

Sistemi Operativi II

Corso di Laurea in Ingegneria Informatica

Facolta' di Ingegneria, Universita' "La Sapienza"

Docente: Francesco Quaglia

## **Gestione dei buffer e I/O scheduling:**

1. Richiami sulle tecniche di I/O
2. Gestione dei buffer
3. Schedulazione del disco
4. I/O in UNIX e Windows

# Tipologia dei dispositivi

## Leggibili dall'uomo

- terminali video
- tastiera mouse
- stampanti

## Leggibili dalla macchina

- 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

## 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

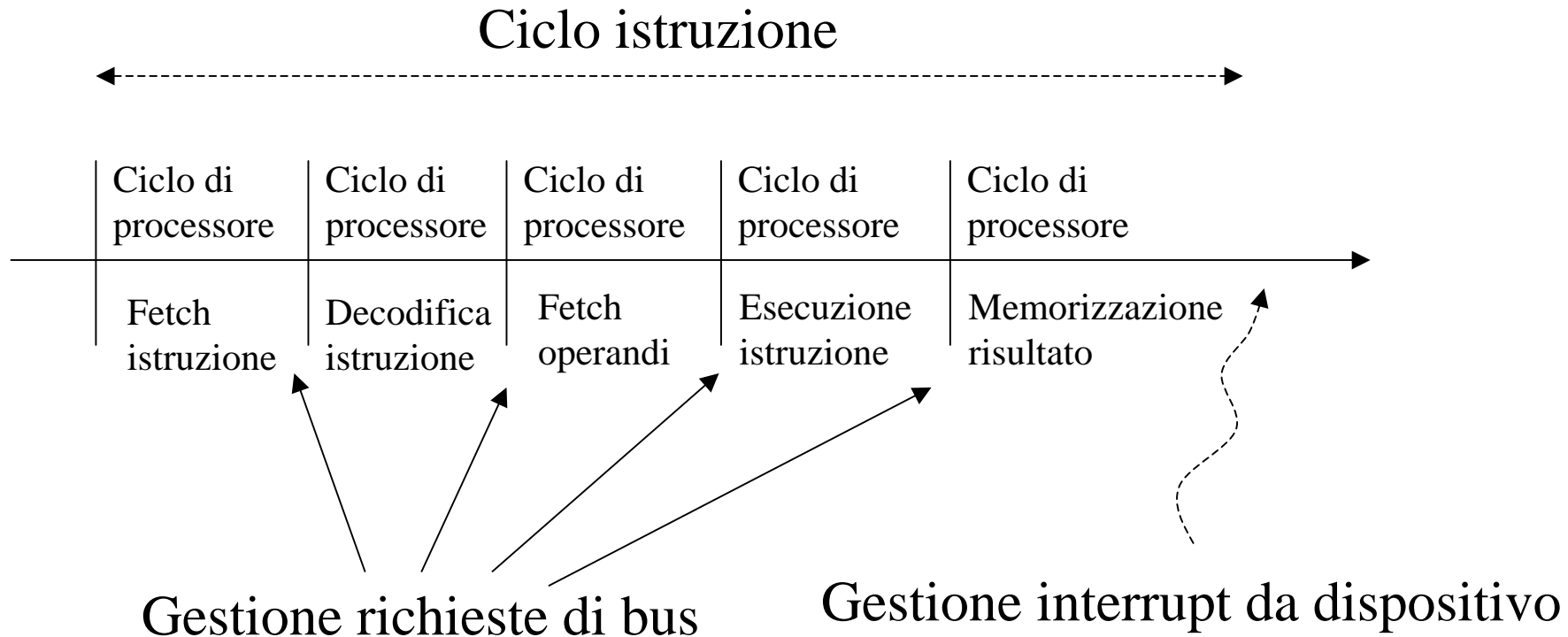
# Evoluzione storica

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

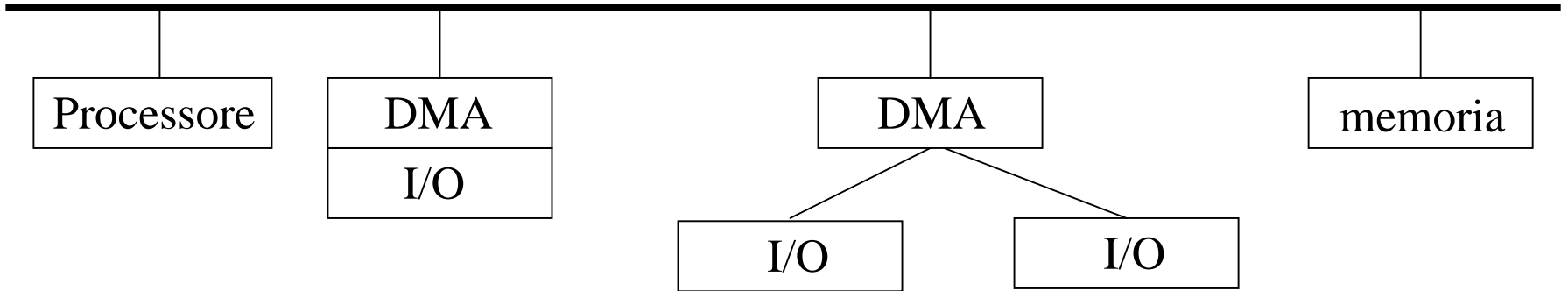
- le interruzioni relative alla richiesta di bus vengono gestite ad ogni ciclo di processore
- le interruzioni dei dispositivi (compreso il DMA) vengono gestite tra una istruzione macchina e l'altra



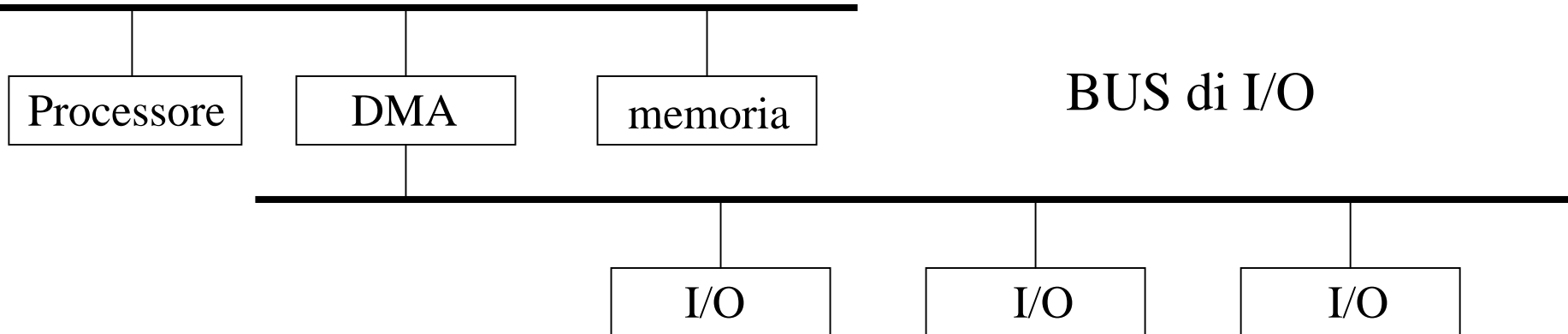
# Configurazioni



BUS singolo - DMA separato



BUS singolo - DMA e I/O integrati



BUS di I/O

# Progettazione della funzione di I/O: obiettivi

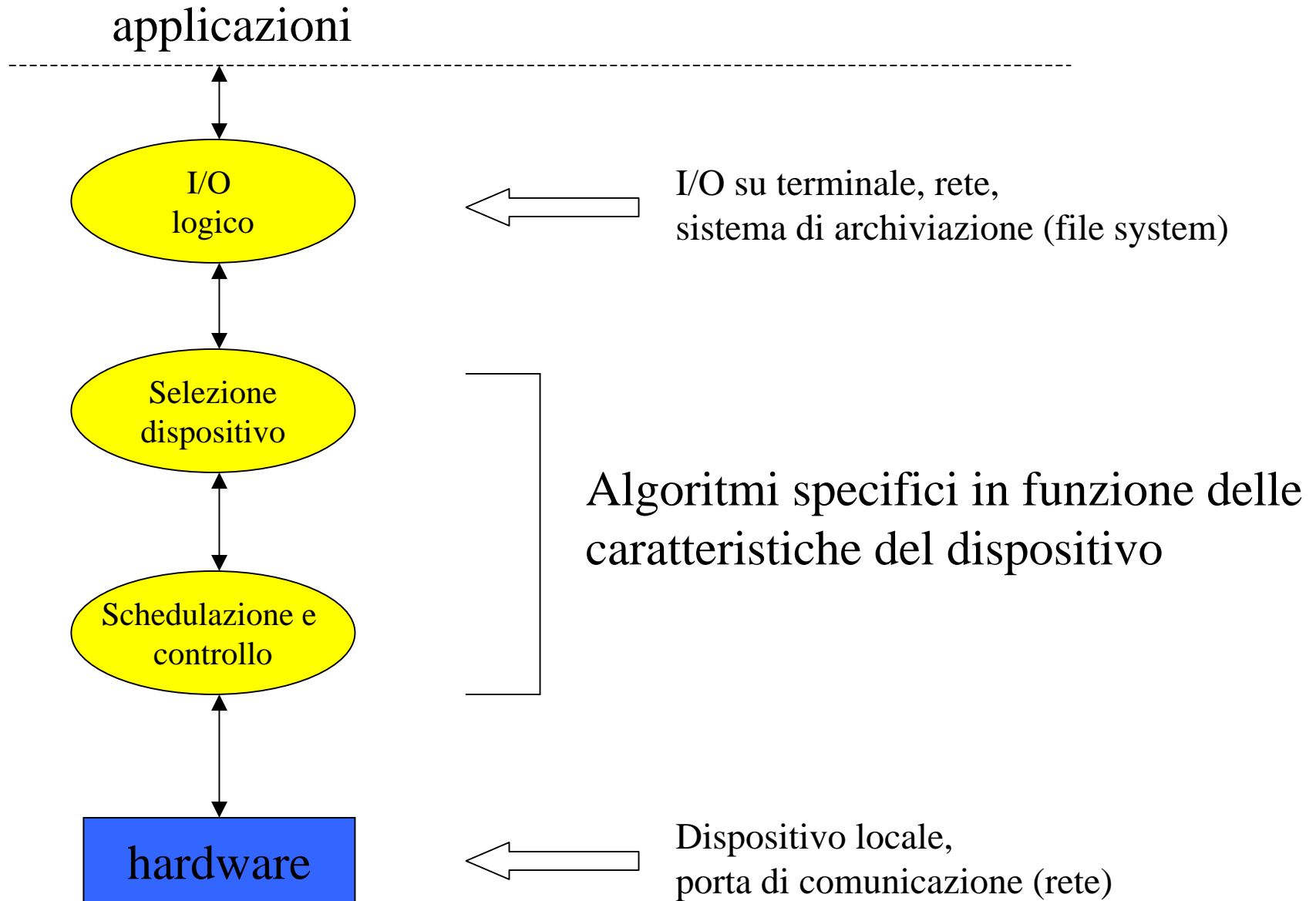
## Efficienza

- 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

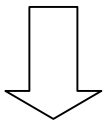
# Un modello di organizzazione



# 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)



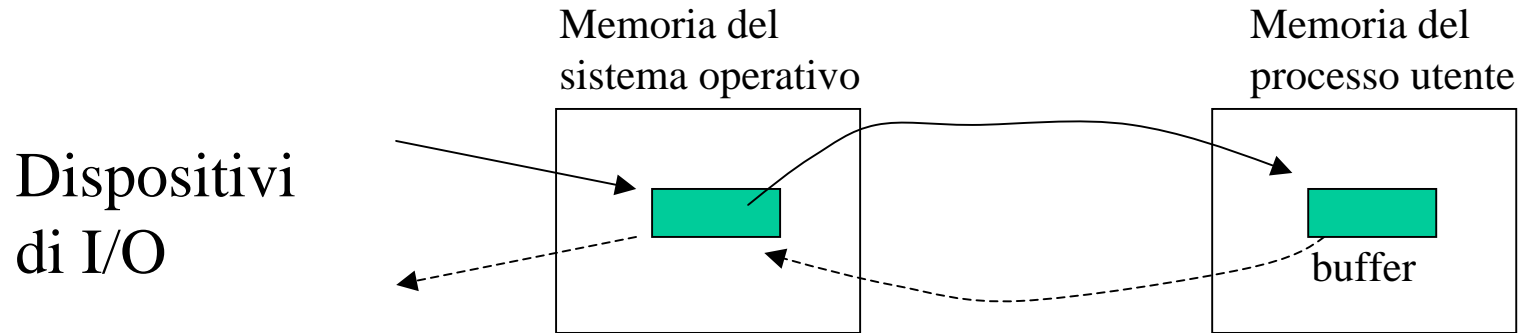
- l'I/O viene quindi tipicamente effettuato su aree di memoria riservate al sistema operativo
- questo introduce la necessità di gestire buffer destinati alle operazioni di I/O

---

## 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

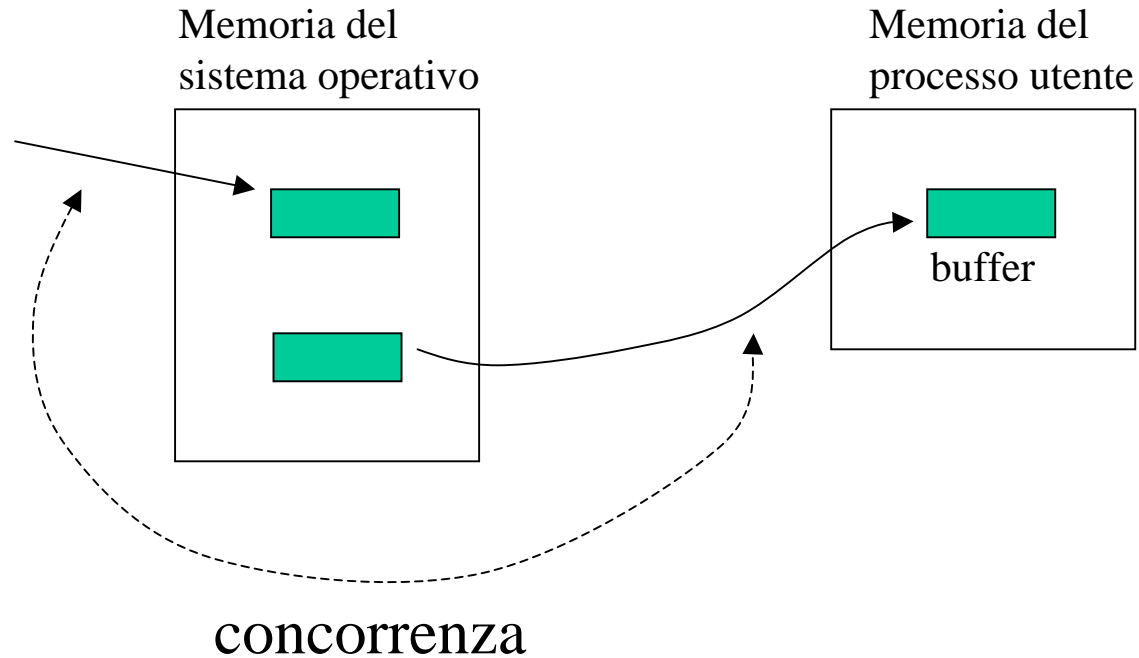
C = tempo tra due richieste di input

M = tempo per la copia del blocco nel buffer utente

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

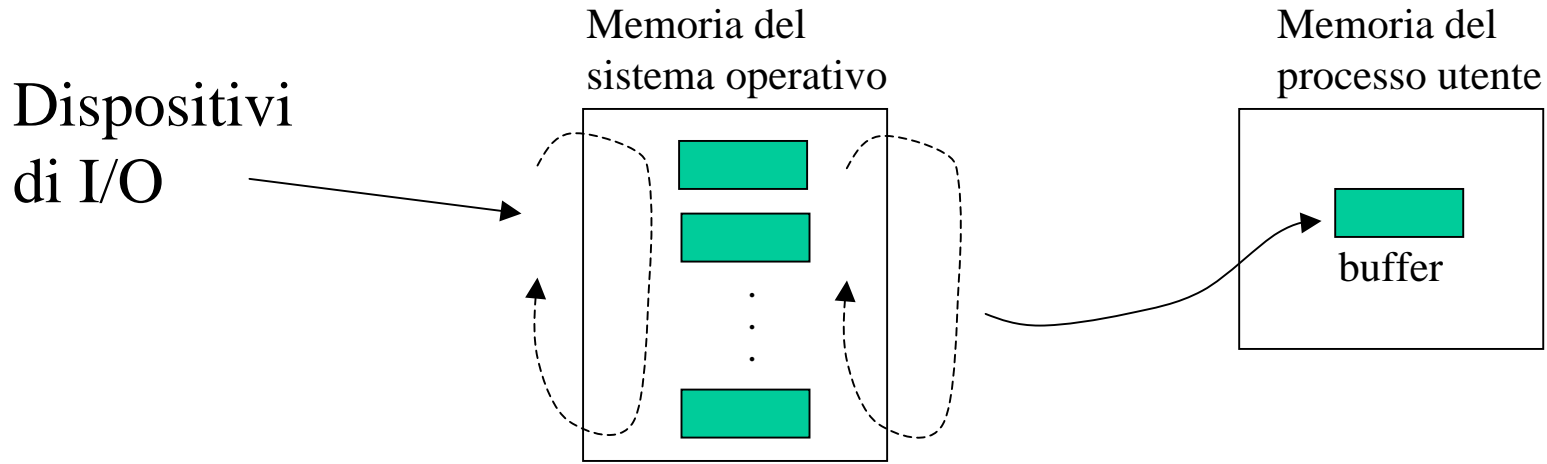
# Buffer doppio

Dispositivi  
di I/O



- 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

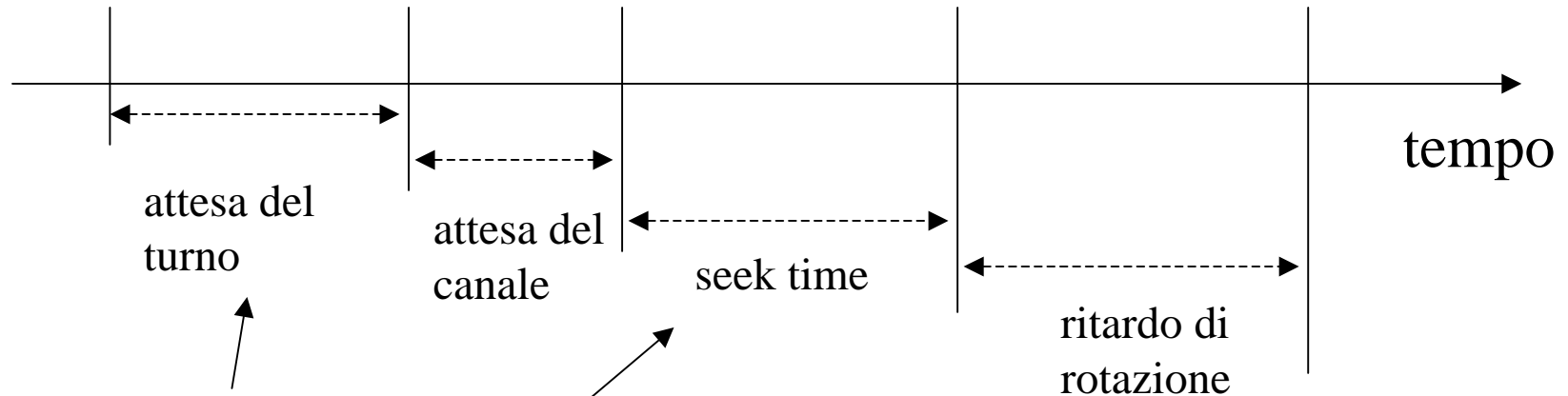
- nessuna quantità di buffer sarà mai sufficiente per evitare il blocco dei processi nel caso  $C < T$  ed i processi siano esclusivamente I/O bound
- in pratica i processi esibiscono comportamenti misti (I/O e CPU bound), quindi una quantità di buffer limitata favorisce l'uso efficiente di risorse e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

# Schedulazione del disco

## 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

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

# 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

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

$$\text{lunghezza media di ricerca} = \frac{\sum_{i=1}^{|\text{insieme dist}|} \text{dist}_i}{|\text{insieme dist}|} = 27.5$$

# 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

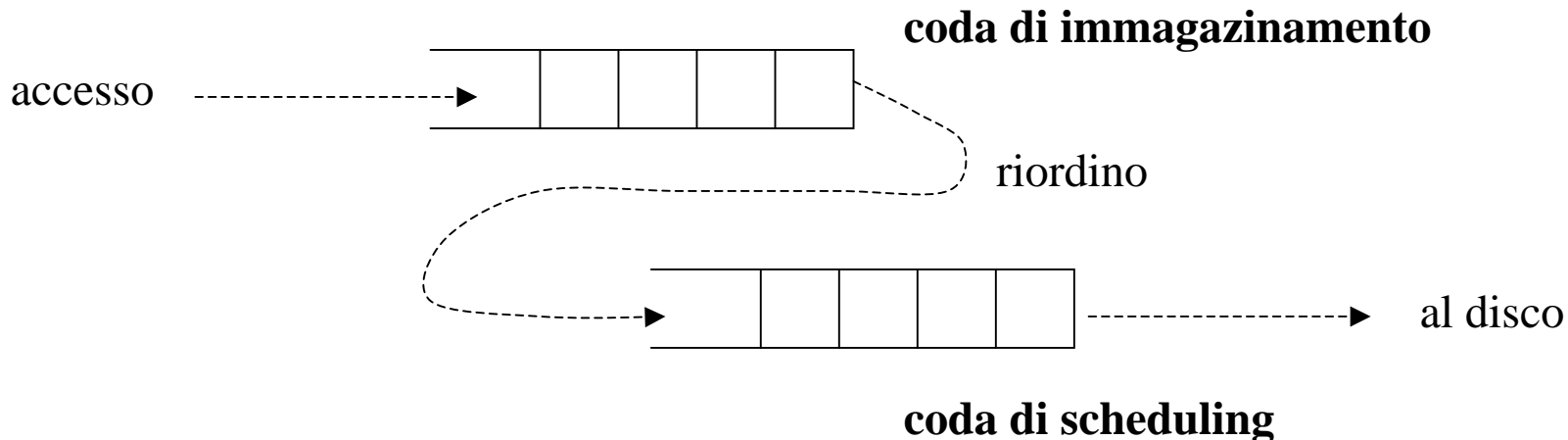
*|insieme dist|*

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

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

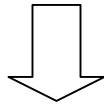


# Cache del disco

- 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

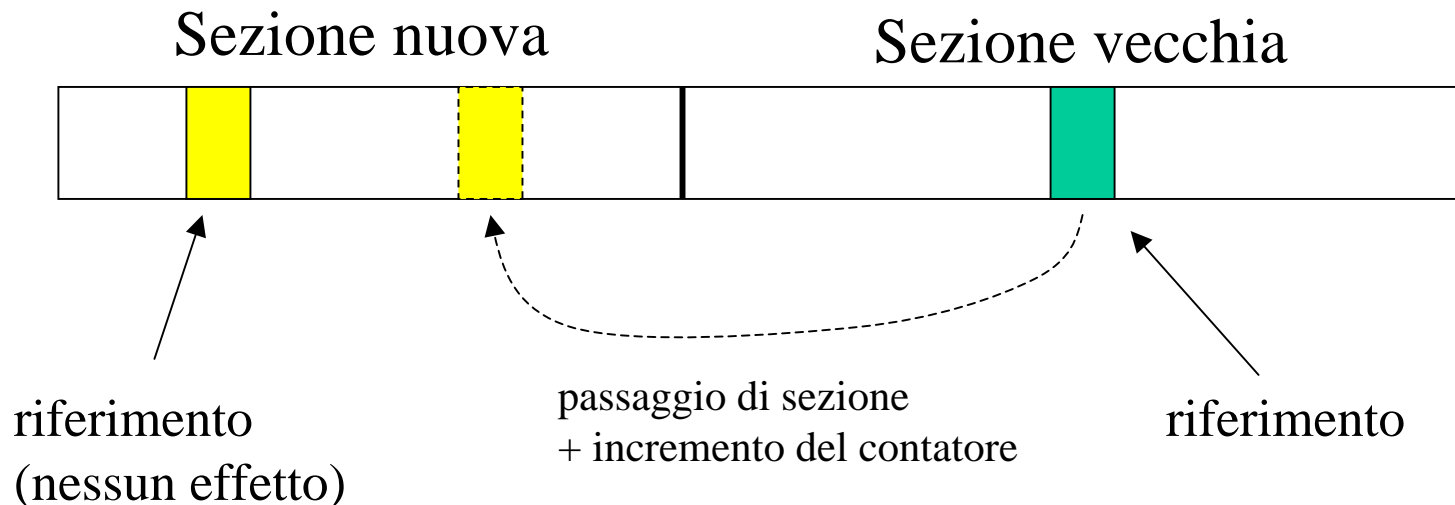
- Least-Recently-Used: viene mantenuta una lista di gestione a stack
- Least-Frequently-Used: si mantiene un contatore di riferimenti al blocco indicante il numero di accessi da quando è stato caricato



Problemi sulla variazione di località

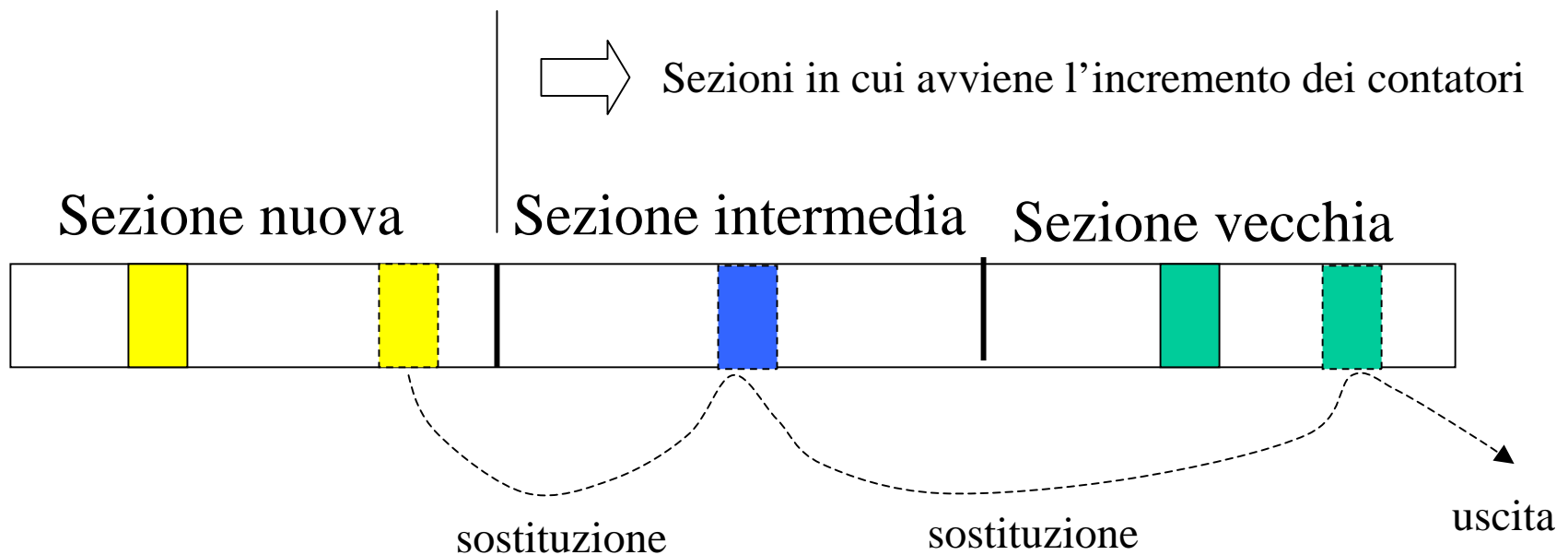
# Buffer cache a due sezioni

- 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

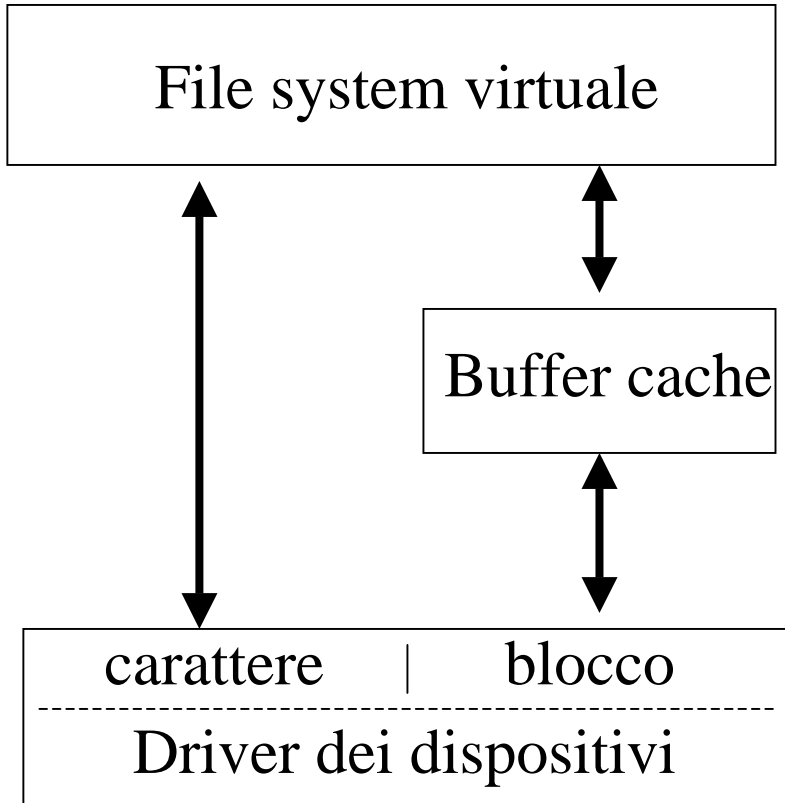
- file system

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

# I/O in 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

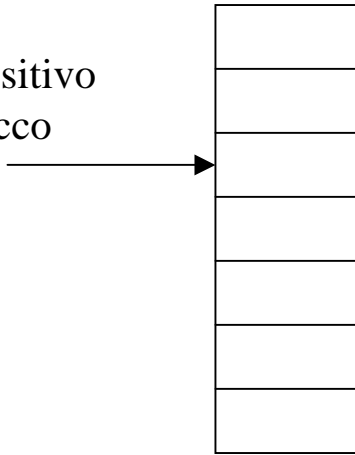
`fcntl` performs one of various miscellaneous operations on `fd`. The operation in question is determined by `cmd`.

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.

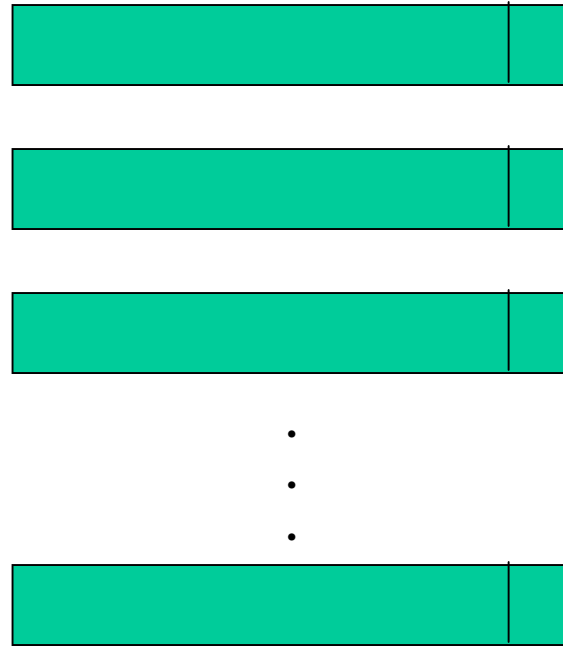
# Ricerca nel buffer cache

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

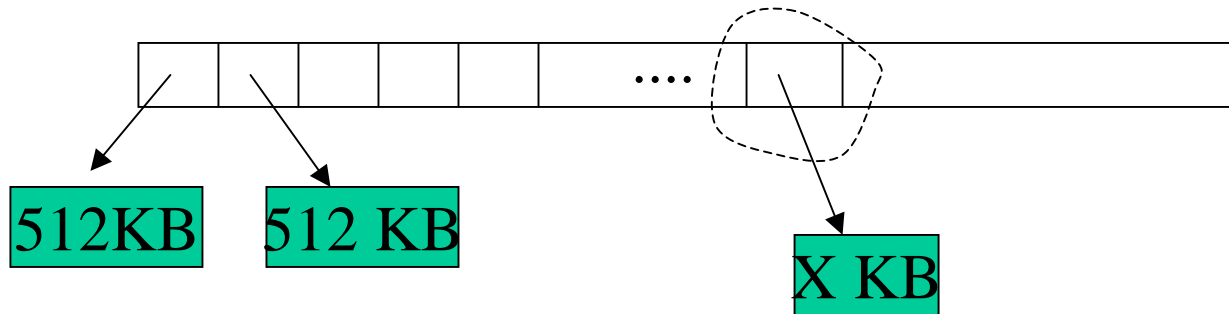
# Gestione della partizione di Swapping

- 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

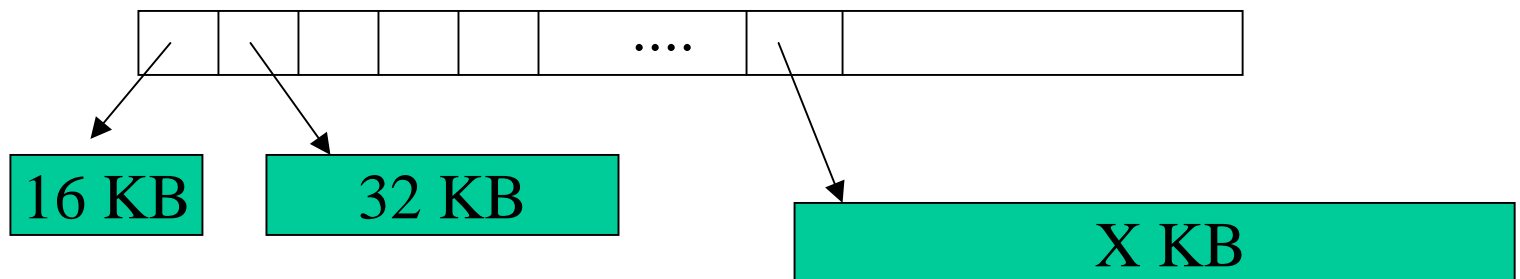
## Segmento testo

- granularità ai 512 KB, eccetto l'ultima entry

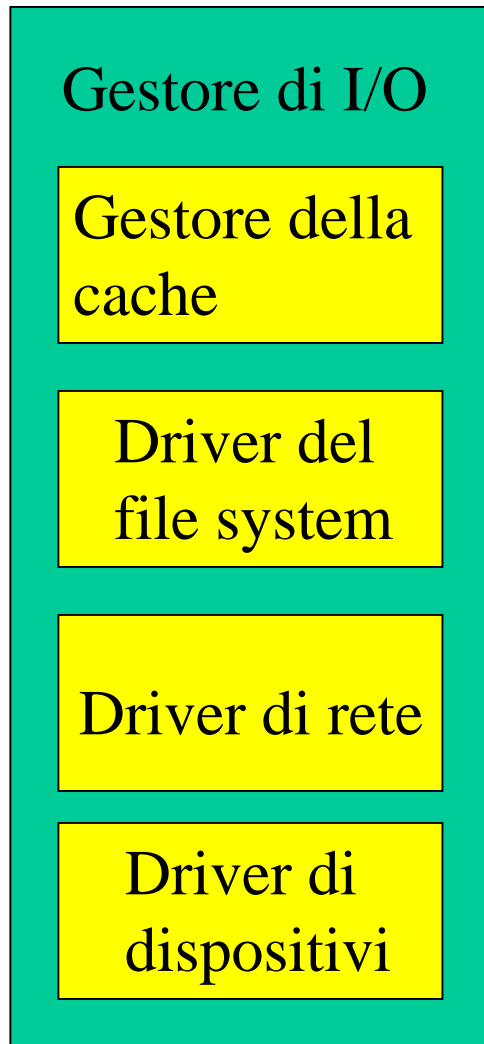


## Segmento dati

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



# I/O in Windows NT/2000



- lettura pigra: gli aggiornamenti vanno su disco su base periodica
- sostituzione cache LRU
- swapping tramite un thread di sistema
- 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)

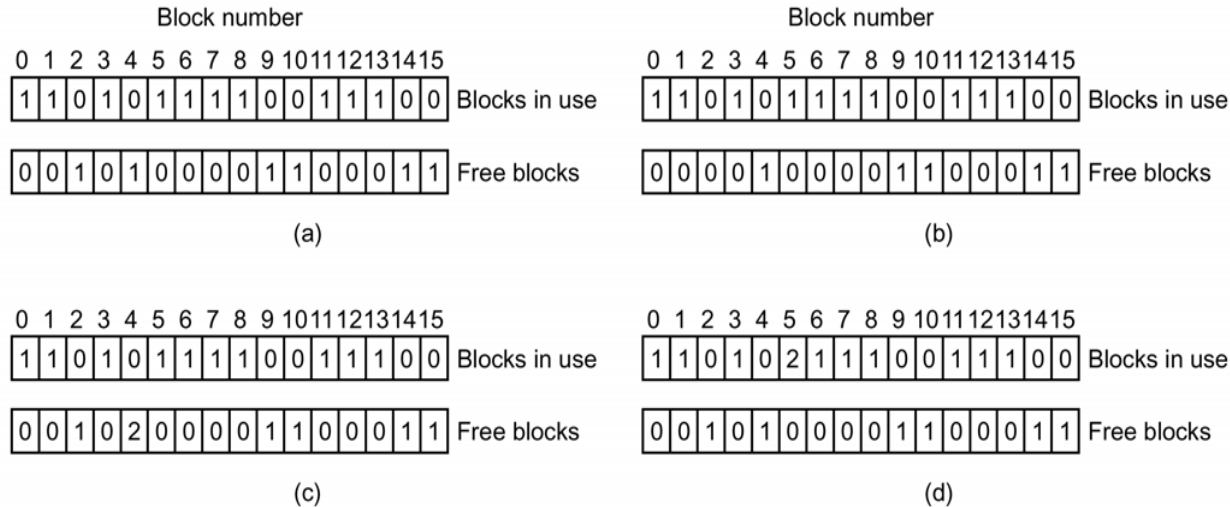
## Semantica della sessione

- 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 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.