

# Implementazione del File System

Ivan Scagnetto

Università di Udine — Facoltà di Scienze MM.FF.NN.

A.A. 2007-2008

Copyright ©2000–04 Marino Miculan (miculan@dimi.uniud.it)

La copia letterale e la distribuzione di questa presentazione nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta.

# Implementazione del File System

I dispositivi tipici per realizzare file system: **dischi**

- trasferimento **a blocchi** (tip. 512 byte, ma variabile)
- accesso **diretto** a tutta la superficie, sia in lettura che in scrittura
- dimensione **finita**

# Struttura dei file system

**programmi di applicazioni:** applicativi, ma anche comandi (**ls**, **dir**, ...)

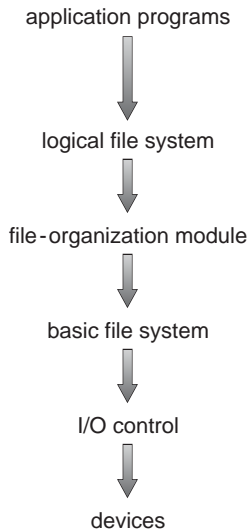
**file system logico:** presenta i diversi file system come un'unica struttura; implementa i controlli di protezione

**organizzazione dei file:** controlla l'allocazione dei blocchi fisici e la loro corrispondenza con quelli logici. Effettua la traduzione da indirizzi logici a fisici.

**file system di base:** usa i driver per accedere ai blocchi fisici sull'appropriato dispositivo.

**controllo dell'I/O:** i driver dei dispositivi

**dispositivi:** i controller hardware dei dischi, nastri, ecc.



# Tabella dei file aperti

- Per accedere ad un file è necessario conoscere informazioni riguardo la sua posizione, protezione, . . .
- questi dati sono accessibili attraverso le directory
- per evitare continui accessi al disco, si mantiene in memoria una **tabella dei file aperti**. Ogni elemento descrive un file aperto (**file control block**)
  - Alla prima open, si caricano in memoria i metadati relativi al file aperto
  - Ogni operazione viene effettuata riferendosi al file control block in memoria
  - Quando il file viene chiuso da tutti i processi che vi accedevano, le informazioni vengono copiate su disco e il blocco deallocato
- Problemi di affidabilità (e.g., se manca la corrente. . .)

# Mounting dei file system

- Ogni file system fisico, prima di essere utilizzabile, deve essere **montato** nel file system logico
- Il montaggio può avvenire
  - al boot, secondo regole implicite o configurabili
  - dinamicamente: supporti rimovibili, remoti, ...
- Il punto di montaggio può essere
  - fissato (A:, C:, ... sotto Windows, sulla scrivania sotto MacOS)
  - configurabile in qualsiasi punto del file system logico (Unix)
- Il kernel esamina il file system fisico per riconoscerne la struttura ed il tipo
- Prima di spegnere o rimuovere il media, il file system deve essere **smontato** (pena gravi inconsistenze!)

# Allocazione contigua

Ogni file occupa un insieme di blocchi contigui sul disco

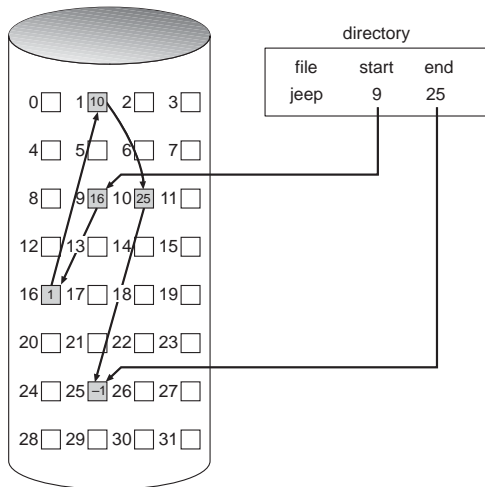
- Semplice: basta conoscere il blocco iniziale e la lunghezza
- L'accesso random è facile da implementare
- Frammentazione esterna. Problema di allocazione dinamica.
- I file non possono crescere (a meno di deframmentazione)
- Frammentazione interna se i file devono allocare tutto lo spazio che gli può servire a priori
- Traduzione dall'indirizzo logico a quello fisico (per blocchi da 512 byte):

$$LA/512 \begin{cases} Q \\ R \end{cases}$$

- Il blocco da accedere =  $Q$  + blocco di partenza
- Offset all'interno del blocco =  $R$

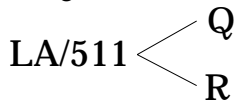
# Allocazione concatenata

Ogni file è una **linked list** di blocchi, che possono essere sparpagliati ovunque sul disco



# Allocazione concatenata

- Allocazione su richiesta; i blocchi vengono semplicemente collegati alla fine del file
- Semplice: basta sapere l'indirizzo del primo blocco
- Non c'è frammentazione esterna
- Bisogna gestire i blocchi liberi
- Non supporta l'accesso diretto (seek)
- Traduzione indirizzo logico:

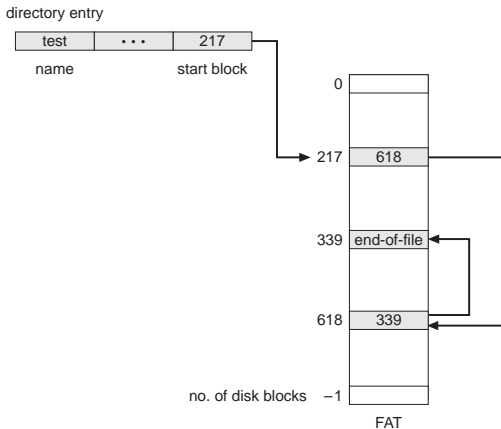


- Il blocco da accedere è il Q-esimo della lista
- Offset nel blocco =  $R + 1$



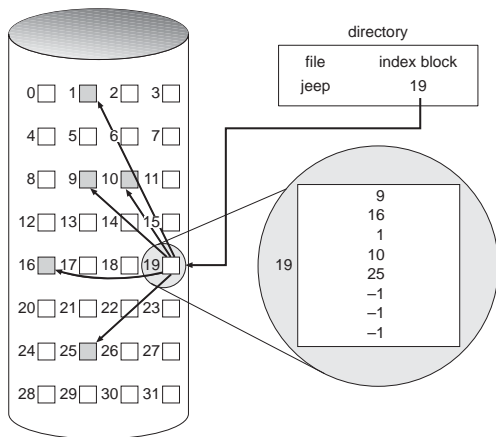
# Allocazione concatenata

Variante: *File-allocation table (FAT)* di MS-DOS e Windows.  
Mantiene la linked list in una struttura dedicata, all'inizio di ogni partizione



# Allocazione indicizzata

Si mantengono tutti i puntatori ai blocchi di un file in una **tabella indice**.



# Allocazione indicizzata

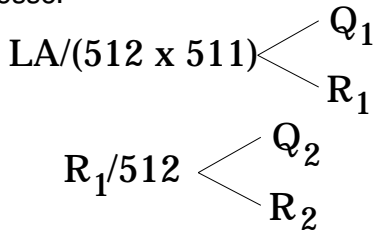
- Supporta accesso random
- Allocazione dinamica senza frammentazione esterna
- Traduzione: file di max 256K word e blocchi di 512 word:  
serve 1 blocco per l'indice

$$LA/512 \begin{cases} Q \\ R \end{cases}$$

- $Q$  = offset nell'indice
- $R$  = offset nel blocco indicato dall'indice

# Allocazione indicizzata

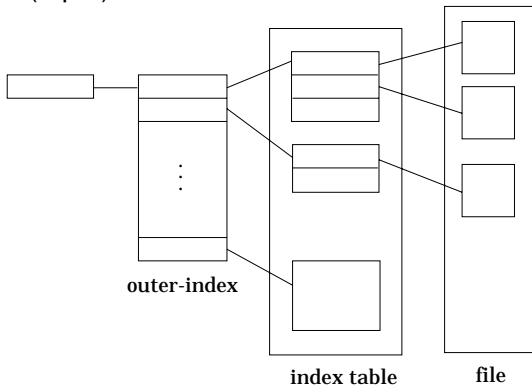
- Problema: come implementare il blocco indice
  - è una struttura supplementare: overhead  $\Rightarrow$  meglio piccolo
  - dobbiamo supportare anche file di grandi dimensioni  $\Rightarrow$  meglio grande
- Indice concatenato: l'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza, maggiore costo di accesso.



- $Q_1$  = blocco dell'indice da accedere
- $Q_2$  = offset all'interno del blocco dell'indice
- $R_2$  = offset all'interno del blocco del file

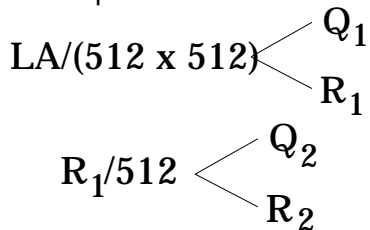
# Allocazione indicizzata

Indice a due (o più) livelli.



Dim. massima: con blocchi da 4K, si arriva a  $(4096/4)^2 = 2^{20}$   
blocchi =  $2^{32}B = 4GB$ .

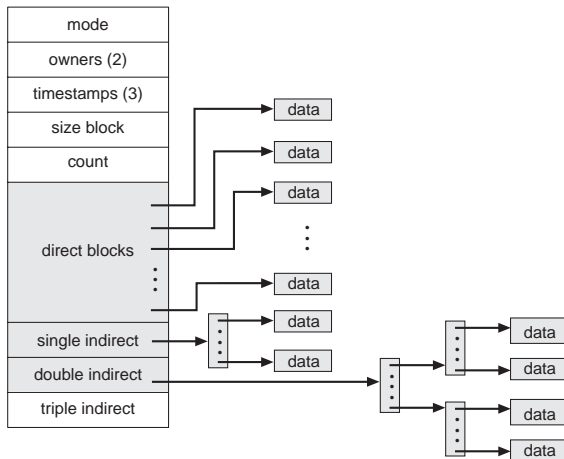
- Con blocchi da 512 parole:



- $Q_1$  = offset nell'indice esterno
- $Q_2$  = offset nel blocco della tabella indice
- $R_2$  = offset nel blocco del file

# Unix: Inodes

Un file in Unix è rappresentato da un **inode** (nodo indice), che sono allocati in numero finito alla creazione del file system



Ogni inode contiene

**Modo:** bit di accesso, di tipo e speciali del file

**UID e GID** del possessore

**Dimensione** del file in byte

**Timestamp** di ultimo accesso (**atime**), di ultima modifica (**mtime**), di ultimo cambiamento dell'inode (**ctime**)

**Numero di link** hard che puntano a questo inode

**Blocchi diretti:** puntatori ai primi 12 blocchi del file

**Primo indiretto:** indirizzo del blocco indice dei primi indiretti

**Secondo indiretto:** indirizzo del blocco indice dei secondi indiretti

**Terzo indiretto:** indirizzo del blocco indice dei terzi indiretti (mai usato!)



- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte

$$\begin{aligned}L_{max} &= 12 + 1024 + 1024^2 + 1024^3 \\ &> 1024^3 = 2^{30} \text{ blk} \\ &= 2^{42} \text{ byte} = 4 \text{ TB}\end{aligned}$$

molto oltre le capacità dei sistemi a 32 bit.

I blocchi non utilizzati sono indicati da una **lista di blocchi liberi**  
— che spesso lista non è

- Vettore di bit (**block map**): 1 bit per ogni blocco

0101110101010111111011000000101000000010110101011

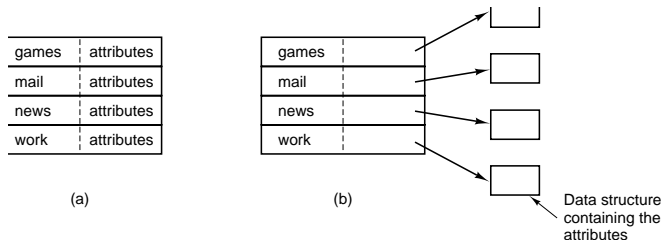
$$\text{bit}[i] = \begin{cases} 0 & \Rightarrow \text{block}[i] \text{ occupato} \\ 1 & \Rightarrow \text{block}[i] \text{ libero} \end{cases}$$

- Comodo per operazioni assembler di manipolazione dei bit
- Calcolo del numero del blocco  
(numero di bit per parola) \*  
(numero di parole di valore 0) +  
offset del primo bit a 1

- La bit map consuma spazio. Esempio:
  - block size =  $2^{12}$  bytes
  - disk size =  $2^{35}$  bytes (32 gigabyte)
  - $n = 2^{35}/2^{12} = 2^{23}$  bits =  $2^{20}$  byte = 1M byte
- Facile trovare blocchi liberi contigui
- Alternativa: **Linked list (free list)**
  - Inefficiente - non facile trovare blocchi liberi contigui
  - Non c'è spreco di spazio.

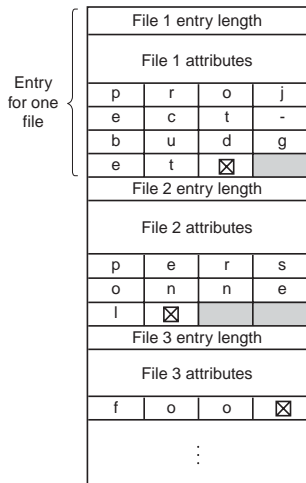
# Implementazione delle directory

Le directory sono essenziali per passare dal nome del file ai suoi attributi (anche necessari per reperire i blocchi dei dati).

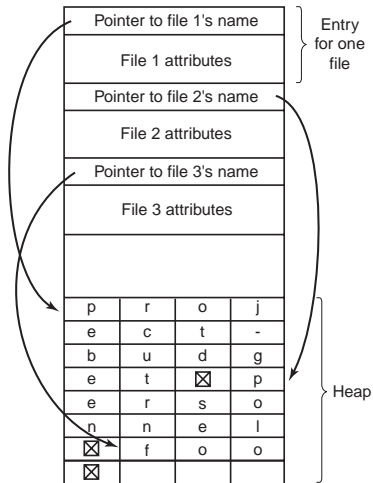


- a) Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows)
- b) Gli attributi risiedono in strutture esterne (eg. inodes), e nelle directory ci sono solo i puntatori a tali strutture (UNIX)

# Directory: a dimensioni fisse, o a heap



(a)



(b)

# Directory: liste, hash, B-tree

- Lista lineare di file names con puntatori ai blocchi dati
  - semplice da implementare
  - lenta nella ricerca, inserimento e cancellazione di file
  - può essere migliorata mettendo le directory in cache in memoria
- Tabella hash: lista lineare con una struttura hash per l'accesso veloce
  - si entra nella hash con il nome del file
  - abbassa i tempi di accesso
  - bisogna gestire le *collisioni*: ad es., ogni entry è una lista
- B-tree: albero binario bilanciato
  - ricerca binaria
  - abbassa i tempi di accesso
  - bisogna mantenere il bilanciamento

Dipende da

- algoritmi di allocazione spazio disco e gestione directory
- tipo di dati contenuti nelle directory
- grandezza dei blocchi
  - blocchi piccoli per aumentare l'efficienza (meno frammentazione interna)
  - blocchi grandi per aumentare le performance
  - e bisogna tenere conto anche della paginazione!

# Sulla dimensione dei blocchi

Consideriamo questo caso: un disco con

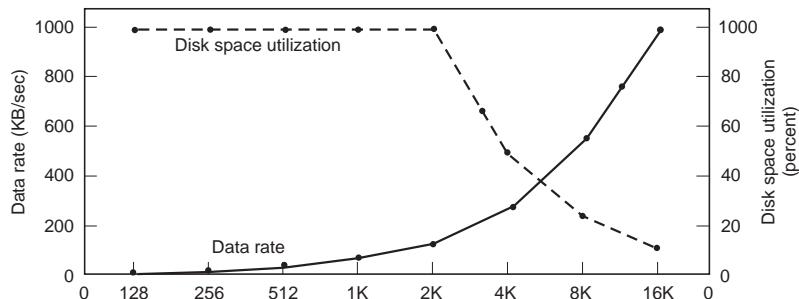
- 131072 byte per traccia
- tempo di rotazione = 8.33 msec
- seek time = 10 msec

Il tempo per leggere un blocco di  $k$  byte è allora

$$10 + 4.165 + (k/131072) * 8.33$$



# Sulla dimensione dei blocchi



La mediana della lunghezza di un file Unix è circa 2KB.

Tipiche misure: 1K-4K (Linux, Unix); sotto Windows il cluster size spesso è imposto dalla FAT (anche se l'accesso ai file è assai più complicato).

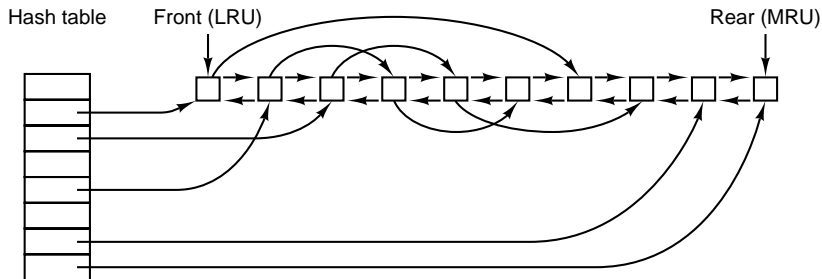
UFS e derivati ammettono anche il **fragment** (tipicamente 1/4 del block size).

*disk cache* – usare memoria RAM per bufferizzare i blocchi più usati. Può essere

- sul controller: usato come **buffer di traccia** per ridurre la latenza a 0 (quasi)
- (gran) parte della memoria principale, prelevando pagine dalla free list. Può arrivare a riempire tutta la memoria RAM: “un byte non usato è un byte sprecato”.

# Migliorare le performance: caching

I buffer sono organizzati in una coda con accesso hash



- La coda può essere gestita LRU, o CLOCK, ...
- Un blocco viene salvato su disco quando deve essere liberato dalla coda.
- Se blocchi critici vengono modificati ma non salvati mai (perché molto acceduti), si rischia l'inconsistenza in seguito ai crash.

# Migliorare le performance: caching

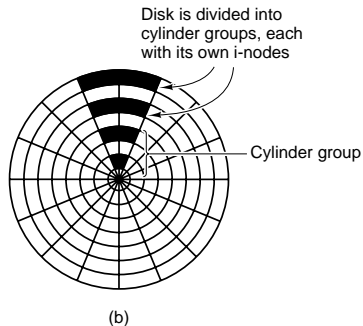
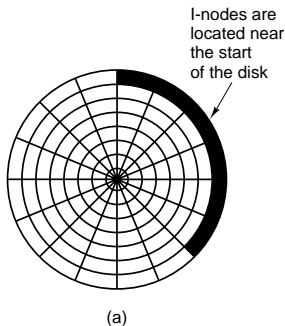
- Variante di LRU: dividere i blocchi in categorie a seconda se
  - il blocco verrà riusato a breve? in tal caso, viene messo in fondo alla lista.
  - il blocco è critico per la consistenza del file system? (tutti i blocchi tranne quelli dati) allora ogni modifica viene immediatamente trasferita al disco.

Anche le modifiche ai blocchi dati vengono trasferite prima della deallocazione:

- asincrono: ogni 20-30 secondi (Unix, Windows)
- sincrono: ogni scrittura viene immediatamente trasferita anche al disco (**write-through cache**, DOS).

- *read-ahead*: leggere blocchi in cache **prima** che siano realmente richiesti.
  - Aumenta il throughput del device
  - Molto adatto a file che vengono letti in modo sequenziale; Inadatto per file ad accesso casuale (es. librerie)
  - Il file system può tenere traccia del modo di accesso dei file per migliorare le scelte.
- Ridurre il movimento del disco
  - durante la scrittura del file, sistemare vicini i blocchi a cui si accede di seguito (facile con bitmap per i blocchi liberi, meno facile con liste)

- Ridurre il movimento del disco
  - raggruppare (e leggere) i blocchi in gruppi (cluster)
  - collocare i blocchi con i metadati (inode, p.e.) presso i rispettivi dati



# Affidabilità del file system

- I dispositivi di memoria di massa hanno un MTBF relativamente breve
- Inoltre i crash di sistema possono essere causa di perdita di informazioni in cache non ancora trasferite al supporto magnetico.
- Due tipi di affidabilità:
  - **Affidabilità dei dati**: avere la certezza che i dati salvati possano venir recuperati.
  - **Affidabilità dei metadati**: garantire che i metadati non vadano perduti/alterati (struttura del file system, bitmap dei blocchi liberi, directory, inodes. . .).
- Perdere dei dati è costoso; perdere dei metadati è critico: può comportare la perdita della **consistenza del file system** (spesso irreparabile e molto costoso).

## Possibili soluzioni per aumentare l'affidabilità dei dati

- Aumentare l'affidabilità dei dispositivi (es. RAID).
- Backup (automatico o manuale) dei dati dal disco ad altro supporto (altro disco, nastri, ... )
  - dump **fisico**: direttamente i blocchi del file system (veloce, ma difficilmente incrementale e non selettivo)
  - dump **logico**: porzioni del virtual file system (più selettivo, ma a volte troppo astratto (link, file con buchi. . . ))

Recupero dei file perduti (o interi file system) dal backup: dall'amministratore, o direttamente dall'utente.



# Consistenza del file system

- Alcuni blocchi contengono informazioni critiche sul file system (specialmente quelli contenenti metadati)
- Per motivi di efficienza, questi blocchi critici non sono sempre sincronizzati (a causa delle cache)
- Consistenza del file system: in seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contraddittorie.
- Due approcci al problema della consistenza del file system:
  - **curare le inconsistenze** dopo che si sono verificate, con programmi di controllo della consistenza (**scandisk, fsck**): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze. Lenti, e non sempre funzionano.

**prevenire le inconsistenze:** i journalled file system.

# Journalled File System

Nei file system **journalled** (o **journaling**) si usano strutture e tecniche da DBMS (B+tree e “transazioni”) per aumentare affidabilità (e velocità complessiva)

- Variazioni dei metadati (inodes, directories, bitmap, ...) sono scritti immediatamente in un'area a parte, il **log** o **giornale**, prima di essere effettuate.
- Dopo un crash, per ripristinare la consistenza dei metadati è sufficiente ripercorrere il log  $\Rightarrow$  non serve il *fsck*!
- Adatti a situazioni mission-critical (alta affidabilità, minimi tempi di recovery) e grandi quantità di dati.
- Esempi di file system journalled:

**XFS** (SGI, su IRIX e Linux): fino a  $2^{64} = 16$  exabytes  $>$  16 milioni TB

**JFS** (IBM, su AIX e Linux): fino a  $2^{55} = 32$  petabyte  $>$  32 mila TB

**ReiserFS e EXT3** (su Linux): fino a 16TB e 4TB, rispettivamente

# Esempio di layout di un disco fisico

