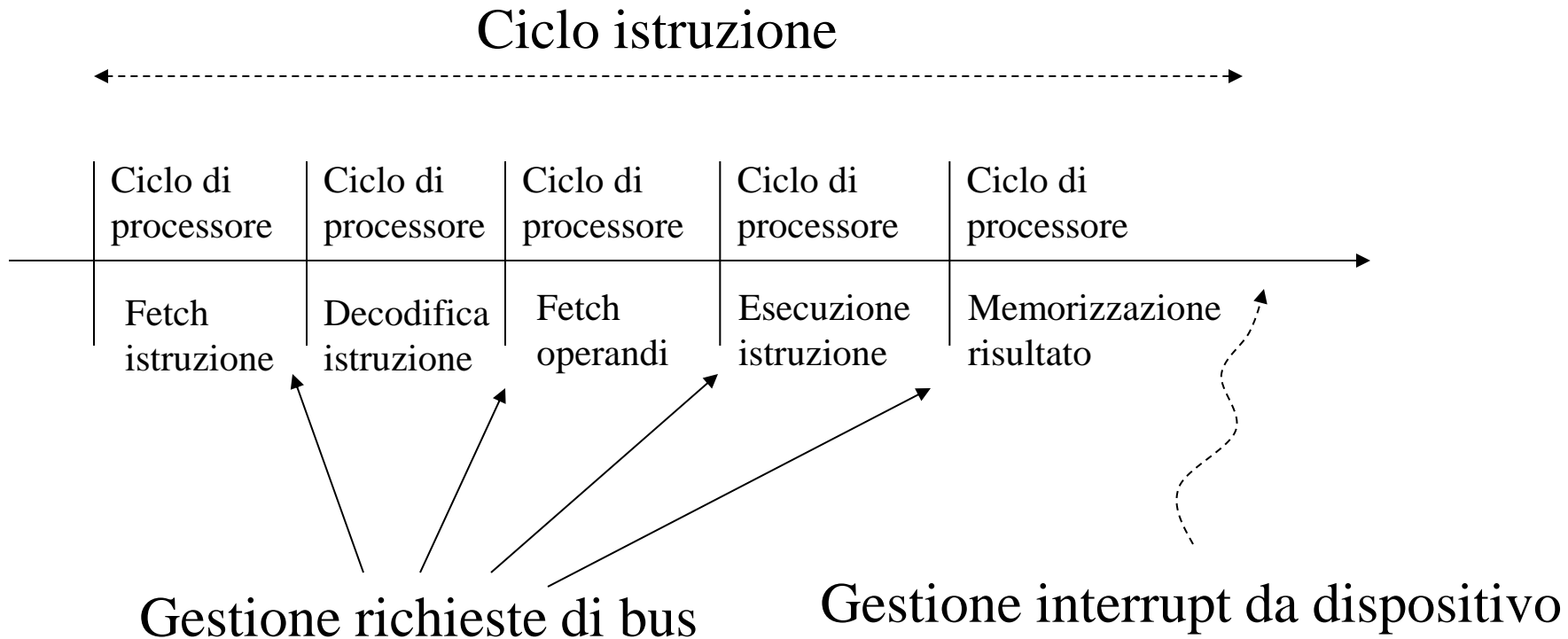
Evoluzione storica

1. Driver a carico del programmatore dell'applicazione, I/O programmato
2. Driver preprogrammato, I/O programmato
3. Driver preprogrammato, I/O interrupt driven
4. Driver preprogrammato, DMA

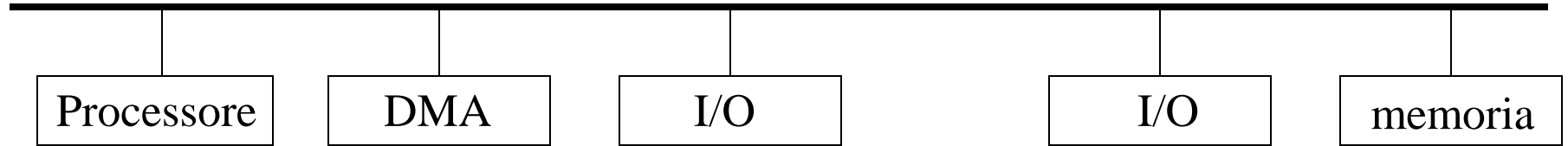
	Senza interrupt	Con interrupt
Trasferimento di I/O tramite processore	I/O programmato	I/O interrupt-driven
Trasferimento di I/O tramite DMA		DMA

DMA ed interrupt

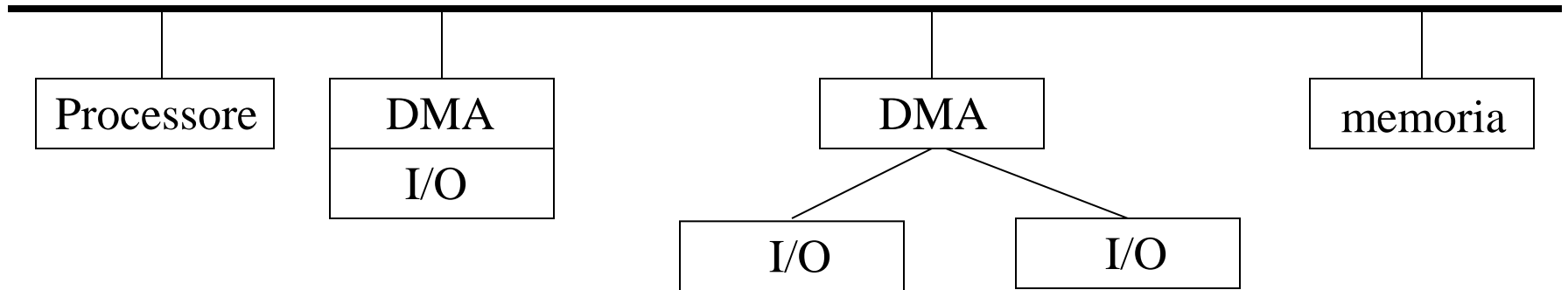
- le interruzioni relative alla richiesta di bus vengono gestite ad ogni ciclo di processore
- le interruzioni dei dispositivi (compreso il DMA) vengono gestite tra una istruzione macchina e l'altra



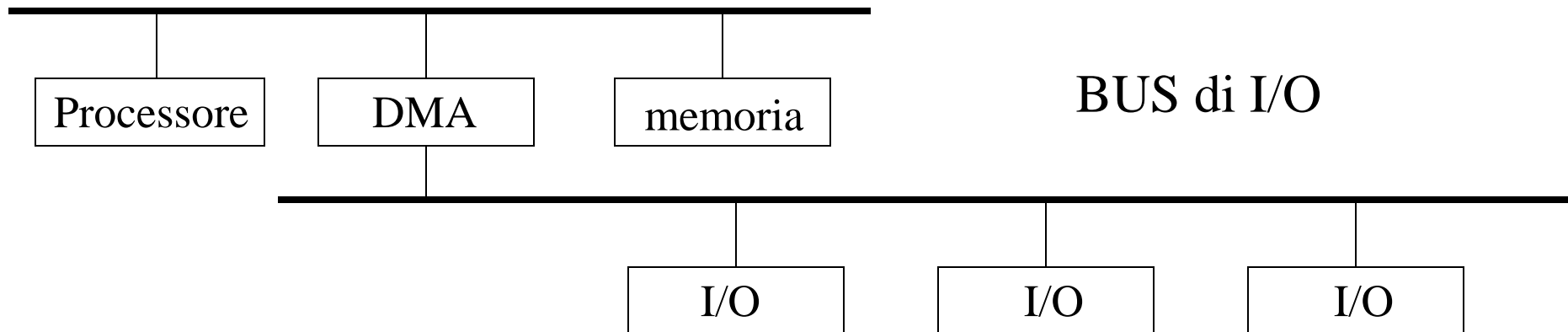
Configurazioni



BUS singolo - DMA separato



BUS singolo - DMA e I/O integrati



BUS di I/O

Progettazione della funzione di I/O: obiettivi

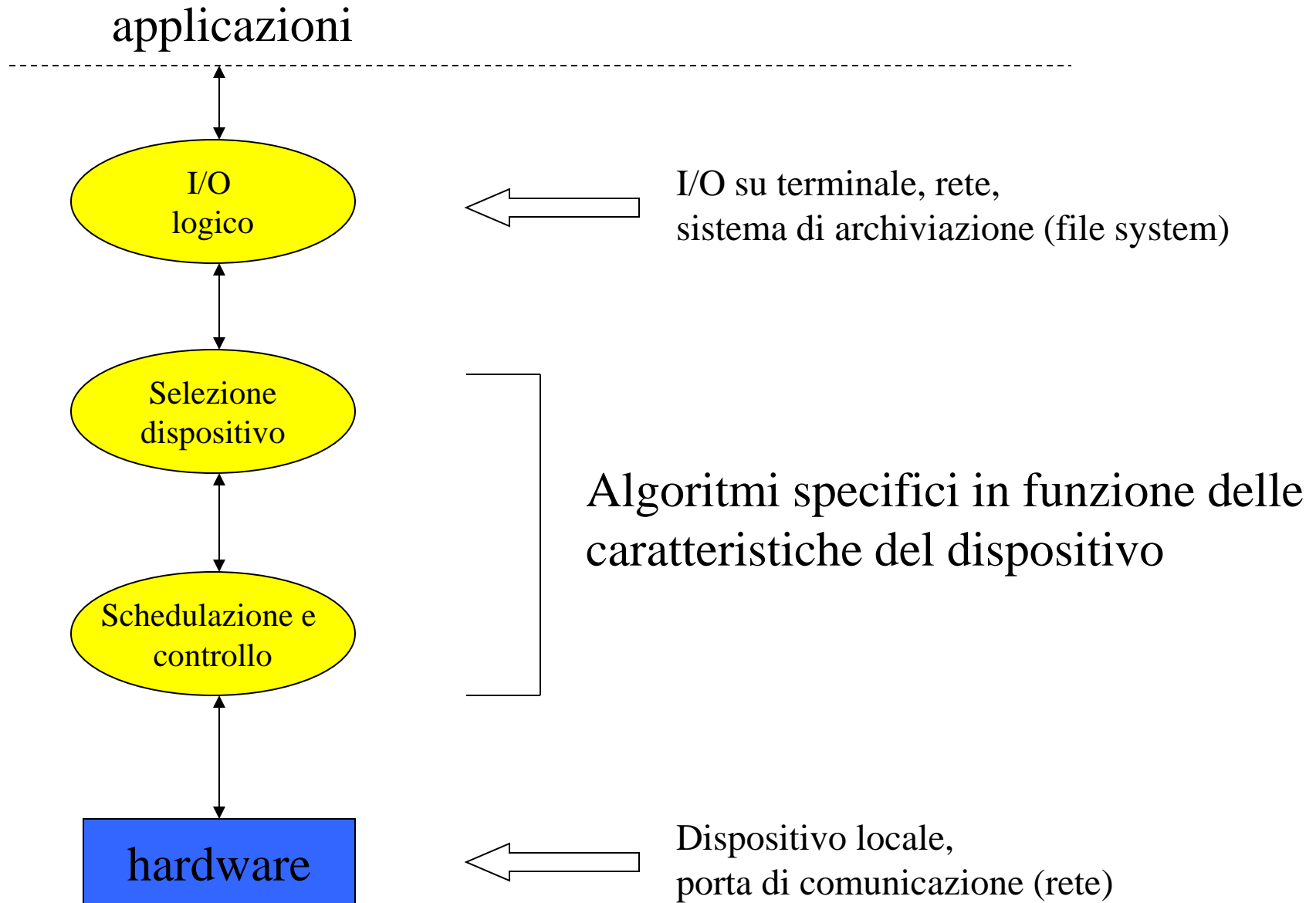
Efficienza

- l'interazione con i dispositivi è tipicamente il collo di bottiglia
- lo swapping tende ad aumentare il throughput tramite incremento del livello di multiprogrammazione
- lo swapping richiede però I/O efficiente per essere applicabile
- necessità di algoritmi efficienti per la gestione dell'I/O su disco

Generalità

- uniformità di trattamento dei dispositivi da parte delle applicazioni
- fornitura di servizi di I/O con punti di accesso (interfacce) standard indipendentemente dalla tipologia di dispositivo
- approccio gerarchico alla progettazione, che nasconda i dettagli di più basso livello

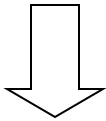
Un modello di organizzazione



Necessità di bufferizzazione

I/O effettuato direttamente sulla memoria riservata ai processi può provocare

1. Sottoutilizzo dei dispositivi e della CPU nel caso in cui l'area di memoria destinata per l'I/O non sia “swappabile”
2. Deadlock nel caso in cui l'area di memoria destinata all'I/O sia swappabile (processo bloccato in attesa dell'I/O – I/O bloccato in attesa che il processo venga riportato in memoria)

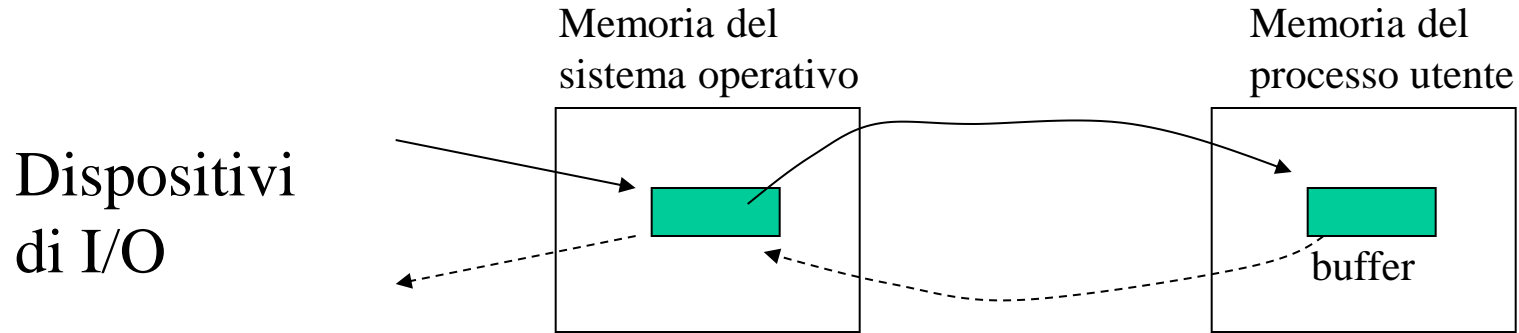


- l'I/O viene quindi tipicamente effettuato su aree di memoria riservate al sistema operativo
- questo introduce la necessità di gestire buffer destinati alle operazioni di I/O

Orientamento

- a **blocchi**: bufferizzazione e trasferimento di blocchi di bytes
- a **flusso** (stream): bufferizzazione e trasferimento di singoli bytes

Singolo buffer



- viene assegnato un singolo buffer per la gestione delle operazioni di I/O
- in fase di input, al momento della copia del contenuto del buffer in spazio utente, viene effettuata la lettura del blocco successivo (**lettura in avanti – input anticipato**)
- la logica di swapping può essere influenzata nel caso lo swapping sia effettuato sullo stesso dispositivo di I/O coinvolto nelle operazioni
- sottoutilizzo del buffer nel caso di I/O orientato a flusso (problema minore nel caso di gestione di sequenza di caratteri a linee)

- una soluzione generale per il caso di I/O orientato a flusso è lo schema di sincronizzazione produttore/consumatore con granularità al byte applicato tra processo utente e sistema operativo

Prestazioni

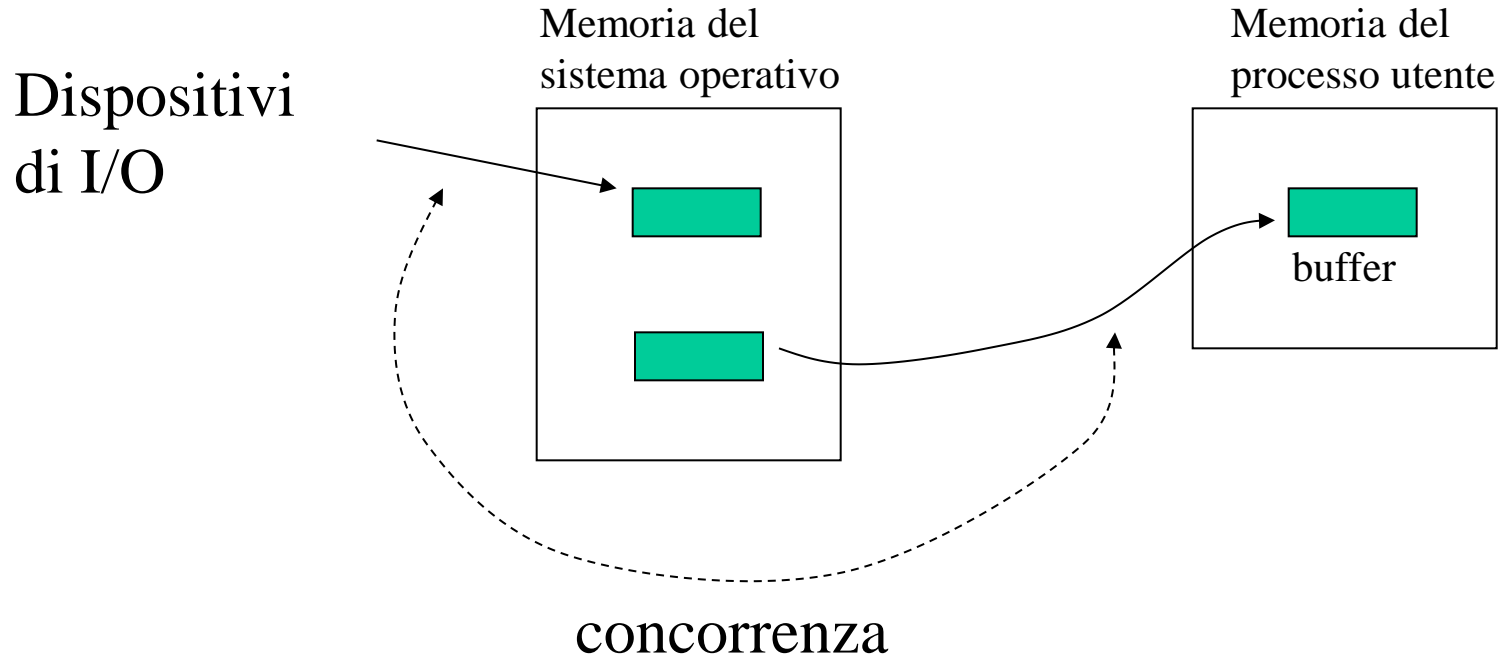
T = tempo per l'input di un blocco

C = tempo tra due richieste di input

M = tempo per la copia del blocco nel buffer utente

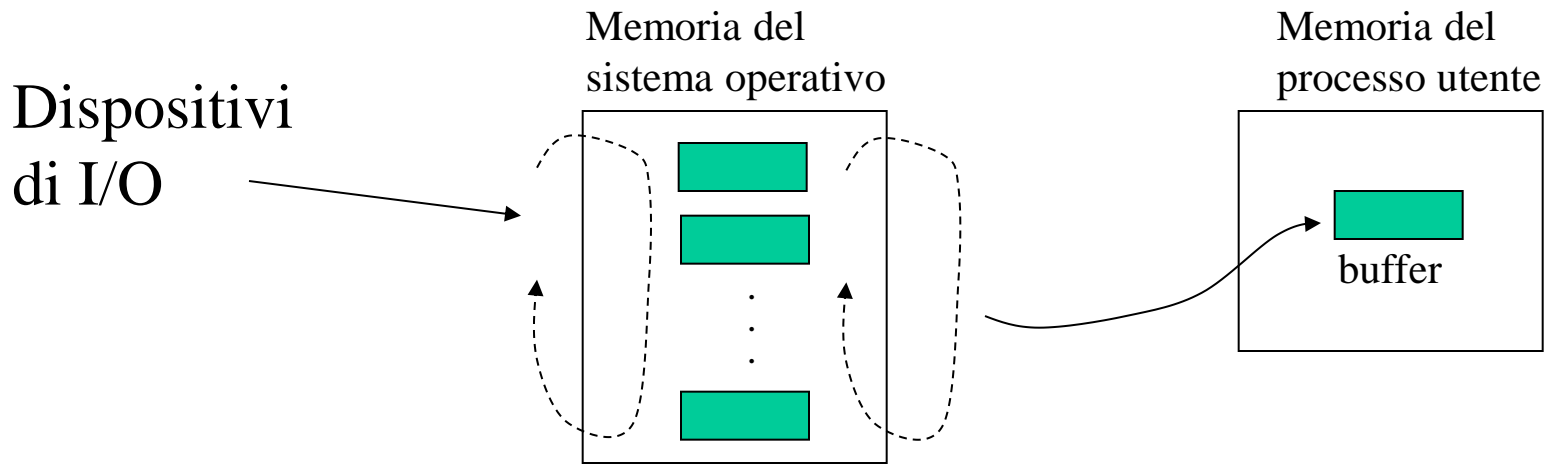
$$\text{Latenza} = \max[C, T] + M$$

Buffer doppio



- per I/O a blocchi, da vantaggi nel caso di brevi sequenze di richieste di I/O quando $C < T$
- non produce particolari vantaggi nel caso di I/O orientato a flusso di byte

Buffer circolare



- per I/O a blocchi, produce vantaggi nel caso di lunghe sequenze di richieste di I/O quando $C < T$
 - non produce particolari vantaggi nel caso di I/O orientato a flusso di byte
-

Note

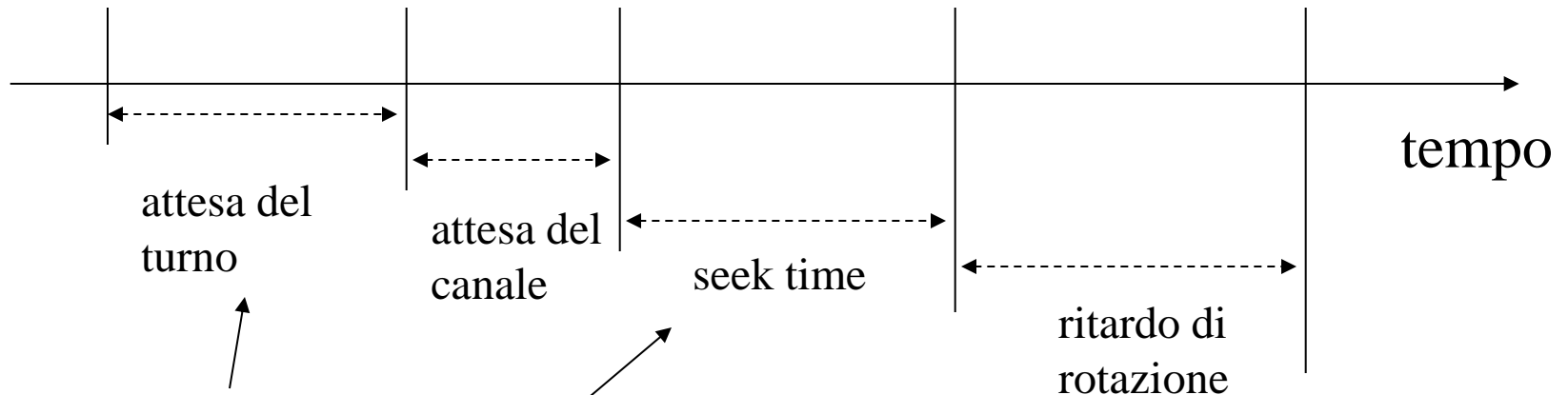
- nessuna quantità di buffer sarà mai sufficiente per evitare il blocco dei processi nel caso $C < T$ ed i processi siano esclusivamente I/O bound
- in pratica i processi esibiscono comportamenti misti (I/O e CPU bound), quindi una quantità di buffer limitata favorisce l'utilizzo efficiente di risorse e la minimizzazione del tempo di turnaround

Schedulazione dei dischi magnetici a rotazione

Parametri

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

Punto di vista del processo



dipendenza dalla
politica di schedulazione

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.
- $r = 300/3600$ rpm (floppy vs hard disk)

fattore critico

Scheduling FCFS

- 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

$$\text{lunghezza media di ricerca} = \frac{\sum_{i=1}^{|insieme\ dist|} dist_i}{|insieme\ dist|} = 55.3$$

Scheduling su base priorità

- la sequenzializzazione delle richieste di I/O è fuori dal controllo del software di gestione del disco
- si tende a favorire processi con più alta priorità
- l'obiettivo non è l'ottimizzazione dell'utilizzo del disco (non viene minimizzato il seek time)
- rischio di starvation

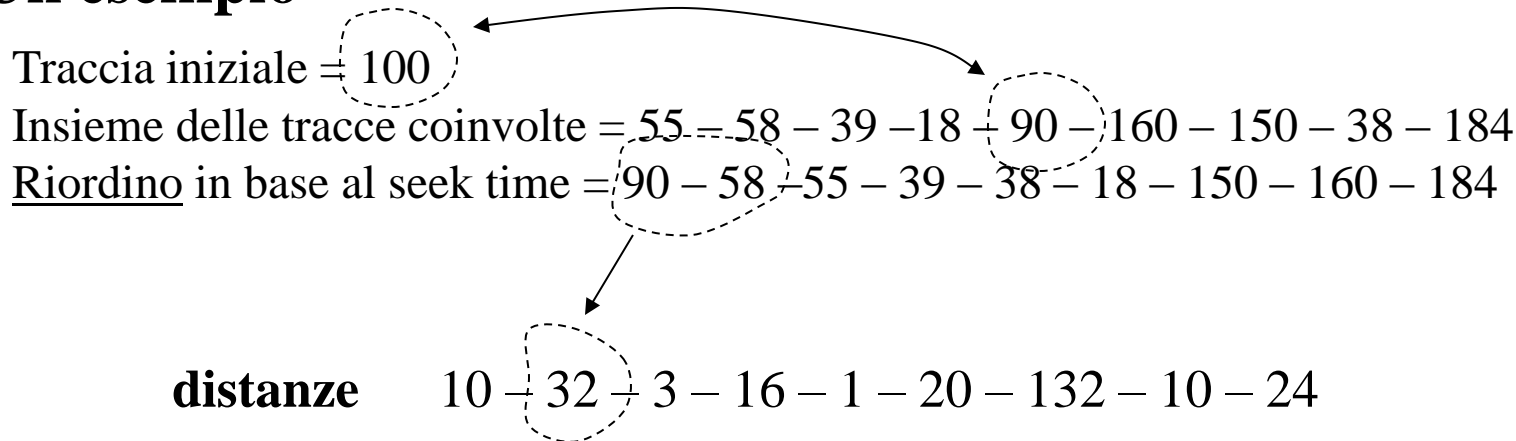
Scheduling LIFO

- le richieste di I/O vengono servite nell'ordine inverso rispetto all'ordine di arrivo
- può produrre starvation
- minimizza il seek time in caso di accessi sequenziali e allocazione contigua (sistemi di basi di dati)

Scheduling SSTF (Shortest Service Time First)

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

Un esempio



lunghezza media di ricerca

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

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

|insieme dist|

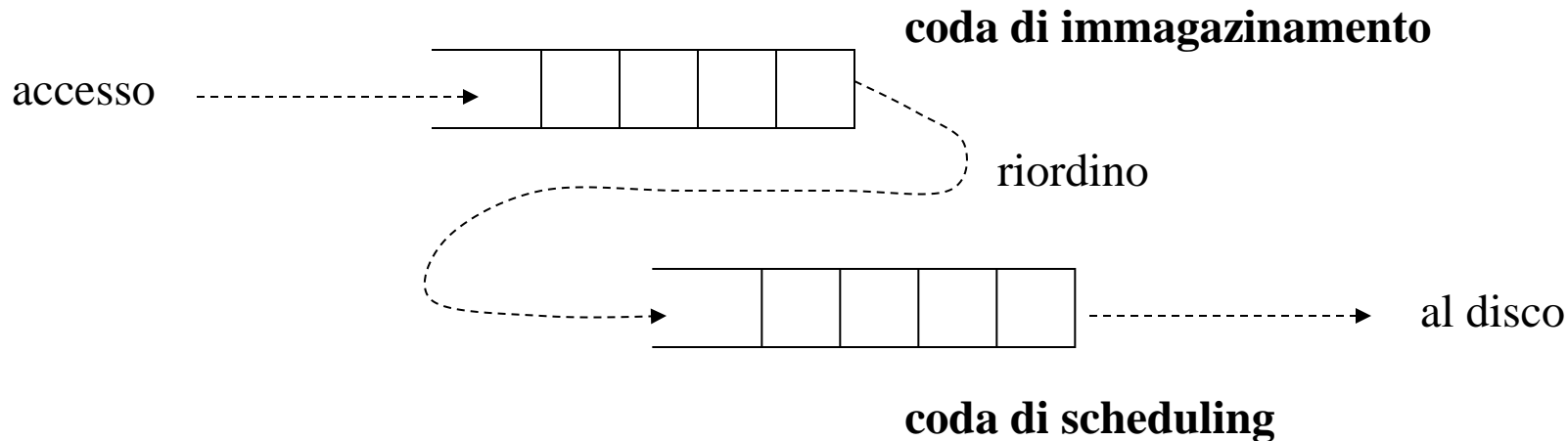
$$\sum_{i=1} dist_i$$

lunghezza media di ricerca

$$\frac{\sum_{i=1} dist_i}{|insieme dist|} = 27.8$$

Scheduling FSCAN

- 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

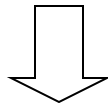


Cache dei dispositivi

- il sistema operativo mantiene una regione di memoria che funge da buffer temporaneo (**buffer cache**) per i dati acceduti in I/O
- hit nel buffer cache evita l'interazione con il disco (diminuzione della latenza e del carico su disco)
- efficienza legata alla località delle applicazioni

Strategia di sostituzione dei blocchi

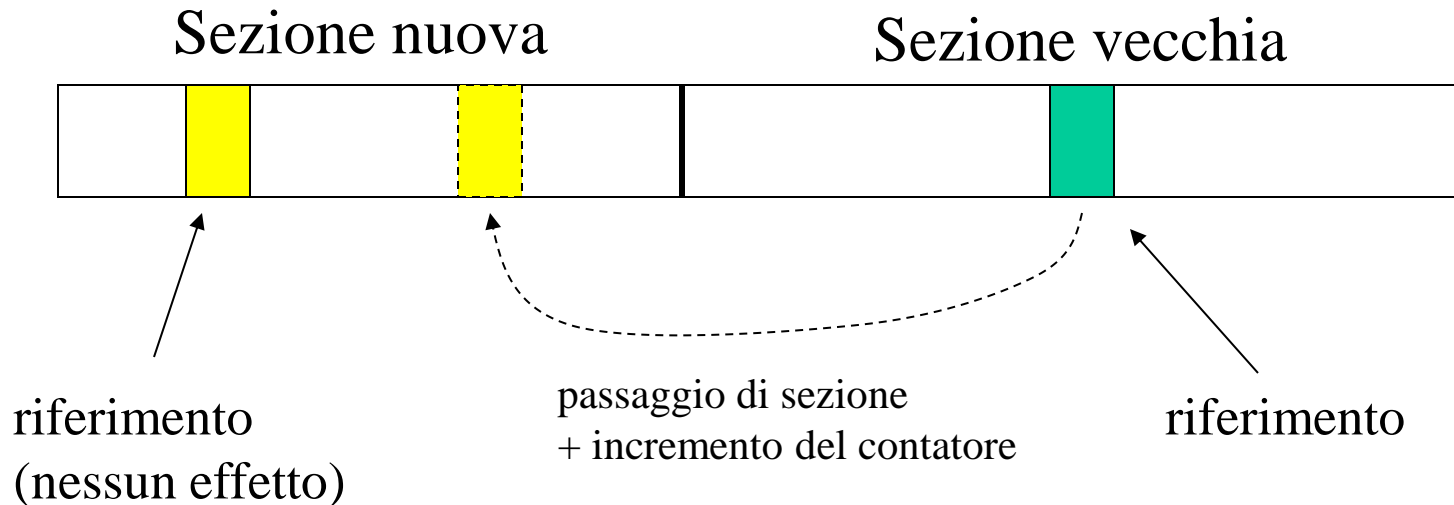
- Least-Recently-Used: viene mantenuta una lista di gestione a stack
- Least-Frequently-Used: si mantiene un contatore di riferimenti al blocco indicante il numero di accessi da quando è stato caricato



Problemi sulla variazione di località

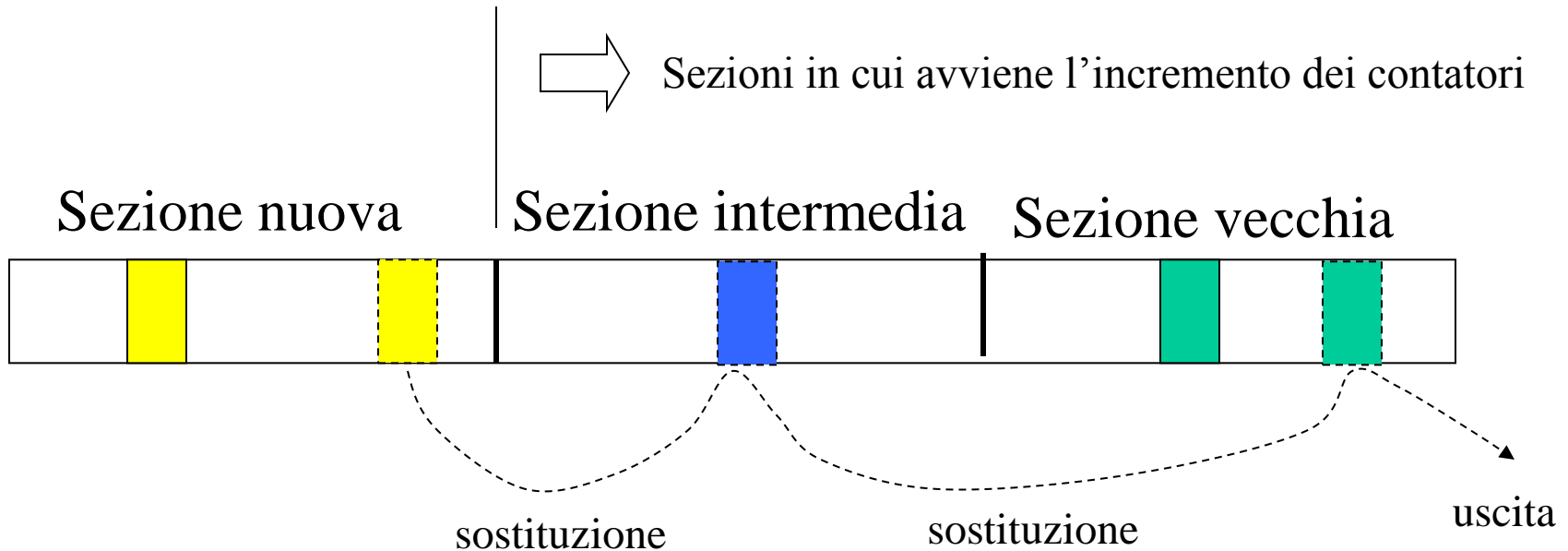
Buffer cache a due sezioni

- 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

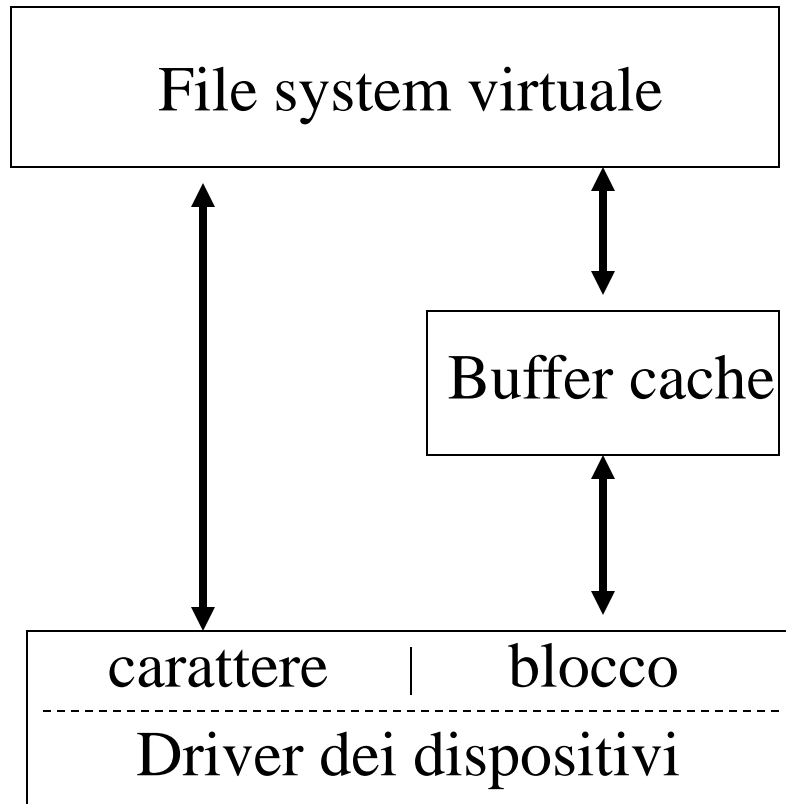
- file system

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

Architettura di I/O in sistemi UNIX



Strutture dati

- lista dei buffer liberi
- lista dei buffer attualmente associati ad ogni dispositivo
- lista dei buffer con I/O in corso o in coda sui dispositivi

Modalità

- sincrona
- asincrona (post di receive asincrone - notifica tramite segnale)
- multiplexing
- signal driven (SIGIO – fcntl()/ioctl())

Sostituzione nel buffer cache secondo la politica LRU

Schedulazione del disco

- basata sull'**elevator algorithm** e varianti
- variante LINUX 2.6:
 1. Una richiesta per lo stesso settore o settori adiacenti a quelli di una richiesta pendente viene “fusa” alla richiesta pendente
 2. Se ci sono richieste pendenti da un intervallo non minimale di tempo, la nuova richiesta viene accodata
 3. Altrimenti la nuova richiesta viene inserita nell'ordine consono per la scansione della testina (quindi possibilmente anche in coda)

I/O asincrono: un modo di supportarlo

NAME

fcntl - manipulate file descriptor

SYNOPSIS

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

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

DESCRIPTION

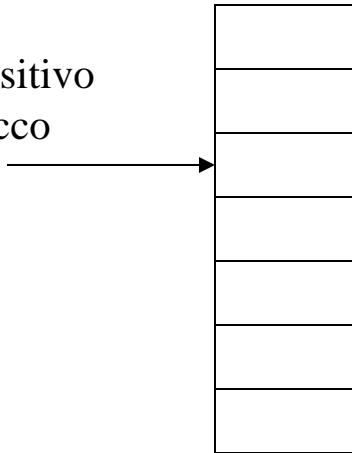
fcntl performs one of various miscellaneous operations on fd. The operation in question is determined by cmd.

If you set the **O_ASYNC** status flag on a file descriptor (either by providing this flag with the open(2) call, or by using the **F_SETFL** command of **fcntl**), a SIGIO signal is sent whenever input or output becomes possible on that file descriptor.

Ricerca nel buffer cache

Tabella hash per la lista
di dispositivi

numero di dispositivo
e numero di blocco



Liste di buffer per dispositivo



•
•
•



Lista di buffer liberi



accesso hash + ricerca lineare

Code di caratteri

- non esiste la cache (i caratteri vengono consumati)
- viene adottato il modello produttore/consumatore applicato a buffer appositi nello spazio di sistema operativo

Quadro riassuntivo

	I/O senza buffer	I/O con buffer cache	I/O con code di caratteri
Disco	X	X	
Nastro	X	X	
Terminali			X
Com.			X
Stampanti	X		X

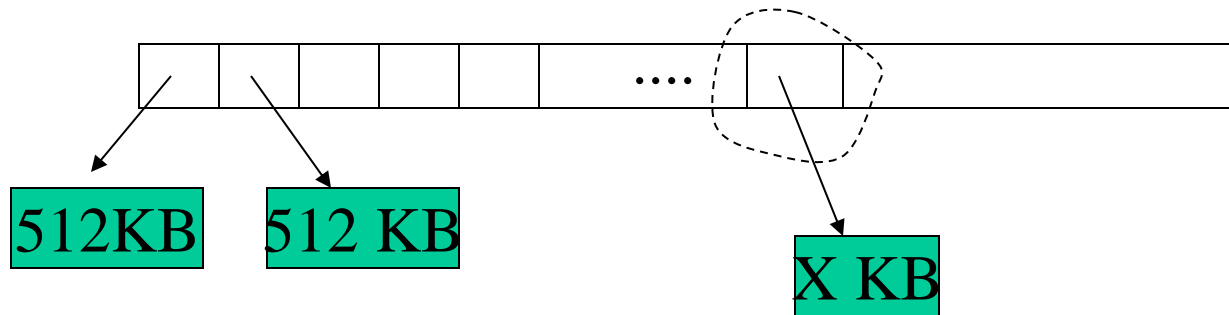
Gestione della partizione di Swapping

- al lancio di un processo si riserva un segmento dell'area di Swap per la parte testo e per i dati
- la parte dati non inizializzata viene tipicamente resettata
- segmenti addizionali di area di Swap possono essere assegnati in caso di necessità (fino ad un limite massimo)
- al lancio del processo, le pagine del segmento testo e dati vengono caricate dal file system nell'area di Swap
- processi istanza della stessa applicazione condividono il segmento di testo nell'area di Swap
- i segmenti dell'area di Swap assegnati ad un dato processo vengono identificati tramite una mappa di Swap associata al processo
- le operazioni di swapping sono attuate da un apposito processo di sistema

Mappe di Swap

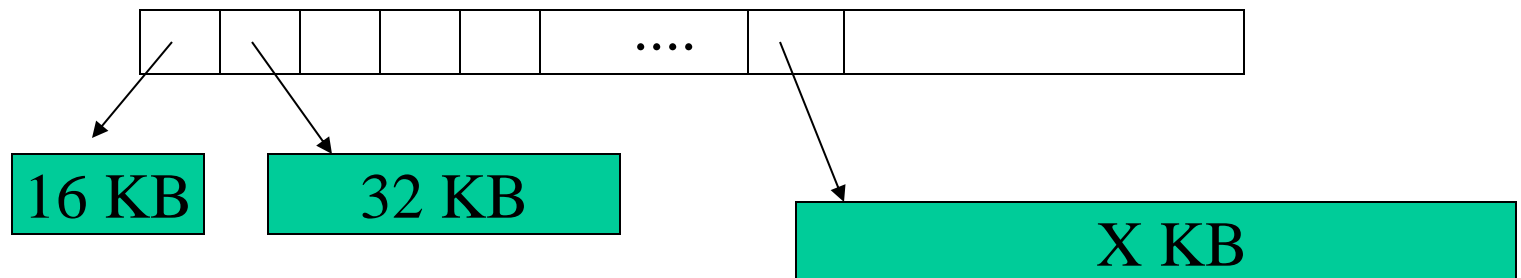
Segmento testo

- granularità ai 512 KB, eccetto l'ultima entry

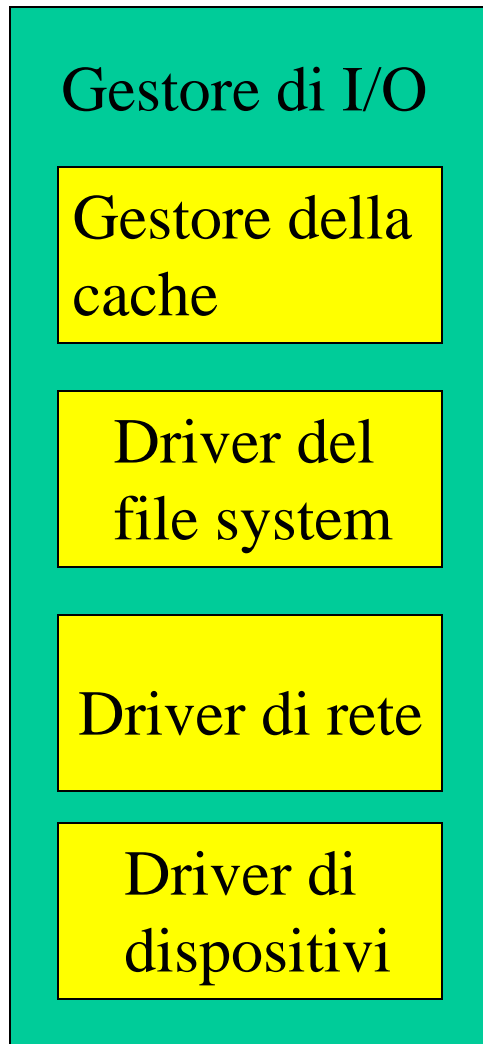


Segmento dati

- i -esima entry identifica un blocco di taglia $2^i \times 16$ KB
- frammentazione interna proporzionale alla taglia del processo



Architettura di I/O in sistemi Windows



- lettura pigra: gli aggiornamenti vanno su disco su base periodica
- sostituzione cache LRU
- swapping tramite un thread di sistema (su file)
- I/O asincrono:
 1. Segnalazione di oggetto
 2. Allertabile (APC – asynchronous procedure call)

I/O asincrono

- sfrutta il parametro di tipo *LPOVERLAPPED* nelle API di I/O_

lpOverlapped

[in] A pointer to an [OVERLAPPED](#) structure.

This structure is required if *hFile* is created with FILE_FLAG_OVERLAPPED.

If *hFile* is opened with FILE_FLAG_OVERLAPPED, the *lpOverlapped* parameter must not be NULL. It must point to a valid **OVERLAPPED** structure.

If *hFile* is created with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the function can report incorrectly that the read operation is complete.

The **OVERLAPPED** structure contains information used in asynchronous (or overlapped) input and output (I/O).

```
typedef struct _OVERLAPPED {
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union {
        struct {
            DWORD Offset;
            DWORD OffsetHigh;
        };
        PVOID Pointer;
    };
    HANDLE hEvent;
} OVERLAPPED,
*LPOVERLAPPED;
```

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)

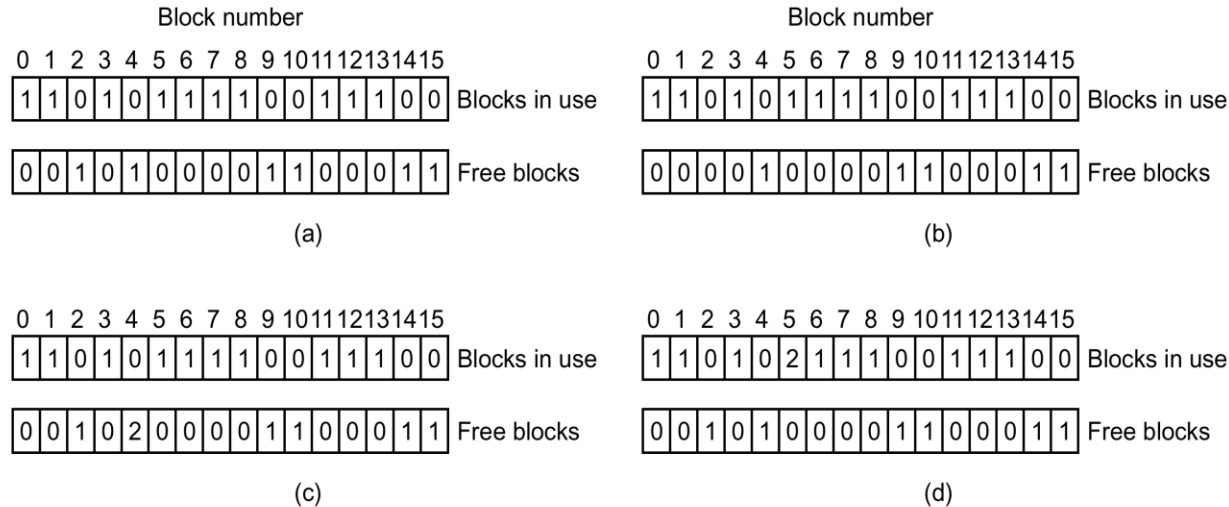
Semantica della sessione

- 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)

Consistenza del 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 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 blocco:
 - Blocchi in uso: a quanti e quali file appartengono (si spera uno)
 - Blocchi liberi: quante volte compaiono nella lista libera (si spera una o nessuna)

Ricostruzione del File System



a) Situazione consistente

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

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

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

Sincronizzazione del file system in Posix

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.