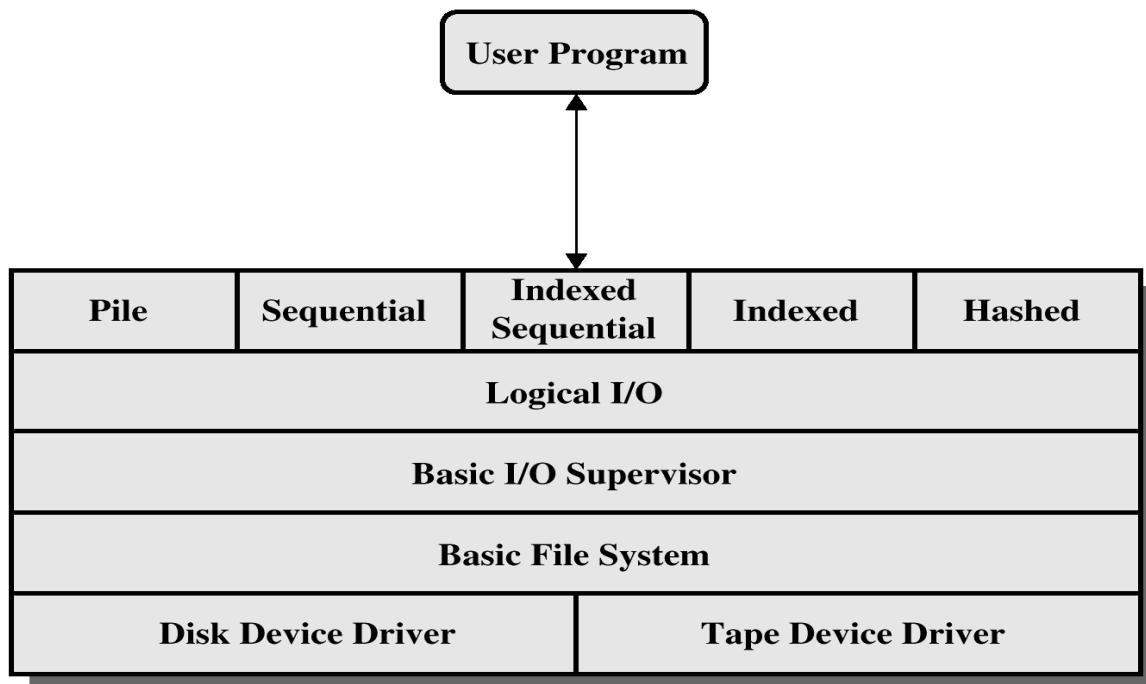

Parte V

Il File System

Il File System

- I/O Virtuale: l'accesso alla memoria di massa avviene tramite il SO
- La memoria di massa è organizzata in unità virtuali denominate **file** (archivio)
- File System: parte del SO che gestisce l'accesso ai file
- Per ciascun file il SO mantiene dati per:
 - identificazione
 - protezione
 - accesso fisico

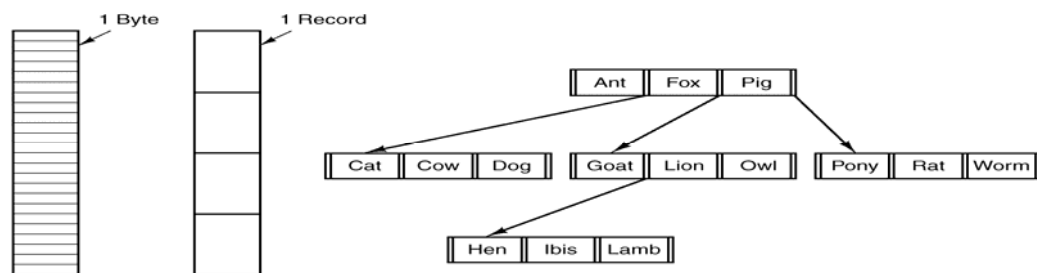
Architettura del File System



Architettura del File System (2)

- Device Drivers: gestiscono e mascherano tutte le caratteristiche a basso livello dei dispositivi
- Basic File System: indirizzamento fisico e buffering, a livello blocchi
- Basic I/O Supervisor: selezione del device, gestione dello stato, scheduling
- Logical I/O: accesso a livello della organizzazione logica (e.g. records)
- Metodi di accesso: permettono di accedere ai record del file con diverse modalità

Organizzazioni di file



- Organizzazione logica del file: vista dalle applicazioni
- A caratteri: sequenza indistinta di caratteri
- A record: sequenza di record di formato dato
- Collegata: insieme di blocchi tra loro collegati da una struttura di accesso

File Sequenziali

- L'accesso avviene in modalità puramente sequenziale
- È possibile solo leggere e scrivere il *record corrente*
- Per accedere ad un record diverso occorre spostarsi sul file passando su tutti i record intermedi
- Adatta solo ad elaborazioni puramente sequenziali
- Primitive tipiche offerte dal SO:
 - OPEN
 - CLOSE
 - READ
 - WRITE
 - REWIND

File ad accesso casuale

- Possibile accedere *direttamente* a qualsiasi record
 - tramite il *numero* del record
 - tramite una *chiave* contenuta nel record
- Parametri del problema
 - ***n*** : numero di record Es. 10^6
 - ***r*** : dimensione del record Es. 200 byte
 - ***B***: dimensione del blocco Es. 4k byte
 - ***c***: dimensione della chiave Es. 20 byte

Organizzazione del file

- I record sono organizzati in blocchi senza mai frazionare un record su due blocchi
- ***R*** = numero di record per blocco

$$R = \left\lfloor \frac{B}{r} \right\rfloor \qquad R = \left\lfloor \frac{4k}{200} \right\rfloor = 20$$

- ***F*** = numero di blocchi del file

$$F = \left\lceil \frac{n}{R} \right\rceil \qquad F = \left\lceil \frac{10^6}{20} \right\rceil = 50.000$$

Accesso sequenziale

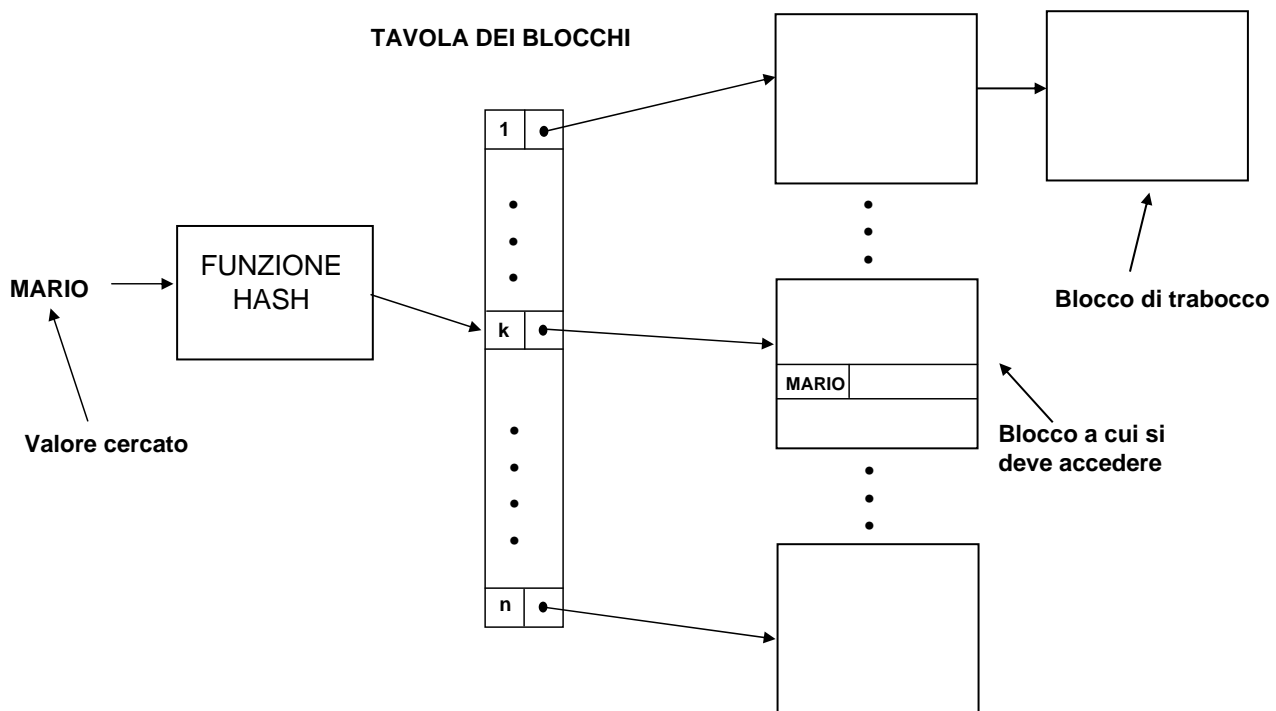
- Non presuppone nessuna particolare organizzazione del file
- Occorre effettuare una scansione sequenziale del file dati fino a trovare il record cercato
- Elevato costo di accesso
 - Operazione elementare: accesso ad un record di chiave data
 - Metrica di costo: numero di accessi a disco

$$C_{max} = F \quad C_{medio} = F/2$$

Accesso Hash

- File organizzato in base ad una struttura hash
- La funzione hash permette di associare a ciascuna chiave il blocco del file in cui il record avente quella chiave è contenuto
- La stessa funzione hash è utilizzata:
 - Per inserire ciascun record all'atto della costruzione del file
 - Per determinare a quale blocco occorre accedere quando si cerca un record di chiave data
- La funzione hash non è univoca, e pertanto si creano *liste di trabocco*

File Hash



Funzione Hash: esempio

- Chiave: stringa di 12 caratteri alfanumerici
- Ciascun carattere rappresentato da un byte
- 3 gruppi di 4 byte ciascuno trattati come interi a 32 bit
- I tre interi sono sommati, e del risultato si prende un gruppo di 10 bit (ad esempio dal bit 12 al bit 21)
- Questo gruppo è considerato come un numerale binario naturale
- Codominio della funzione: $[0,1023]$
- Cardinalità del codominio: 1024

Accesso Hash: costo

- Costo di accesso unitario nel caso ideale
- Nei casi reali occorre scandire la lista di trabocco
- Sulla lista occorre effettuare una *ricerca sequenziale*
- Il costo dipende dalla lunghezza media delle liste:
 - **S**: cardinalità del codominio della funzione hash
 - **L** : lunghezza media delle liste di trabocco
 - **F** : numero di blocchi complessivo del file

$$L = \frac{F}{S} \qquad C_{max} = \frac{F}{S} \qquad C_{medio} = \frac{F}{2S}$$

Accesso Hash: esempio

- Parametri del problema
 - **n** : numero di record 10⁶
 - **r** : dimensione del record 200 byte
 - **B**: dimensione del blocco 4k byte
 - **F**: dimensione del file dati 50.000 blocchi
 - **S**: cardinalità codominio 5.000
- Lunghezza media delle liste: $50.000/5.000 = 10$
- Costo medio di accesso: $50.000/2 \cdot 5.000 = 5$

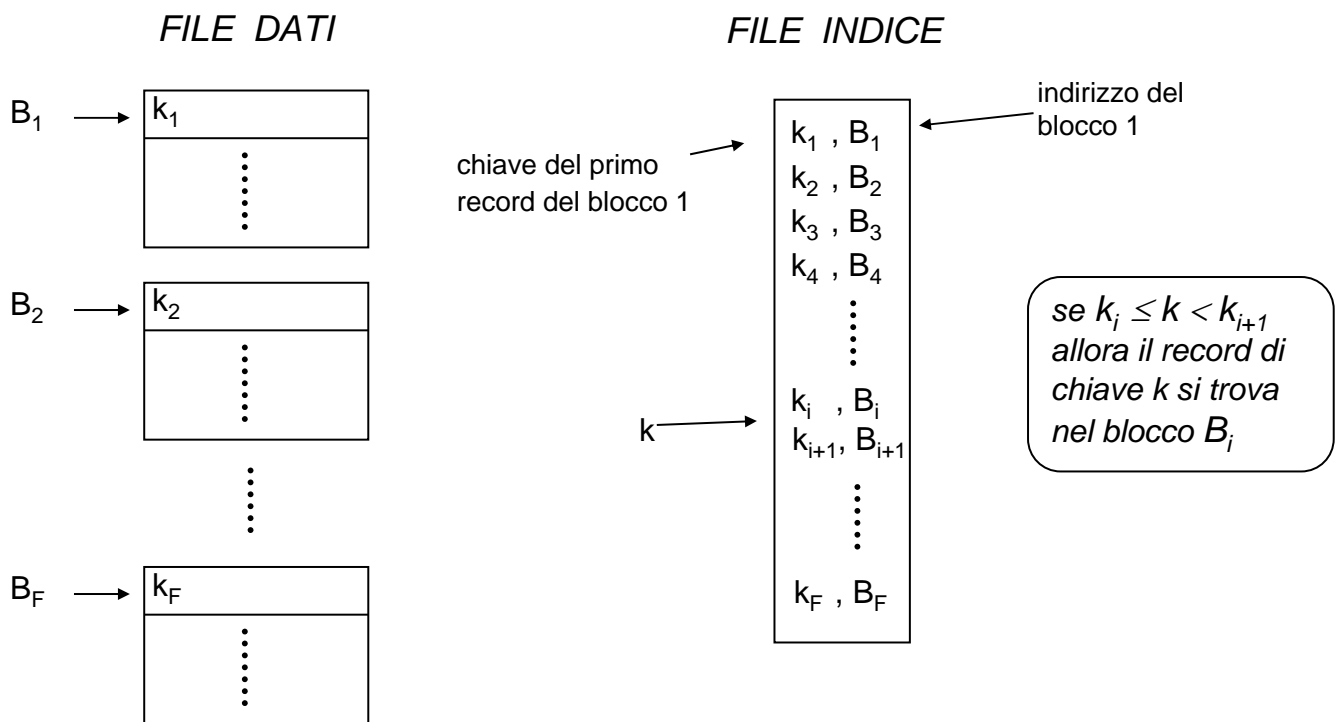
Accesso con indice

- File indice: struttura accessoria che viene affiancata al file dati per abbattere i costi di accesso
- Equivalente all'*indice analitico* di un libro:
 - Accesso senza indice: sfogliare il libro per cercare l'informazione desiderata
 - Accesso con indice: sfogliare l'indice e poi accedere direttamente alle pagine che interessano
- Possibile costruire diversi indici sullo stesso file dati, per supportare l'accesso su più campi del record

Indici ISAM

- *ISAM: Indexed Sequential Access Method*
- *File dati*: ordinato rispetto alla chiave
- *File indice*: ha un record per ogni blocco del file dati
- Ogni record del file indice contiene la prima chiave di un blocco del file dati e l'indirizzo del blocco
- Ricerca in due fasi di un record di chiave data:
 1. *Ricerca sequenziale sull'indice: individua l'indirizzo del blocco contenete il record*
 2. *Accesso diretto al blocco ed al record*

Indici ISAM: organizzazione



Indici ISAM: dimensioni dell'indice

- I parametri del problema:
 - F : dimensione in blocchi del file dati
 - c : dimensione in byte della chiave
 - b : dimensione in byte di un indirizzo su disco
 - B : dimensione in byte di un blocco su disco
- Il file indice ha F record uno per ogni blocco del file dati
- R_i : numero di record per blocco del file indice:
- I : dimensione in blocchi del file indice:

$$R_i = \left\lfloor \frac{B}{(c+b)} \right\rfloor \quad I = \left\lceil \frac{F}{R_i} \right\rceil$$

Indici ISAM: costo di accesso

- Accesso in due fasi:
 1. Scansione sequenziale del file indice
 2. Accesso diretto al blocco del file dati
- Costo di accesso:
 - C_{\min} : $1+1$
(un'accesso all'indice ed uno al file dati)
 - C_{medio} : $I/2+1$
(scansione di metà del file indice ed un accesso al file dati)
 - C_{\max} : $I+1$
(scansione di tutto il file indice ed un accesso al file dati)

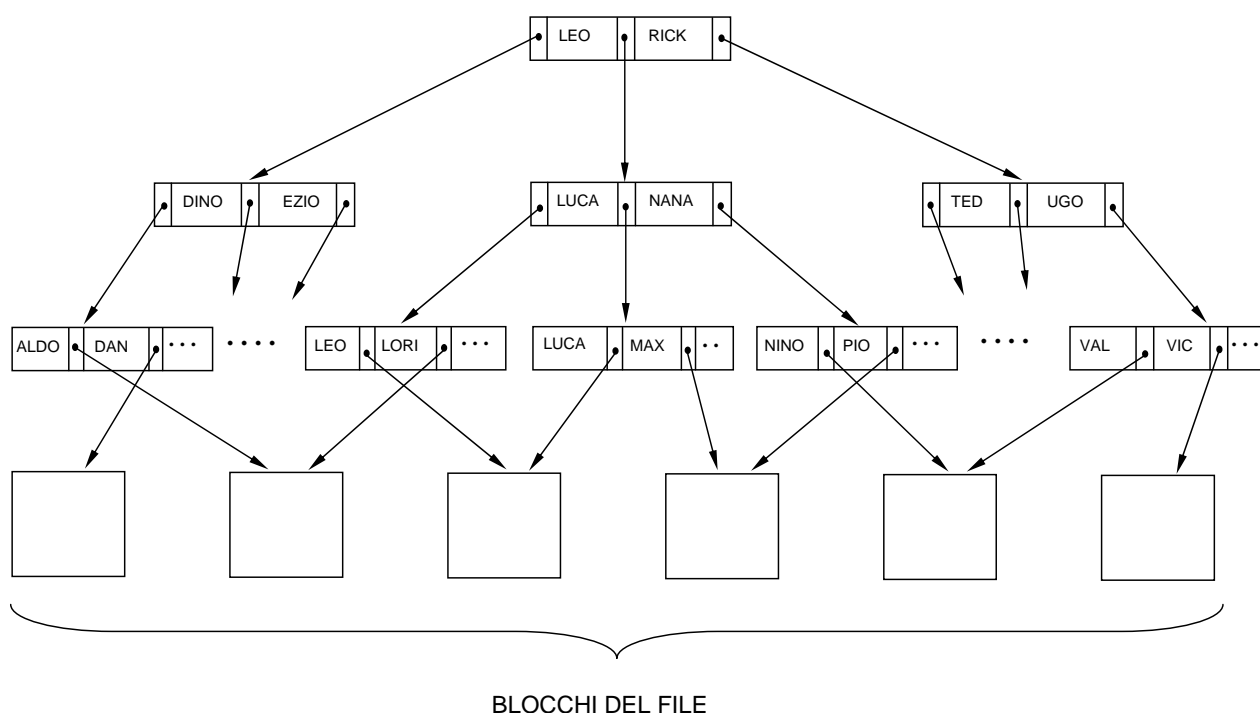
Indici ISAM: esempio

- Parametri del problema
 - n : numero di record 10^6
 - r : dimensione del record 200 byte
 - B : dimensione del blocco 4k byte
 - F : dimensione del file dati 50.000 blocchi
 - c : dimensione della chiave 16 byte
 - b : indirizzi su disco 4 byte
- Record per blocco del file indice:
$$R_i = \lfloor B/(c+b) \rfloor = \lfloor 4k/(16+4) \rfloor = 204:$$
- Dimensione del file indice:
$$I = \lceil F / R_i \rceil = \lceil 50.000 / 204 \rceil = 246$$
- Costo medio di accesso:
$$C_{\text{medio}} = I/2 + 1 = 246/2 + 1 = 124$$

Indici B-tree

- L'indice è costituito da un file strutturato ad albero
- Ciascun nodo dell'albero è un blocco su disco
- I nodi contengono liste di chiavi accoppiate a puntatori a sottoalberi
- Si scende sull'albero *ricorsivamente*:
 - Si confronta la chiave con la lista nella radice
 - Si determina il sottoalbero su cui proseguire
- Lunghezza di ricerca pari alla profondità dell'albero
- Profondità (e costo) logaritmici nel fattore di ramificazione
- Sono gli indici più diffusi

Indici B-tree



Gestione del disco

- Ciascun file è allocato sul disco come sequenza di blocchi non necessariamente contigui
- *Unità di allocazione* costituite da più blocchi
- Dimensione delle *unità di allocazione (cluster)* dipende dalla dimensione del disco e degli indirizzi su disco
- Gestione degli spazi liberi sul disco tramite :
 - Lista Libera: enumera tutte le sequenze di unità di allocazione libere *consecutive*
 - Bit Map: dedica un bit ad ogni unità di allocazione, per indicare se è libera o no

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)

(a) Lista Libera

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

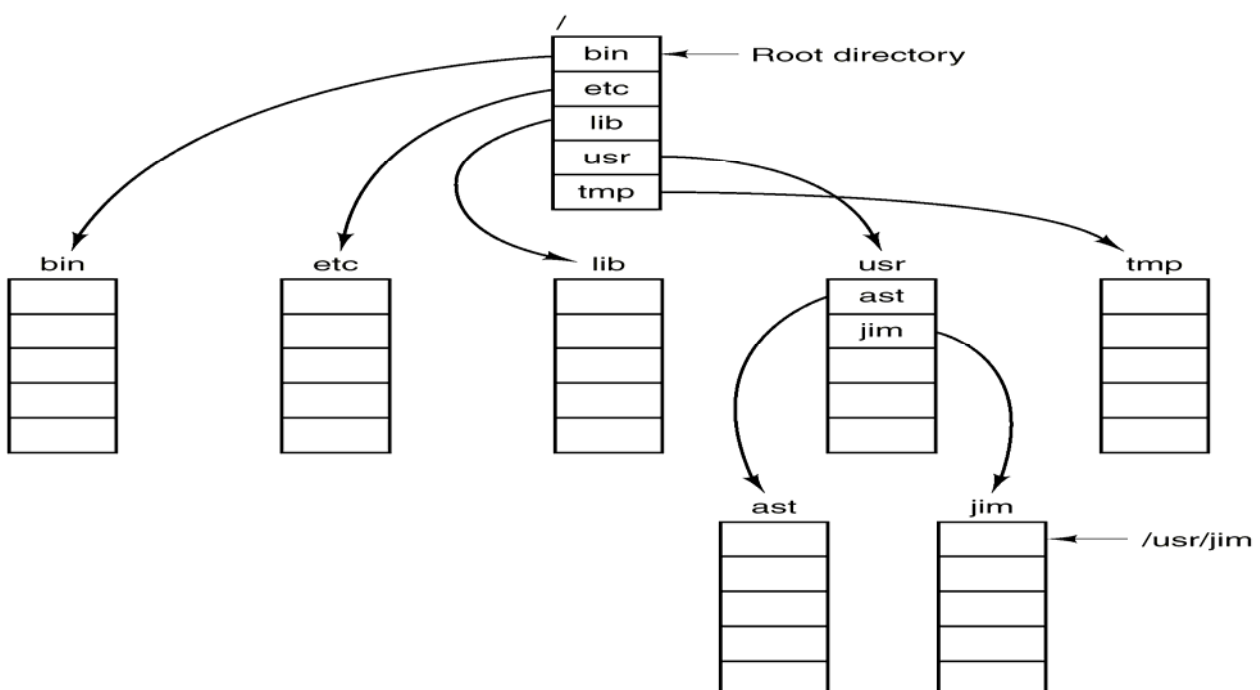
(b)

(b) Bit Map

File System di Unix

- I file sono sequenze di byte senza struttura
- Text file: sequenze di righe separate da \n
- Per ogni file aperto viene conservato il puntatore al prossimo byte
- File speciali associati ai dispositivi di I/O
- Trattamento uniforme di memoria di massa e dispositivi di I/O
- Struttura ad albero delle directory

Unix: struttura delle directory



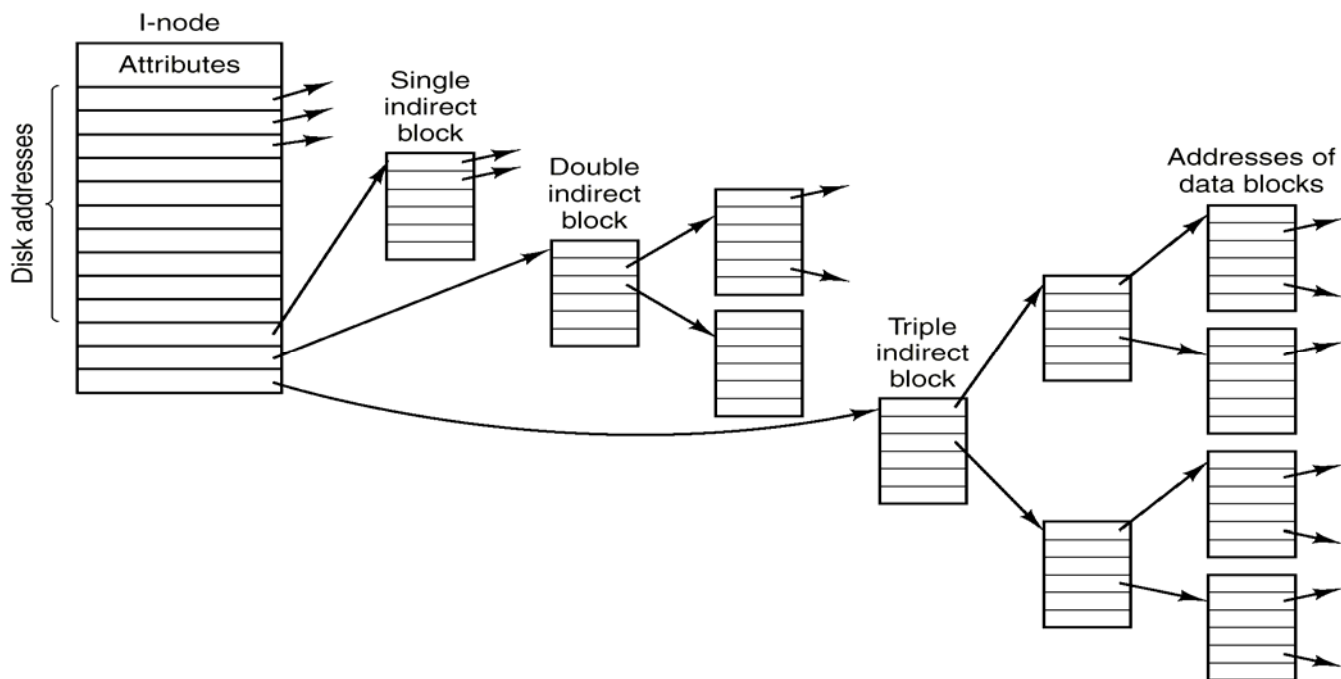
Unix: chiamate di sistema

- Chiamate di sistema:
 - `creat(name,mode)`; RWX RWX RWX
 - `open(name,mode)`;
 - `close(fd)`;
 - `read(fd,buffer,count)`;
 - `write(fd,buffer,count)`;
- La chiamata di apertura restituisce il descrittore `fd`
- La stringa di 9 bit RWX RWX RWX, fornita all'atto della creazione stabilisce la griglia dei diritti di accesso

i-node

- Le directory Unix associano ad ogni file il numero dello i-node ad esso corrispondente
- L'i-node raccoglie tutte le informazioni associate al file
 - Tipo di file
 - Identificatore dell'owner
 - Gruppo dell'owner
 - Griglia di accesso RWX RWX RWX
 - Istante dell'ultima lettura
 - 13 indirizzi di blocchi su disco che consentono l'accesso ai blocchi del file

i-node: struttura



i-node: indirizzamento dei blocchi

- Indirizzi 1-10: puntano direttamente a blocchi del file
- Indirizzo 11: punta ad un blocco che contiene indirizzi di blocchi del file: *indirezione semplice*
- Indirizzo 12: *doppia indirezione*
- Indirizzo 13: *tripla indirezione*
- Gestisce in modo efficiente i file piccoli penalizzando gradualmente l'accesso a file grandi
- La dimensione massima dei file dipende da quella dei blocchi e dei loro indirizzi

i-node: esempio

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

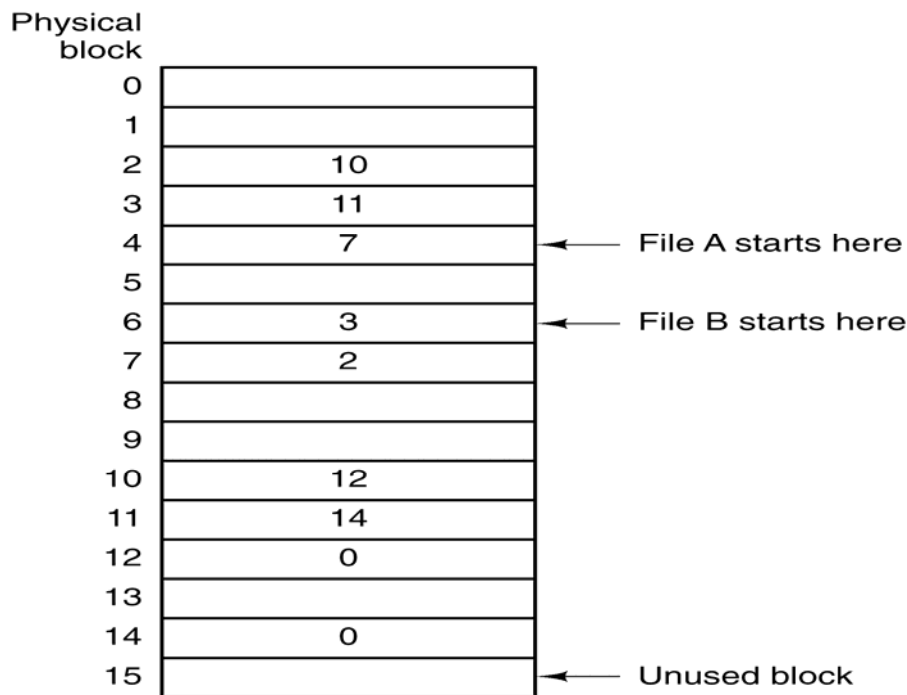
Maxfile = $(10+128 +128^2 +128^3) \cdot 512$ byte \approx 1Gbyte.

NB *La dimensione massima del file può essere anche limitata dal numero di byte dell'indirizzo di blocco*

FAT: File Allocation Table

- Struttura di accesso utilizzata da DOS e Windows
- La FAT mantiene la sequenza di blocchi allocati a ciascun file come lista
- Ad ogni elemento della FAT corrisponde un blocco (o un unità di allocazione)
- La directory punta all'elemento della FAT corrispondente al primo blocco del file
- Questo a sua volta punta a quello corrispondente al blocco successivo
- La fine del file è indicata da un valore particolare (e.g. 0)

FAT: File Allocation Table



FAT 16 e FAT 32

- MS/DOS prevedeva indirizzi su disco di 16 bit
- Un disco DOS contiene quindi una FAT di 64K elementi di 16 bit detta FAT 16 (ereditata poi da Windows 95)
- La FAT viene allocata all'inizio del disco (o della partizione)
- Con la FAT 16 un disco (o una partizione) non può avere più di 64k unità di allocazione
- Per formattare dischi di grandi dimensioni occorre aumentare le dimensioni delle unità di allocazione
- Questo peggiora la frammentazione interna
- Il problema è stato risolto con la FAT 32

FAT 16 e 32: esempio

- Disco da 32 MB = 2^{25} byte FAT 16
 - 2^{16} unità di allocazione (cluster)
 - Dimensione minima dei cluster: $2^{25} / 2^{16} = 2^9 = 512$ byte
- Disco di 2 GB = 2^{31} byte FAT 16
 - 2^{16} unità di allocazione (cluster)
 - Dimensione minima dei cluster: $2^{31} / 2^{16} = 2^{15} = 32k$ byte
- Disco di 32 GB = 2^{35} byte FAT 32
 - 2^{32} unità di allocazione (cluster)
 - Dimensione minima dei cluster: $2^{35} / 2^{32} = 2^3 = 8$ byte
 - Non ha senso comunque avere cluster inferiori a 512 byte

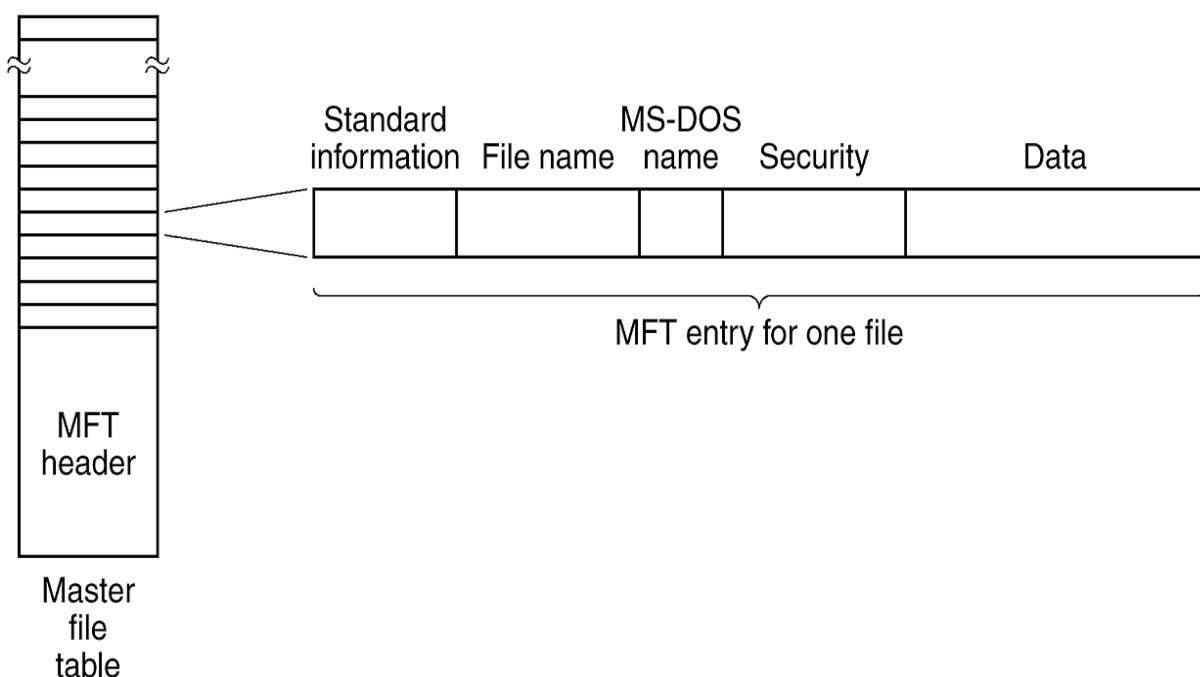
NTFS (NT File System)

- Windows NT permette di utilizzare la vecchia FAT 16, ma ha anche un suo file system nativo NTFS
- Dischi divisi in volumi (partizioni), e organizzati in *cluster* da 512 a 64K byte
- In luogo della FAT ogni volume viene gestito tramite una **MFT (Master File Table)**
- Ad ogni file corrisponde un elemento nella MFT
- Elementi della MFT di 1K o della dimensione del cluster

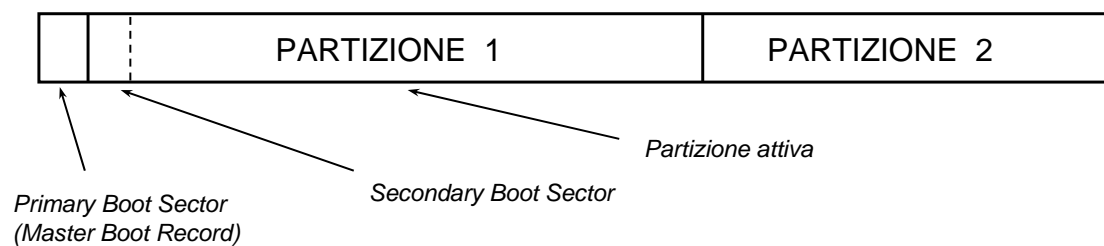
La Master File Table (MFT)

- 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
- 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

Struttura della MFT

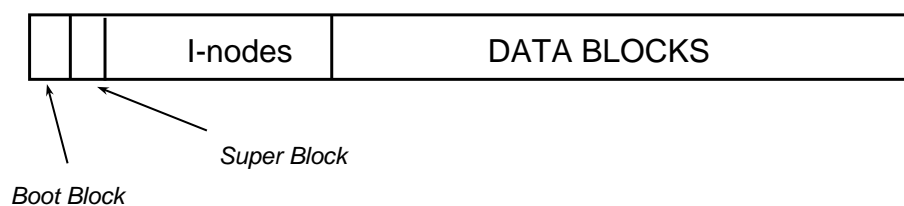


Partizionamento di dischi



- Sullo stesso disco possono essere costruite più partizioni
- Ciascuna partizione viene formattata con un suo file system
- Il settore 0 del disco (*primary boot sector*) contiene la tavola delle partizioni ed il *Master Boot Record*, che individuano la partizione attiva
- Il primo settore della partizione attiva (*secondary boot sector*) contiene il codice di boot del SO
- Le altre partizioni vengono montate come ulteriori file system

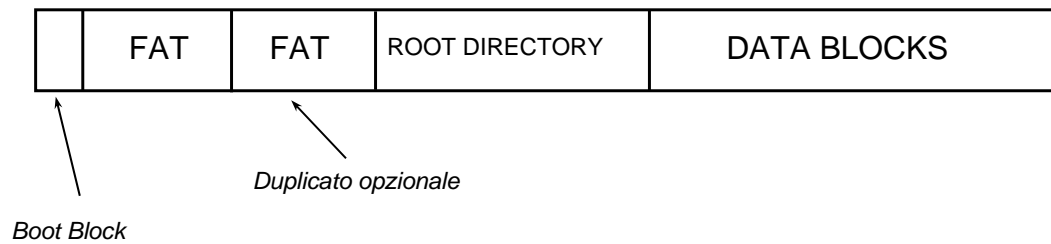
Partizione UNIX



- Boot Block: secondary boot sector
- Super Block: informazione di gestione, n. di i-nodes, n. di blocchi, inizio della free list ecc.
- I-nodes: creati in numero fisso nella formattazione, posizioni fisse; se i file sono tanti possono non bastare
- Data blocks: da allocare ai file

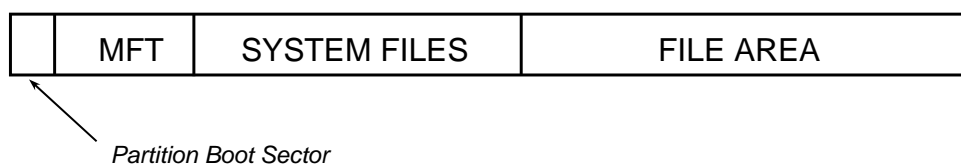
NB Numero di i-node fisso per la root directory, (e.g. 2)

Partizione FAT



- Boot Block: secondary boot sector
- FAT: File Allocation Table
- FAT duplicata: copia ridondante per motivi di affidabilità
- Root Directory: in posizione fissa, permette di accedere alla struttura gerarchica del file system
- Data blocks: da allocare ai file

Partizione NTFS (Volume)



- Partition Boot Sector: codice di boot e informazioni sul volume
- Master File Table (MFT): contiene un elemento per ogni file (vedi prima)
- System Files:
 - MFT2: duplicato delle prime righe di MFT
 - Bit map: indica i blocchi liberi
 - Log file: per il recovery
- File Area: blocchi da allocare ai file

Frammentazione dei file

- L'allocazione contigua dei blocchi di un file favorisce le prestazioni
- Risparmio in tempi di *seek* e di *latency*
- Un vincolo rigido di allocazione contigua non è perseguibile e causa sprechi di spazio
- Frammentazione di un file: misura il numero di segmenti non contigui in cui il file è ripartito
- La frammentazione ha un effetto negativo sulle prestazioni
- Possibile *deframmentare* un file tramite le apposite utility

Consistenza del File System

- Crash di sistema possono portare il file system in uno stato *inconsistente*
- Possibile ricostruire il file system con apposite utility:
 - *fsck* in UNIX
 - *scandisk* in Windows
- Vengono analizzate le strutture del file system (FAT, 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

Block number															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1
(a)															
Block number															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	1
(b)															
Block number															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1
(c)															
Block number															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1
(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: toglì un'occorrenza*

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

La gestione dei dischi

- Una gestione efficiente dei dischi ha un forte impatto sulle prestazioni
- Accessi concorrenti allo stesso disco:
 - in un contesto multiprogrammato
 - in un contesto di rete (*disk server*)
- Si creano *code di richieste* per ciascun disco
- Problema di scheduling
- L'ottimizzazione della gestione degli accessi:
 - diminuisce l'utilizzazione del disco
 - migliora il tempo di risposta
 - migliora i tempi di risposta dei processi

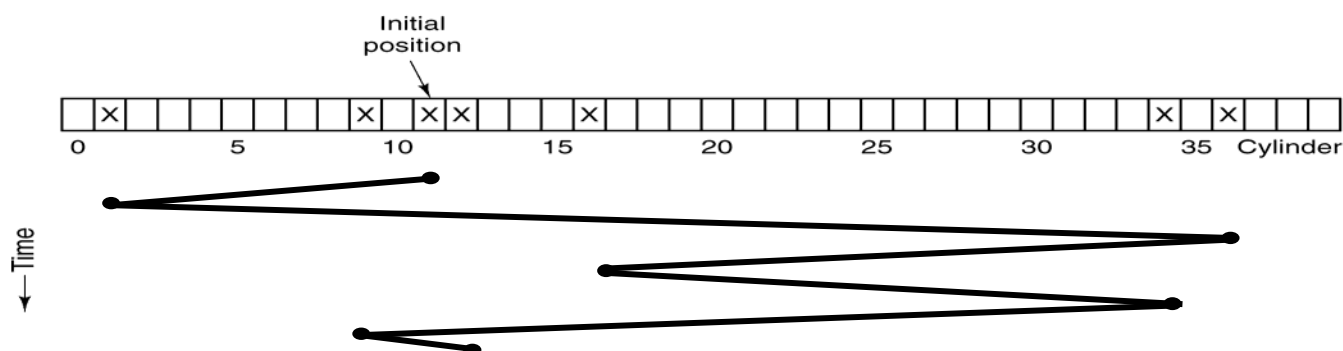
Disk scheduling

- Gestione efficiente di una coda di richieste al disco
- Di ciascuna richiesta si conosce:
 - Cilindro
 - Settore
- Possibile riordinare le richieste
- Controller intelligenti possono gestire direttamente lo scheduling delle richieste

Obiettivo: *minimizzazione del tempo complessivo di seek e di latency*

NB L'ottimizzazione del tempo medio di servizio, può causare un aumento della varianza

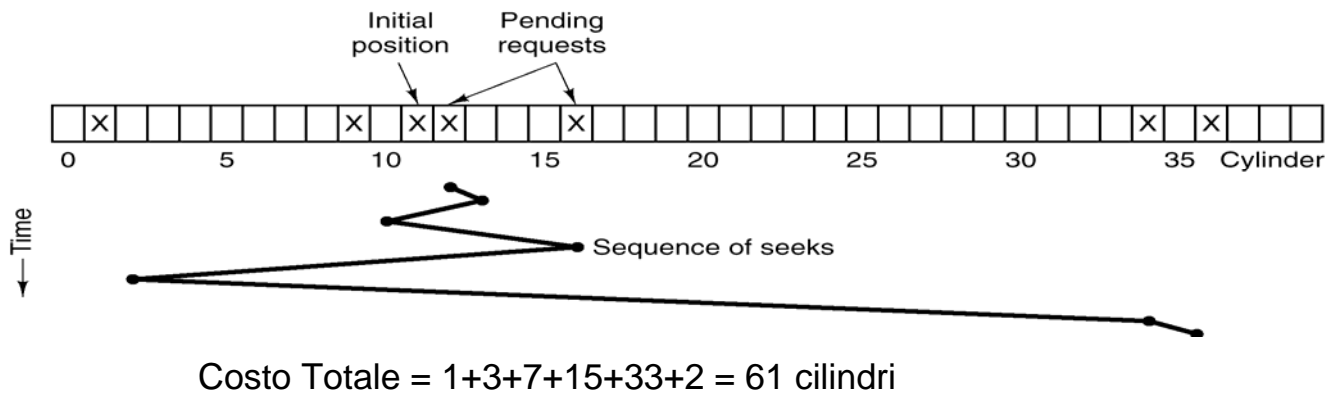
Scheduling FCFS



Costo Totale = $10+35+20+18+25+3 = 111$ cilindri

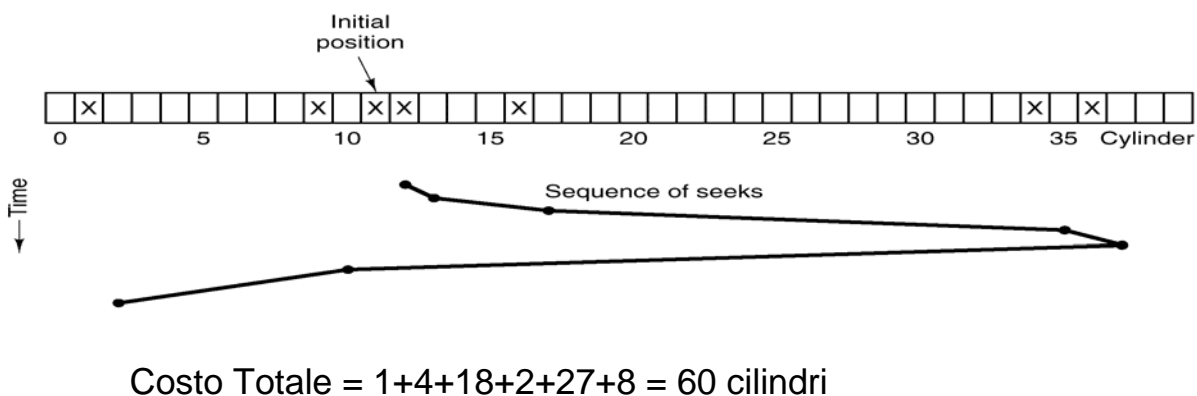
- FCFS: *First Come First Served*
- Le richieste vengono servite in base all'ordine di arrivo
- Nessuna ottimizzazione

Scheduling SSF



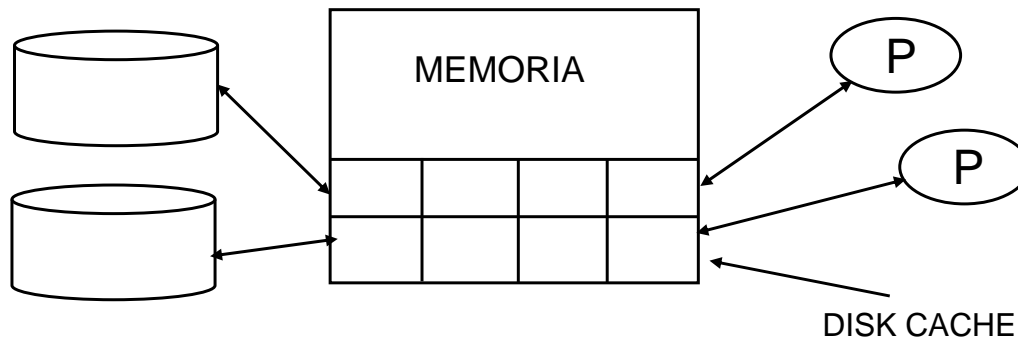
- SSF: *Shortest Seek First*
- Si serve la richiesta che si trova sul cilindro più vicino a quello corrente
- Minimizza sempre il tempo medio di seek
- Può causare un peggioramento della varianza
- Una richiesta può attendere indefinitamente

Algoritmo dell'ascensore



- Analogo a quello degli ascensori nei grattacieli
- Come SSF ma guardando le sole richieste nella direzione in cui la testina si sposta
- Quando non esistono ulteriori richieste in quella direzione il moto si inverte
- Pone un massimo all'attesa: due volte il numero di cilindri

Disk Cache



- Copia dei blocchi recentemente acceduti è conservata in memoria in una speciale area: la *disk cache* (o *buffer cache*)
- I processi trovano spesso il blocco nella buffer cache (*cache hit*), risparmiando operazioni di I/O
- L'accesso si fa a disco solo in caso di *cache miss*
- Molto efficiente grazie alla località

Disk Cache: gestione

- Strategie di rimpiazzamento
 - LRU: costosa da implementare
 - LFU (*Least Frequently Used*) più semplice da implementare, altrettanto efficace
- Gestione delle scritture
 - Write through: ogni scrittura in cache è effettuata anche sul disco
 - Write back: solo quando il blocco esce dalla cache le scritture sono riportate su disco

NB *Con write back cache e disco sono disallineati anche per lunghi periodi*

UNIX Buffer Cache

- Gestisce tutto l'I/O dei dispositivi a blocchi
- Divisa in *slot* (o *buffer*) di dimensioni pari ad un blocco
- Trasferimenti memoria-memoria tra buffer cache e process I/O area tramite DMA
- Gestione della cache tramite tavola di accesso hash
- Algoritmo di rimpiazzamento LRU
- Il SO tende ad estendere la dimensione della buffer cache compatibilmente con le esigenze di paginazione
- Politica di scrittura *write back*, con allineamenti periodici di tutta la buffer cache (sync)

Cache Manager di Windows NT

- Varia dinamicamente le dimensioni della cache e la sua frazione dedicata a ciascuna attività
- Gestione dilazionata delle scritture, per migliorare le prestazioni:
 - **Lazy write**: scritture solo in cache; sono trasferite su disco quando la richiesta di CPU è bassa
 - **Lazy commit**: permette di ritardare il *commit* di transazioni che hanno scritture che sono state effettuate solo in cache (supporto necessario per DBMS e sistemi transazionali)

I/O Sincrono ed Asincrono

- I/O Sincrono

| *Il processo (thread) che richiede una operazione di I/O
| va in blocco fina a che l'operazione è completata*

- I/O Asincrono

| *Il thread continua la sua elaborazione concorrentemente
| all'operazione di I/O bloccandosi solo se necessario*

- L'I/O asincrono permette di aumentare la concorrenza e quindi le prestazioni
- Diversi meccanismi offerti per segnalare al thread l'avvenuta completamento della operazione di I/O