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

Laurea in Informatica — A.A. 2003/04

Trasparenze del Corso di *Sistemi Operativi*

Marino Miculan

Università di Udine

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.

Il File System

Alcune necessità dei processi:

- Memorizzare e trattare grandi quantità di informazioni (> memoria principale)
- Più processi devono avere la possibilità di accedere alle informazioni in modo concorrente e coerente, nello spazio e nel tempo
- Si deve garantire integrità, indipendenza, persistenza e protezione dei dati

L'accesso diretto ai dispositivi di memorizzazione di massa (come visto nella gestione dell'I/O) non è sufficiente.

I File

La soluzione sono i *file* (archivi):

- File = insieme di informazioni correlate a cui è stato assegnato un nome
- Un file è la più piccola porzione unitaria di memoria logica secondaria allocabile dall'utente o dai processi di sistema.
- La parte del S.O. che realizza questa astrazione, nascondendo i dettagli implementativi legati ai dispositivi sottostanti, è il *file system*.
- Esternamente, il file system è spesso l'aspetto più visibile di un S.O. (S.O. *documentocentrici*): come si denominano, manipolano, accedono, quali sono le loro strutture, i loro attributi, etc.
- Internamente, il file system si appoggia alla gestione dell'I/O per implementare ulteriori funzionalità.

Attributi dei file (metadata)

Nome identificatore del file. L'unica informazione umanamente leggibile

Tipo nei sistemi che supportano più tipi di file. Può far parte del nome

Locazione puntatore alla posizione del file sui dispositivi di memorizzazione

Dimensioni attuale, ed eventualmente massima consentita

Protezioni controllano chi può leggere, modificare, creare, eseguire il file

Identificatori dell'utente che ha creato/possiede il file

Varie date e timestamp di creazione, modifica, aggiornamento info...

Queste informazioni (*metadati*: dati sui dati) sono solitamente mantenute in apposite strutture (*directory*) residenti in memoria secondaria.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Denominazione dei file

- I file sono un meccanismo di astrazione, quindi ogni oggetto deve essere denominato.
- Il *nome* viene associato al file dall'utente, ed è solitamente necessario (ma non sufficiente) per accedere ai dati del file
- Le regole per denominare i file sono fissate dal file system, e sono molto variabili
  - lunghezza: fino a 8, a 32, a 255 caratteri
  - tipo di caratteri: solo alfanumerici o anche speciali; e da quale set? ASCII, ISO-qualcosa, Unicode?
  - case sensitive, insensitive
  - contengono altri metadati? ad esempio, il tipo?

Tipi dei file — FAT: name.extension

Tipo	Estensione	Funzione
Eseguibile	exe, com, bin o nessuno	programma pronto da eseguire, in liguaggio macchina
Oggetto	obj, o	compilato, in linguaggio macchina, non linkato
Codice sorgente	c, p, pas, f77, asm, java	codice sorgente in diversi linguaggi
Batch	bat, sh	script per l'interprete comandi
Testo	txt, doc	documenti, testo
Word processor	wp, tex, doc	svariati formati
Librarie	lib, a, so, dll	librerie di routine
Grafica	ps, dvi, gif	FILE ASCII o binari
Archivi	arc, zip, tar	file correlati, raggruppati in un file, a volte compressi

Tipi dei file — Unix: nessuna assunzione

Unix non forza nessun tipo di file a livello di sistema operativo: non ci sono metadati che mantengono questa informazione.

Tipo e contenuto di un file slegati dal nome o dai permessi.

Sono le applicazioni a sapere di cosa fare per ogni file (ad esempio, i client di posta usano i MIME-TYPES).

È possibile spesso "indovinare" il tipo ispezionando il contenuto alla ricerca dei *magic numbers*: utility `file`

```
$ file iptables.sh risultati Lucidi
iptables.sh: Bourne shell script text executable
risultati:   ASCII text
Lucidi:      PDF document, version 1.2
```

Tipi dei file — MacOS classico: molti metadati

Nel MacOS Classic ogni file è composto da 3 componenti:

- *data fork*: sequenza non strutturata, simile a quelli Unix o DOS
- *resource fork*: strutturato, contiene codice, icone, immagini, etichette, dialoghi, ...
- *info*: metadati sul file stesso, tra cui *applicativo creatore* e *tipo*

L'operazione di apertura (*doppio click*) esegue l'applicativo indicato nelle info.

Struttura dei file

- In genere, un file è una sequenza di bit, byte, linee o record il cui significato è assegnato dal creatore.
- A seconda del tipo, i file possono avere struttura
  - nessuna: sequenza di parole, byte
  - sequenza di record: linee, blocchi di lunghezza fissa/variabile
  - strutture più complesse: documenti formattati, archivi (ad albero, con chiavi, ...), eseguibili rilocabili (ELF, COFF)
  - I file strutturati possono essere implementati con quelli non strutturati, inserendo appropriati caratteri di controllo
- Chi impone la struttura: due possibilità
  - il sistema operativo: specificato il tipo, viene imposta la struttura e modalità di accesso. Più astratto.
  - l'utente: tipo e struttura sono delegati al programma, il sistema operativo implementa solo file non strutturati. Più flessibile.

Operazioni sui file

Creazione: due passaggi: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system

Cancellazione: staccare il file dal file system e deallocare lo spazio assegnato al file

Apertura: caricare alcuni metadati dal disco nella memoria pricipale, per velocizzare le chiamate seguenti

Chiusura: deallocare le strutture allocate nell'apertura

Lettura: dato un file e un *puntatore di posizione*, i dati da leggere vengono trasferiti dal *media* in un buffer in memoria

Scrittura: dato un file e un *puntatore di posizione*, i dati da scrivere vengono trasferiti sul *media*

Append: versione particolare di scrittura

Riposizionamento (seek): non comporta operazioni di I/O

Troncamento: azzerare la lunghezza di un file, mantenendo tutti gli altri attributi

Lettura dei metadati: leggere le informazioni come nome, timestamp, etc.

Scrittura dei metadati: modificare informazioni come nome, timestamps, protezione, etc.

Tabella dei file aperti

Queste operazioni richiedono la conoscenza delle informazioni contenute nelle directory. Per evitare di accedere continuamente alle dir, si mantiene in memoria una *tabella dei file aperti*. Due nuove operazioni sui file:

- Apertura: allocazione di una struttura in memoria (*file descriptor* o *file control block*) contenente le informazioni riguardo un file
- Chiusura: trasferimento di ogni dato in memoria al dispositivo, e deallocazione del file descriptor

A ciascun file aperto si associa

- Puntatore al file: posizione raggiunta durante la lettura/scrittura
- Contatore dei file aperti: quanti processi stanno utilizzando il file
- Posizione sul disco

Metodi di accesso: accesso sequenziale

- Un puntatore mantiene la posizione corrente di lettura/scrittura
- Si può accedere solo progressivamente, o riportare il puntatore all'inizio del file.
- Operazioni:

  read next
  write next
  reset
  no *read* dopo l'ultimo *write*
  (*rewrite*)

- Adatto a dispositivi intrinsecamente sequenziali (p.e., nastri)

Metodi di accesso: accesso diretto

- Il puntatore può essere spostato in qualunque punto del file
- Operazioni:

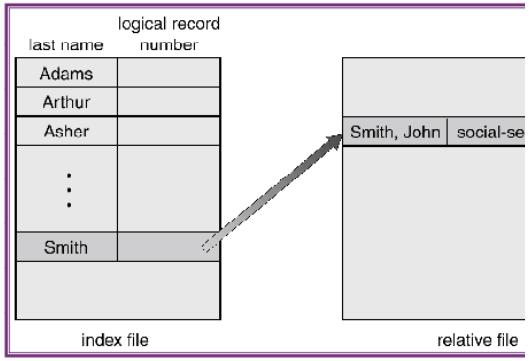
  read n
  write n
  seek n
  read next
  write next
  rewrite n

  n = posizione relativa a quella attuale

- L'accesso sequenziale viene simulato con l'accesso diretto
- Usuale per i file residenti su device a blocchi (p.e., dischi)

Metodi di accesso: accesso indicizzato

- Un secondo file contiene solo parte dei dati, e puntatori ai blocchi (record) del vero file
- La ricerca avviene prima sull'indice (corto), e da qui si risale al blocco

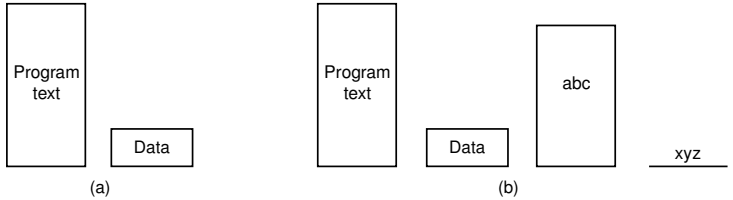


- Implementabile a livello applicazione in termini di file ad accesso diretto
- Usuale su mainframe (IBM, VMS), databases...

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```


File mappati in memoria

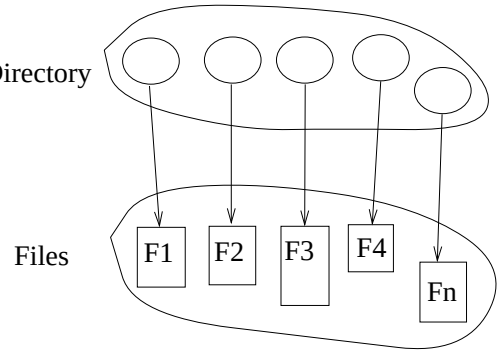
- Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria.



- Relativamente semplice da implementare in sistemi segmentati (con o senza paginazione): il file viene visto come area di swap per il segmento mappato
- Non servono chiamate di sistema `read` e `write`, solo una `mmap`
- Problemi
  - lunghezza del file non nota al sistema operativo
  - accesso condiviso con modalità diverse
  - lunghezza del file maggiore rispetto alla dimensione massima dei segmenti.

Directory

- Una directory è una collezione di nodi contenente informazioni sui file (*metadati*)
- Sia la directory che i file risiedono su disco
- Operazioni su una directory
  - Ricerca di un file
  - Creazione di un file
  - Cancellazione di un file
  - Listing
  - Rinomina di un file
  - Navigazione del file system



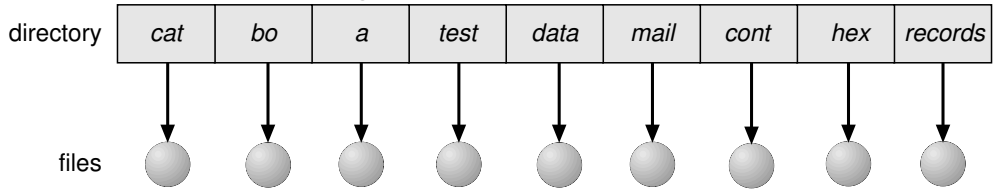
Organizzazione logica delle directory

Le directory devono essere organizzate per ottenere

- efficienza: localizzare rapidamente i file
- nomi mnemonici: comodi per l'utente
  - file differenti possono avere lo stesso nome
  - più nomi possono essere dati allo stesso file
- Raggruppamento: file logicamente collegati devono essere raccolti assieme (e.g., i programmi in C, i giochi, i file di un database, . . .)

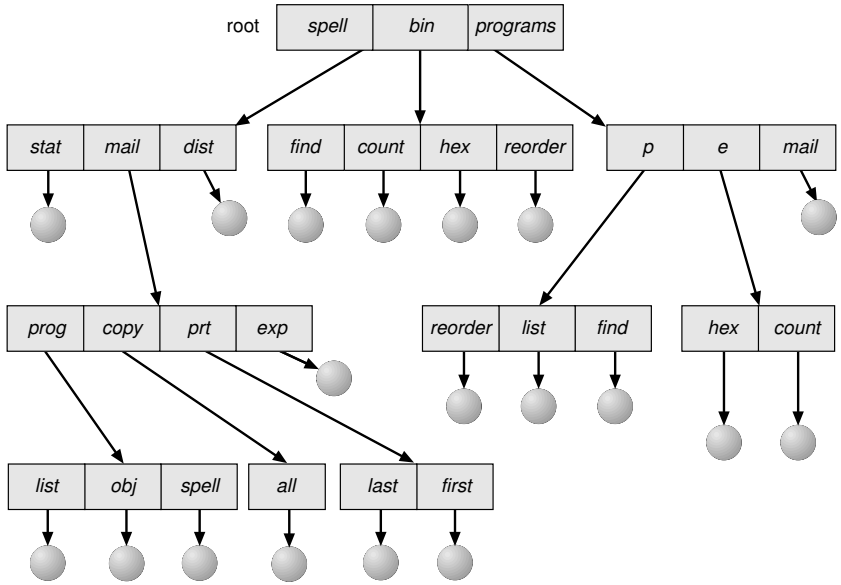
Tipi di directory: unica ("flat")

- Una sola directory per tutti gli utenti



- Problema di raggruppamento e denominazione
- Obsoleta
- Variante: a due livelli (una directory per ogni utente)

Tipi di directory: ad albero



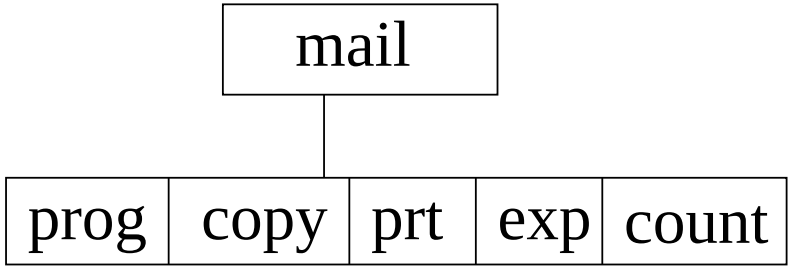
Directory ad albero (cont.)

- Ricerca efficiente
- Raggruppamento
- Directory corrente (working directory): proprietà del processo
  - **cd** /home/miculan/src/C
  - **cat** hw.c
- Nomi assoluti o relativi
- Le operazioni su file e directory (lettura, creazione, cancellazione, . . .) sono relative alla directory corrente

Esempio: se la dir corrente è `/spell/mail`

mkdir count

crea la situazione corrente

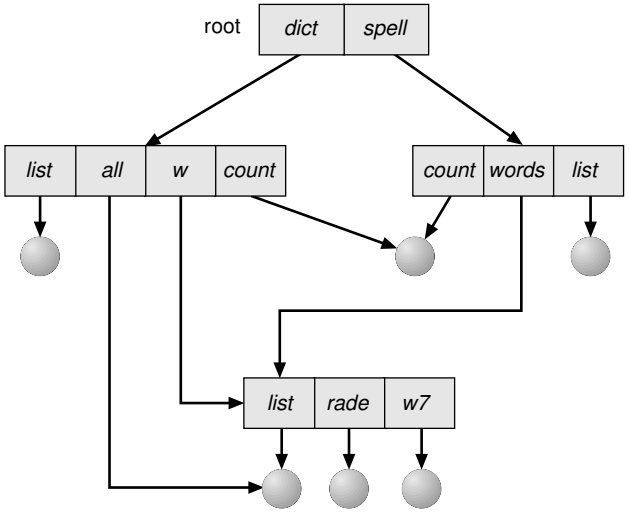


- Cancellando `mail` si cancella l'intero sottoalbero

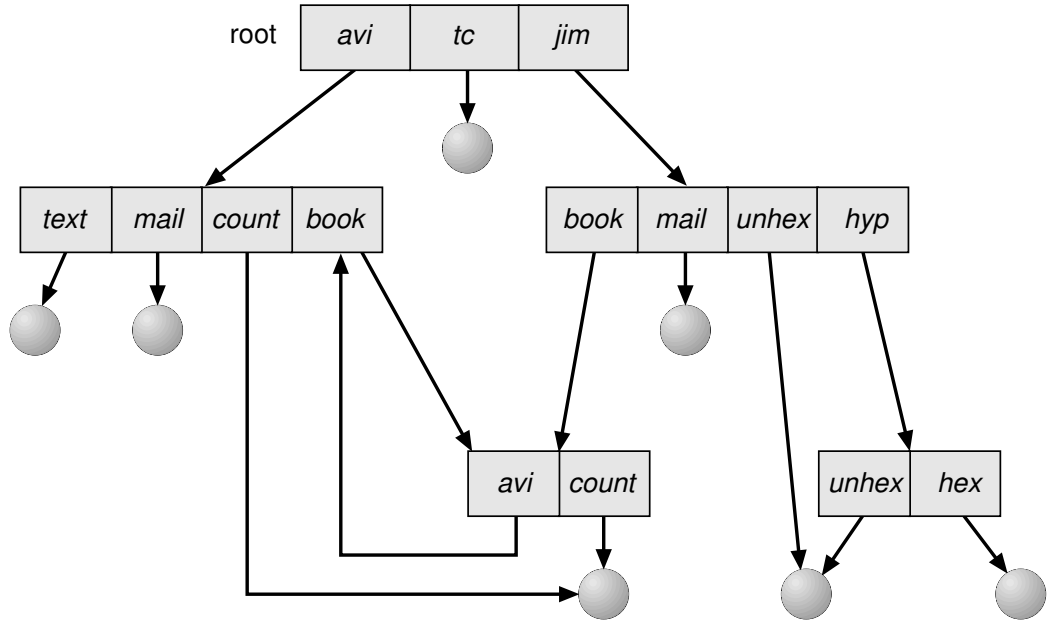
Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory
Due nomi differenti per lo stesso file (aliasing)
Possibilità di puntatori "dangling". Soluzioni

- Puntatori all'indietro, per cancellare tutti i puntatori. Problematici perché la dimensione dei record nelle directory è variabile.
- Puntatori a daisy chain
- Contatori di puntatori per ogni file (UNIX)



Directory a grafo



Directory a grafo (cont.)

I cicli sono problematici per la

- Visita: algoritmi costosi per evitare loop infiniti
- Cancellazione: creazione di *garbage*

Soluzioni:

- Permettere solo link a file (UNIX per i link hard)
- Durante la navigazione, limitare il numero di link attraversabili (UNIX per i simbolici)
- Garbage collection (costosa!)
- Ogni volta che un link viene aggiunto, si verifica l'assenza di cicli. Algoritmi costoso.

Protezione

- Importante in ambienti multiuser dove si vuole condividere file
- Il creatore/possessore (non sempre coincidono) deve essere in grado di controllare
  - cosa può essere fatto
  - e da chi (in un sistema multiutente)
- Tipi di accesso soggetti a controllo (non sempre tutti supportati):
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Matrice di accesso

Sono il metodo di protezione più generale

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Matrice di accesso (cont.)

- per ogni coppia (processo,oggetto), associa le operazioni permesse
- matrice molto sparsa: si implementa come
  - *access control list:* ad ogni oggetto, si associa chi può fare cosa.

    Sono implementate da alcuni UNIX (e.g., *getfacl(1)* e *setfacl(1)* su Solaris)
  - *capability tickets:* ad ogni processo, si associa un insieme di tokens che indicano cosa può fare

Modi di accesso e gruppi in UNIX

Versione semplificata di ACL.

- Tre modi di accesso: **r**ead, **w**rite, **e**xecute
- Tre classi di utenti, per ogni file

				RWX
a)	owner access	7	$\Rightarrow$	1 1 1
b)	groups access	6	$\Rightarrow$	1 1 0
c)	public access	1	$\Rightarrow$	0 0 1

- Ogni processo possiede UID e GID, con i quali si verifica l'accesso

Modi di accesso e gruppi in UNIX

- Per limitare l'accesso ad un gruppo di utenti, si chiede al sistemista di creare un gruppo apposito, sia G, e di aggiungervi gli utenti.
- Si definisce il modo di accesso al file o directory
- Si assegna il gruppo al file:

  chgrp G *game*

Effective User e Group ID

- In UNIX, il dominio di protezione di un processo viene ereditato dai suoi figli, e viene impostato al login
- In questo modo, tutti i processi di un utente girano con il suo UID e GID.
- Può essere necessario, a volte, concedere temporaneamente privilegi speciali ad un utente (es: *ps*, *lpr*, . . .)
  - *Effective* UID e GID (EUID, EGID): due proprietà extra di tutti i processi (stanno nella U-structure).
  - Tutti i controlli vengono fatti rispetto a EUID e EGID
  - Normalmente, EUID=UID e EGID=GID
  - L'utente `root` può cambiare questi parametri con le system call *setuid(2), setgid(2), seteuid(2), setegid(2)*

Setuid/setgid bit

- l'Effective UID e GID di un processo possono essere cambiati per la durata della sua esecuzione attraverso i bit **setuid** e **setgid**
- Sono dei bit supplementari dei file eseguibili di UNIX

```
miculan@coltrane:Lucidi$ ls -l /usr/bin/lpr
-r-sr-sr-x   1 root     lp          15608 Oct 23 07:51 /usr/bin/lpr*
miculan@coltrane:Lucidi$
```

- Se **setuid** bit è attivo, l'EUID di un processo che esegue tale programma diventa lo stesso del possessore del file
- Se **setgid** bit è attivo, l'EGID di un processo che esegue tale programma diventa lo stesso del possessore del file
- I real UID e GID rimangono inalterati

Setuid/setgid bit (cont.)

- Si impostano con il chmod

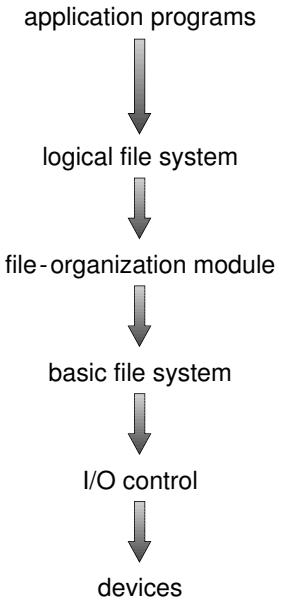
```
miculan@coltrane:C$ ls -l a.out
-rwxr-xr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 2755 a.out
miculan@coltrane:C$ ls -l a.out
-rwxr-sr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 4755 a.out
miculan@coltrane:C$ ls -l a.out
-rwsr-xr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$
```


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, etc.

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 e 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

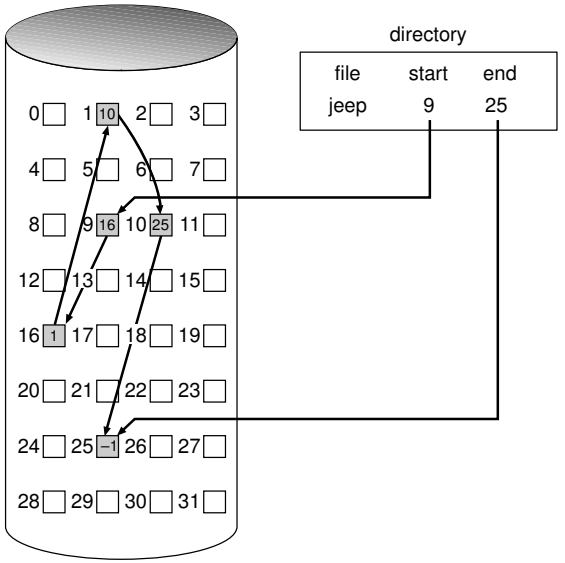
- Traduzione dall'indirizzo logico a quello fisico (per blocchi da 512 byte):

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

  - 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



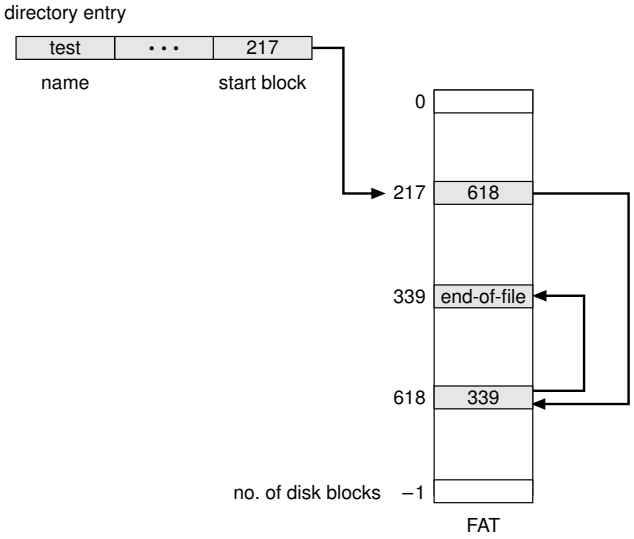
- Allocazione su richiesta; i blocchi vengono semplicemente collegati alla fine del file
- Semplice: basta sapere l'indirizzo del primo blocco
- Non c'è frammentazione esterna
- Bisogna gestire i blocchi liberi
- Non supporta l'accesso diretto (seek)
- Traduzione indirizzo logico:

$$\text{LA}/511 \begin{cases} \text{Q} \\ \text{R} \end{cases}$$

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

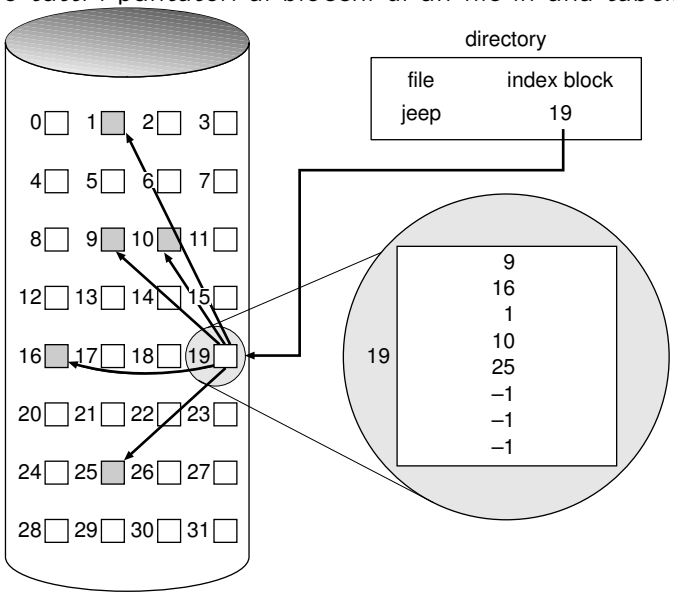
Allocazione concatenata (cont.)

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*.



- 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

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

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

Allocazione indicizzata (cont.)

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

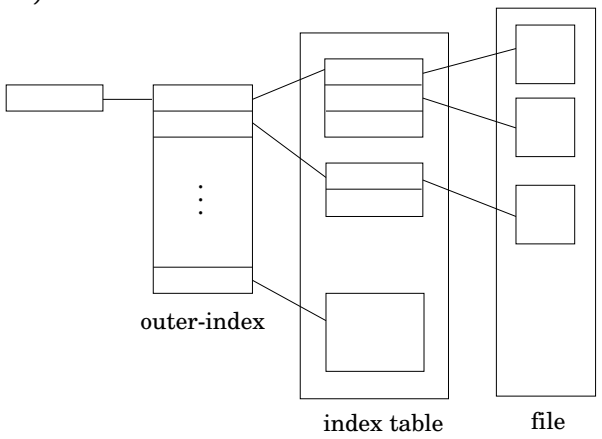
$$\text{LA/(512 x 511)} \begin{cases} \text{Q}_1 \\ \text{R}_1 \end{cases}$$

$$\text{R}_1\text{/512} \begin{cases} \text{Q}_2 \\ \text{R}_2 \end{cases}$$

  - 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 (cont.)

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:

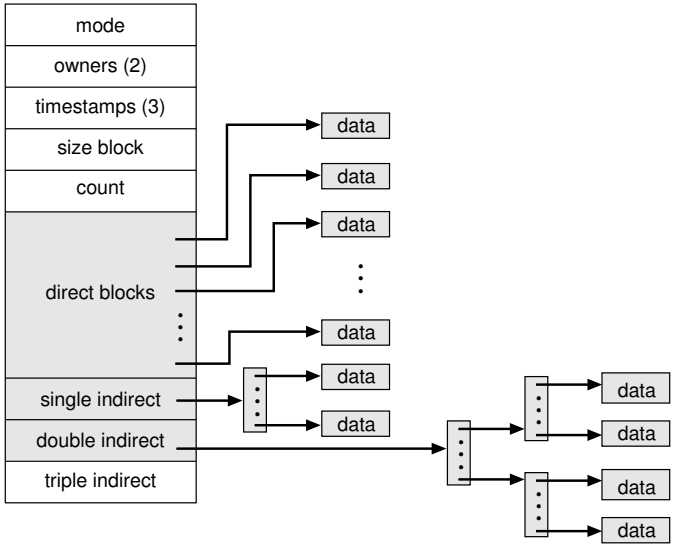
$$\mathrm{LA}/(512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

$$R_1/512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

  - 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!)

Inodes (cont.)

- 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

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

  molto oltre le capacità dei sistemi a 32 bit.

Gestione dello spazio libero

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

  0101110101010111111011000000101000000010111010101111100

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

- 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

Gestione dello spazio libero (Cont.)

- 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

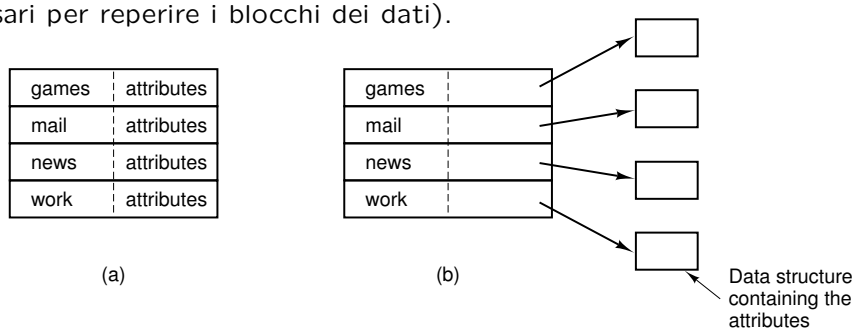
- Alternativa: *Linked list (free list)*

  – Inefficiente - non facile trovare blocchi liberi contigui

  – Non c'è spreco di spazio.

Implementazione delle directory

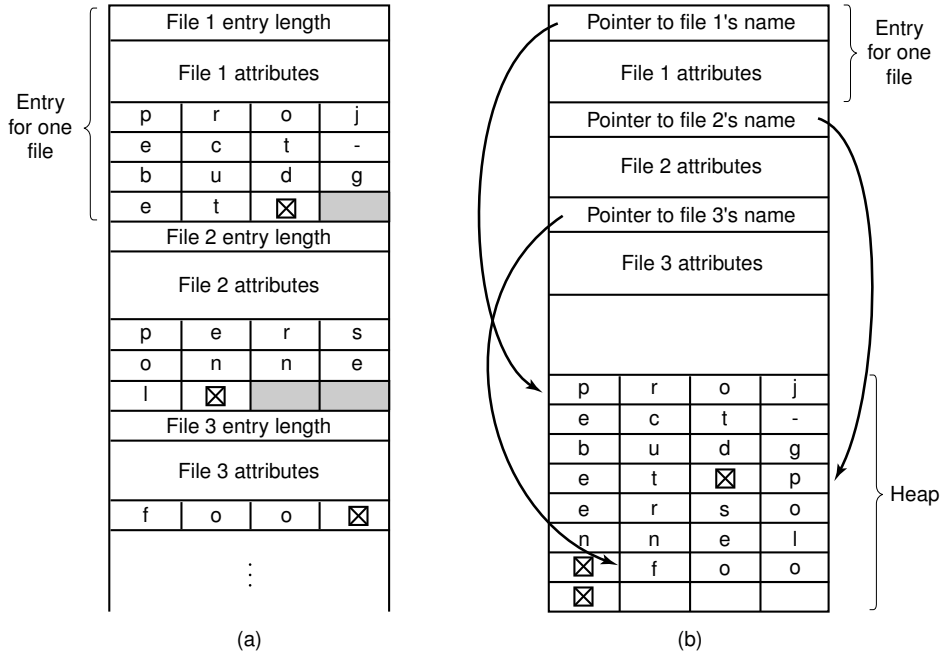
Le directory sono essenziali per passare dal nome del file ai suoi attributi (anche necessari per reperire i blocchi dei dati).



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



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

Efficienza e performance

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!

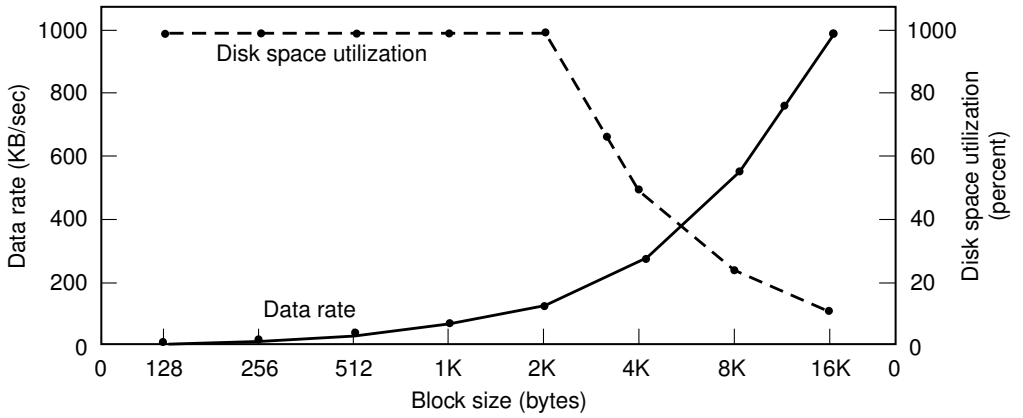
Sulla dimensione dei blocchi

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

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



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).

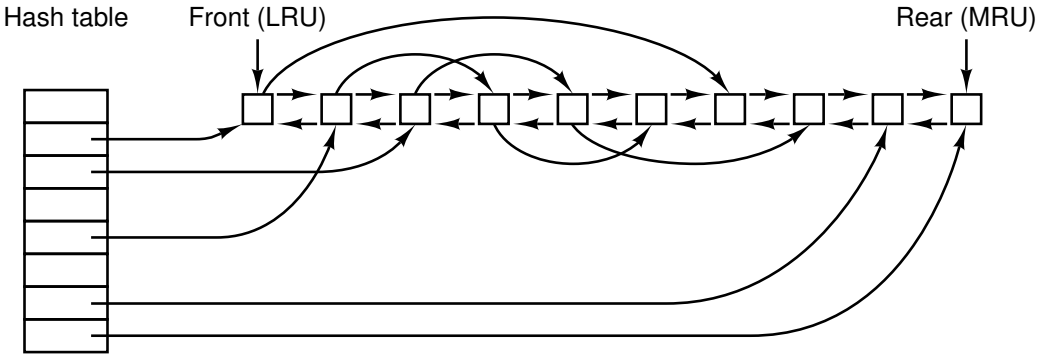
UFS e derivati ammettono anche il *fragment* (tipicamente 1/4 del block size).

Migliorare le performance: caching

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".

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.

- 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.

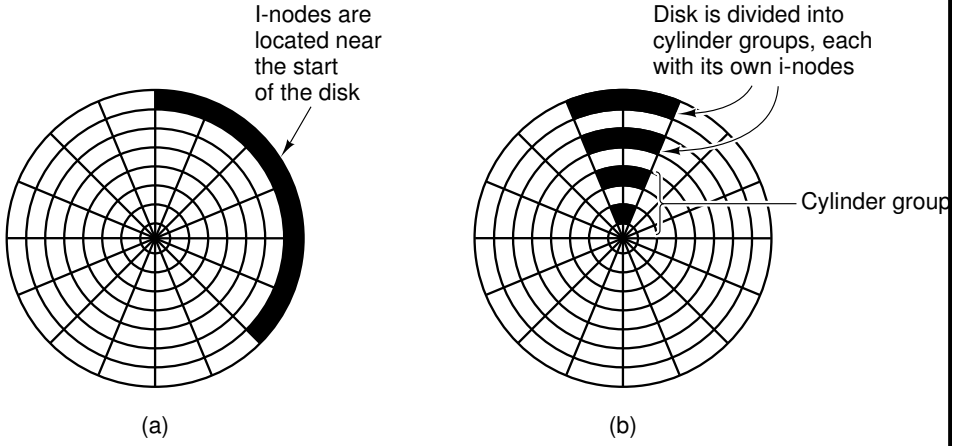
Anche le modifiche ai blocchi dati vengono trasferite prima della deallocazione:

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

Altri accorgimenti

- *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)
  - 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).

Affidabilità del dati

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 contradditorie.
- 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 l'inconsistenze: i journalled file system.

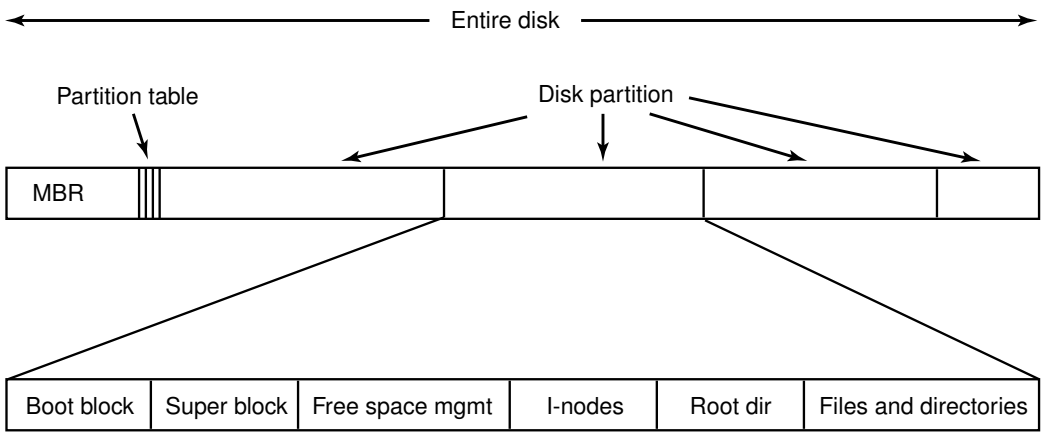
Journalled File System

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

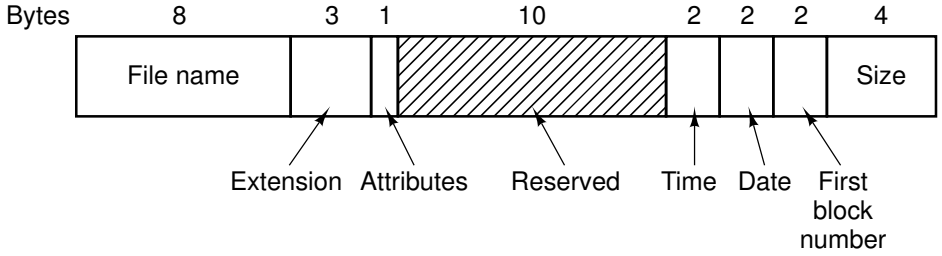
- Variazioni dei metadati (inodes, directories, bitmap, . . .) sono scritti immediatamente in un'area a parte, il *log* o *giornale*, prima di essere effettuate.
- Dopo un crash, per ripristinare la consistenza dei metadati è sufficiente ripercorrere il log $\Rightarrow$ non serve il *fsck*!
- Adatti 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



Directory in MS-DOS



- Lunghezza del nome fissa
- Attributi: read-only, system, archived, hidden
- Reserved: non usati
- Time: ore (5bit), min (6bit), sec (5bit)
- Date: giorno (5bit), mese (4bit), anno-1980 (7bit) (Y2108 BUG!)

FAT12, FAT16, FAT32

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

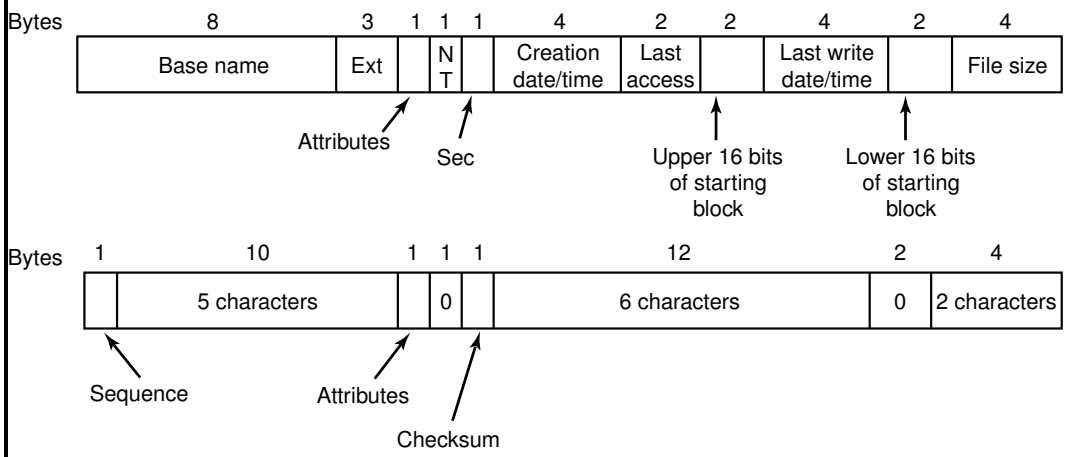
In MS-DOS, tutta la FAT viene caricata in memoria.

Il block size è chiamato da Microsoft *cluster size*

Limite superiore: 2^{32} blocchi da 512 byte = 2TB

Directory in Windows 98

Nomi lunghi ma compatibilità all'indietro con MS-DOS e Windows 3



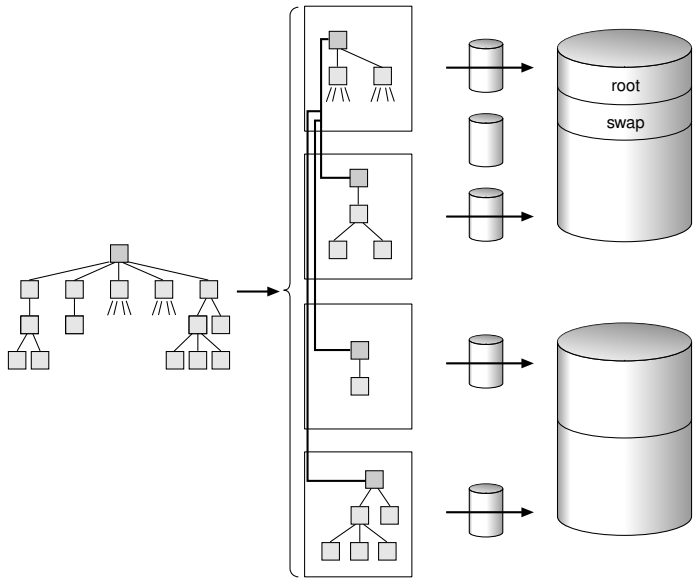
Aove attributes=0x0F (valore invalido per MS-DOS)

Esempio

```
$ dir
THEQUI~1        749 03-08-2000  15:38 The quick brown fox jumps over the...
```

Bytes																							
68	d	o	g				A	0	CK											0			
3	o	v	e				A	0	CK	t	h	e			l	a				0	z	y
2	w	n			f	o	A	0	CK	x			j	u	m	p				0	s	
1	T	h	e			q	A	0	CK	u	i	c	k			b				0	r	o
T	H	E	Q	U	I	~	1	A	NT	S	Creation time			Last acc	Upp	Last write				Low	Size		

UNIX: Il Virtual File System



Il file system *virtuale* che un utente vede può essere composto in realtà da diversi file system *fisici*, ognuno su un diverso dispositivo logico

Il Virtual File System (cont.)

- Il Virtual File System è composto da più file system fisici, che risiedono in dispositivi logici (*partizioni*), che compongono i dispositivi fisici (dischi)
- Il file system / viene montato al boot dal kernel
- gli altri file system vengono montati secondo la configurazione impostata
- ogni file system fisico può essere diverso o avere parametri diversi
- Il kernel usa una coppia <*logical device number, inode number*> per identificare un file
  - Il logical device number indica su quale file system fisico risiede il file
  - Gli inode di ogni file system sono numerati progressivamente

Il Virtual File System (cont.)

Il kernel si incarica di implementare una visione uniforme tra tutti i file system montati: operare su un file significa

- determinare su quale file system fisico risiede il file
- determinare a quale inode, su tale file system corrisponde il file
- determinare a quale dispositivo appartiene il file system fisico
- richiedere l'operazione di I/O al dispositivo

I File System Fisici di UNIX

- UNIX (Linux in particolare) supporta molti tipi di file system fisici: SYSV, UFS, EFS, EXT2, MSDOS, VFAT, ISO9660, HPFS, HFS, NTFS, ...
- Quelli preferiti sono UFS (Unix File System, aka BSD Fast File System), EXT2 (Extended 2), EFS (Extent File System) e i *journalled file systems* (JFS, XFS, EXT3, ...)
- Il file system fisico di UNIX supporta due oggetti:
  - file "semplici" (plain file) (senza struttura)
  - directory (che sono semplicemente file con un formato speciale)
- La maggior parte di un file system è composta da blocchi dati
  - in EXT2: 1K-4K (configurabile alla creazione)
  - in SYSV: 2K-8K (configurabile alla creazione)

Inodes

- Un file in Unix è rappresentato da un *inode* (nodo indice).
- Gli inodes sono allocati in numero finito alla creazione del file system
- Struttura di un inode in System V:

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Inodes (cont)

- I timestamp sono in POSIX "epoch": n. di secondi dal 01/01/1970, UTC. (Quindi l'epoca degli Unix a 32 bit dura 2^{31} secondi, ossia fino alle 3:14:07 UTC di martedì 19 gennaio 2038).
- 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

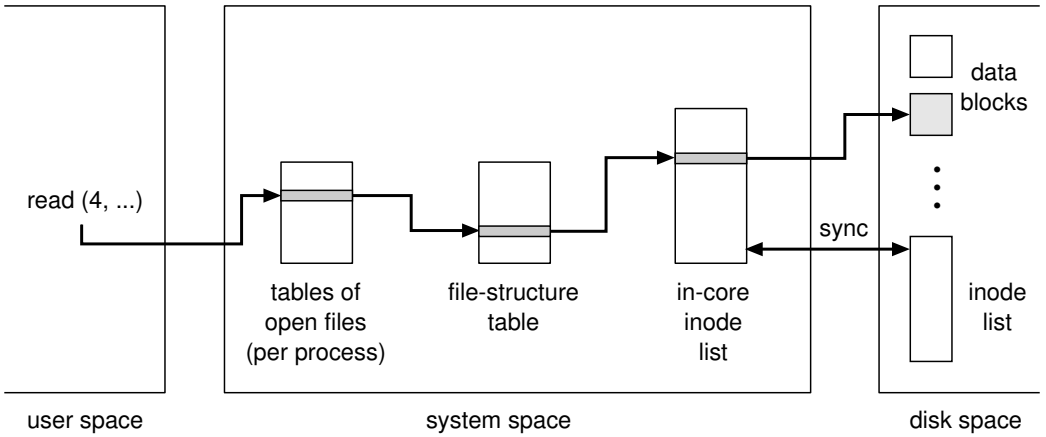
$$\begin{aligned} L_{max} &= 10 + 1024 + 1024^2 + 1024^3 \\ &> 1024^3 = 2^{30} blk \\ &= 2^{42} byte = 4\,TB \end{aligned}$$

molto oltre le capacità dei sistemi a 32 bit.

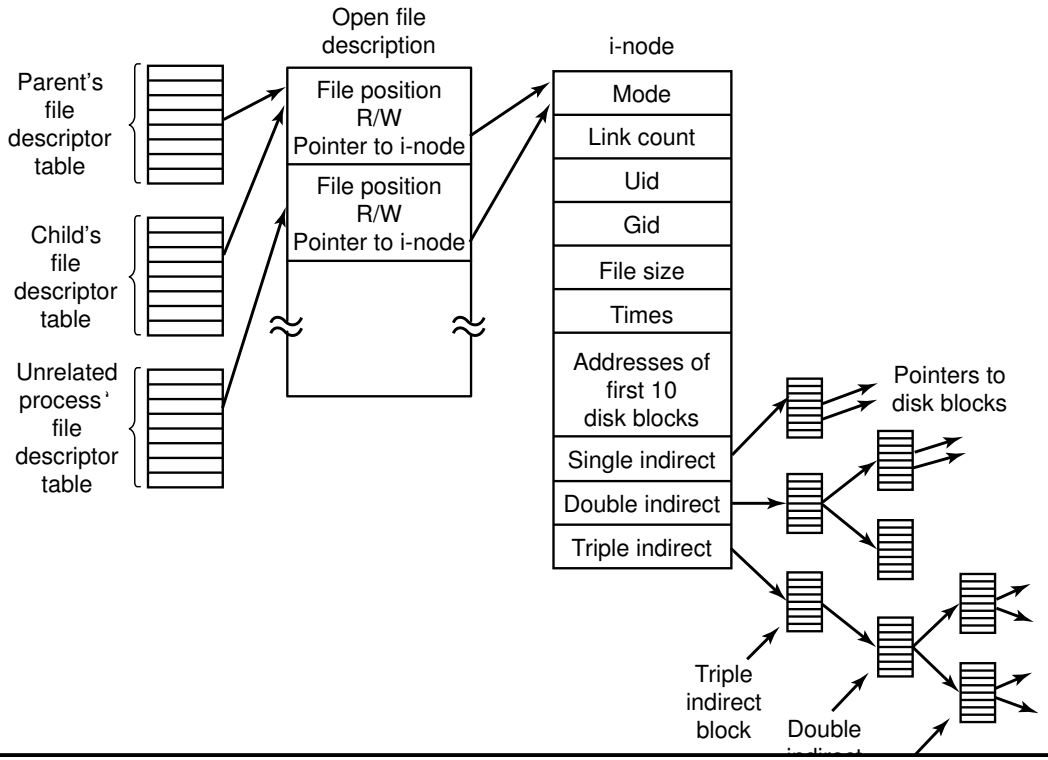
Traduzione da file descriptor a inode

- Le system calls che si riferiscono a file aperti (read, write, close, ...) prendono un *file descriptor* come argomento
- Il file descriptor viene usato dal kernel per entrare in una tabella di file aperti del processo. Risiede nella U-structure.
- Ogni entry della tabella contiene un puntatore ad una *file structure*, di sistema. Ogni file structure punta ad un inode (in un'altra lista), e contiene la posizione nel file.
- Ogni entry nelle tabelle contiene un contatore di utilizzo: quando va a 0, il record viene deallocato

File Descriptor, File Structure e Inode



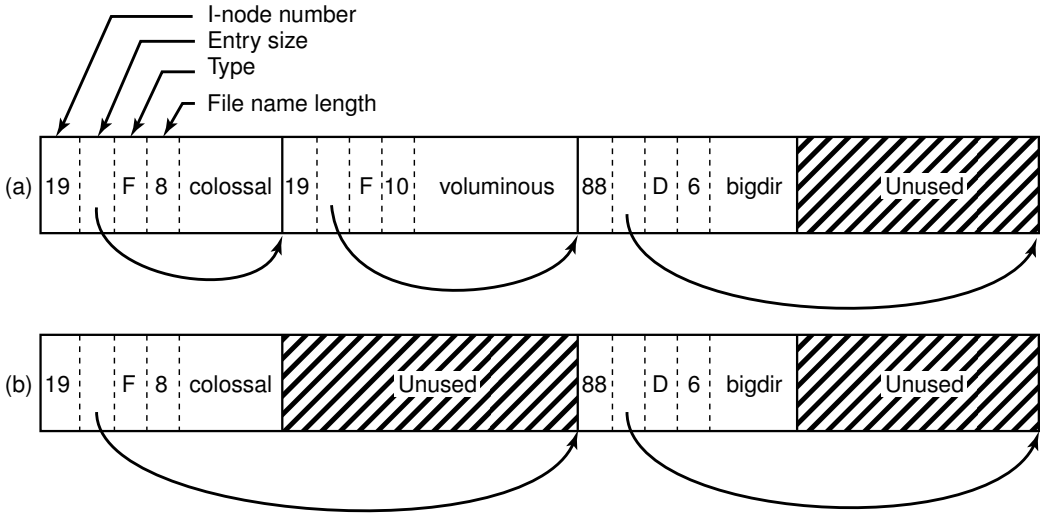
La tabella intermedia è necessaria per la semantica della condivisione dei file tra processi



- Le chiamate di lettura/scrittura e la seek cambiano la posizione nel file
- Ad una *fork*, i figli ereditano (una copia de) la tabella dei file aperti dal padre $\Rightarrow$ condividono la stessa file structure e quindi la posizione nel file
- Processi che hanno aperto indipendentemente lo stesso file hanno copie private di file structure

Directory in UNIX

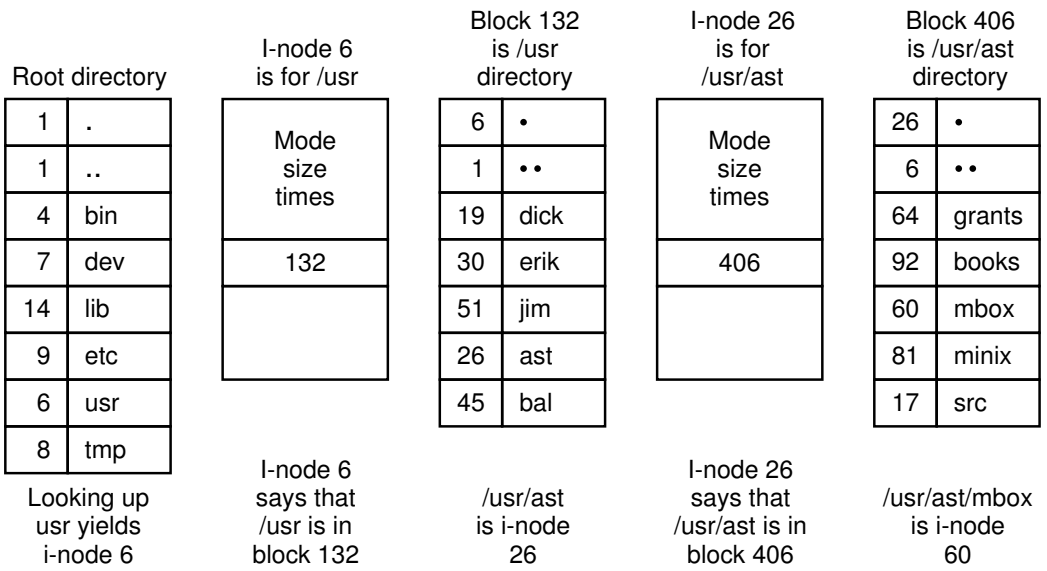
- Il tipo all'interno di un inode distingue tra file semplici e directory
- Una directory è un file con entry di lunghezza variabile. Ogni entry contiene
  - puntatore all'inode del file
  - posizione dell'entry successiva
  - lunghezza del nome del file (1 byte)
  - nome del file (max 255 byte)
- entry differenti possono puntare allo stesso inode (*hard link*)



Traduzione da nome a inode

L'utente usa i nomi (o path), mentre il file system impiega gli inode ⇒ il kernel deve risolvere ogni nome in un inode, usando le directory

- Prima si determina la directory di partenza: se il primo carattere è "/", è la root dir (sempre montata); altrimenti, è la current working dir del processo in esecuzione
- Ogni sezione del path viene risolta leggendo l'inode relativo
- Si ripete finché non si termina il path, o la entry cercate non c'è
- Link simbolici vengono letti e il ciclo di decodifica riparte con le stesse regole. Il numero massimo di link simbolici attraversabili è limitato (8)
- Quando l'inode del file viene trovato, si alloca una *file structure* in memoria, a cui punta il *file descriptor* restituito dalla *open(2)*



Esempio di file system fisico: Unix File System

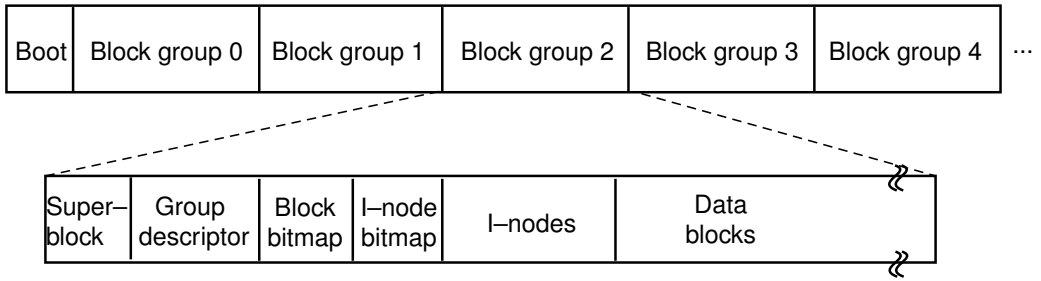
- In UFS (detto anche Berkley Fast File System), i blocchi hanno due dimensioni: il *blocco* (4-8K) e il *frammento* (0.5-1K)
  - Tutti i blocchi di un file sono blocchi tranne l'ultimo
  - L'ultima parte del file è tenuta in frammenti, q.b.
  - Es: un file da 18000 byte occupa 2 blocchi da 8K e 1 da 2K (non pieno)
- Riduce la frammentazione interna e aumenta la velocità di I/O
- La dimensione del blocco e del frammento sono impostati alla creazione del file system:
  - se ci saranno molti file piccoli, meglio un fragment piccolo
  - se ci saranno grossi file da trasferire spesso, meglio un blocco grande
  - il rapporto max è 8:1. Tipicamente, 4K:512 oppure 8K:1K.

Esempio di file system fisico: Unix File System (Cont)

- Si introduce una cache di directory per aumentare l'efficienza di traduzione
- Suddivisione del disco in *cilindri*, ognuno dei quali con il proprio superblock, tabella degli inode, dati. Quando possibile, si allocano i blocchi nello stesso gruppo dell'inode.

  In questo modo si riduce il tempo di seek dai metadati ai dati.

Esempio di file system fisico: EXT2



- Derivato da UFS, ma con blocchi tutti della stessa dimensione (1K-4K)
- Suddivisione del disco in *gruppi* di 8192 blocchi, ma non secondo la geometria fisica del disco
- Il *superblock* (blocco 0) contiene informazioni vitali sul file system
  - tipo di file system
  - primo inode

  - numero di gruppi
  - numero di blocchi liberi e inodes liberi,...
- Ogni gruppo ha una copia del superblock, la propria tabella di inode e tabelle di allocazione blocchi e inode
- Per minimizzare gli spostamenti della testina, si cerca di allocare ad un file blocchi dello stesso gruppo

NTFS: File System di Windows NT/2K/XP

- Un file è un oggetto strutturato costituito da *attributi*.
- Ogni attributo è una sequenza di byte distinta (*stream*), come in MacOS. Ogni stream è in pratica un file a se stante (con nome, dimensioni, puntatori, etc.).
- L'indirizzamento è a 64 bit.
- Tipicamente, ci sono brevi stream per i metadati (nome, attributi, Object ID) e un lungo stream per i veri dati, ma nulla vieta avere più stream di dati (es. file server per MacOS)
- I nomi sono lunghi fino a 255 caratteri Unicode.

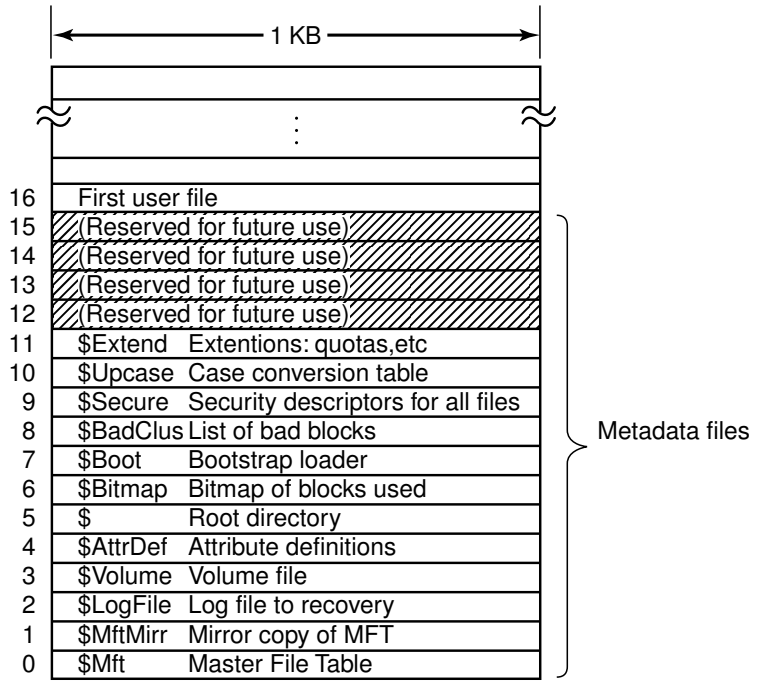
Struttura di NTFS

- Creato da zero, incompatibile all'indietro.
- Diverse partizioni possono essere unite a formare un *volume logico*
- Ha un meccanismo transazionale per i metadati (logging)
- Lo spazio viene allocato a *cluster:* potenze di 2 dipendenti dalla dimensione del disco (tra 512byte e 4K). All'interno di un volume, ogni cluster ha un *logical cluster number*.

Master File Table (MFT)

- È un file di record di 1K, ognuno dei quali descrive un file o una directory
- Può essere collocato ovunque, sul disco,e crescere secondo necessità
- Il primo blocco è indicato nel boot block.
- Le prime 16 entry descrivono l'MFT stesso e il volume (analogo al superblock di Unix). Indicano la posizione della root dir, il bootstrap loader, il logfile, spazio libero (gestito con una bitmap). . .
- Ogni record successivo è uno header seguito da una sequenza di coppie (attributo header,valore). Ogni header contiene il tipo dell'attributo, dove trovare il valore, e vari flag.

  I valori possono seguire il proprio header (*resident attribute*) o essere memorizzati in un blocco separato (*nonresident attribute*)



Attributi dei file NTFS

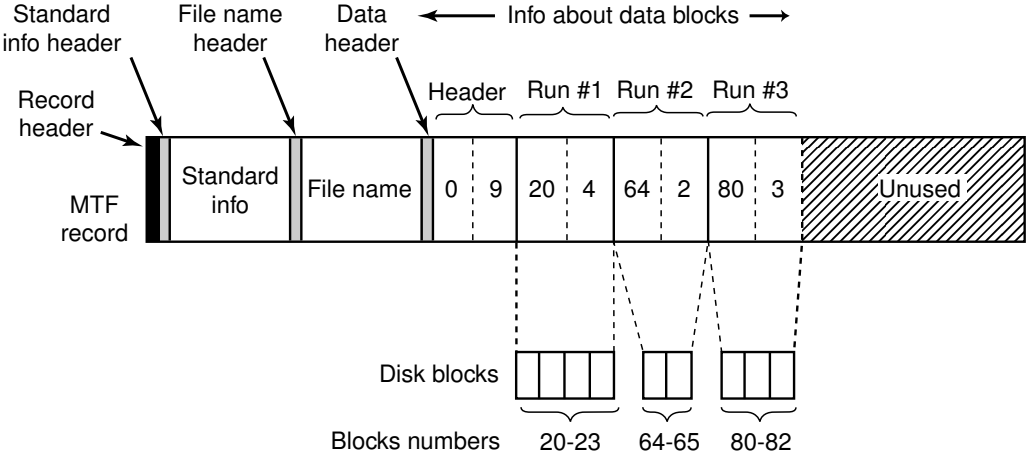
NTFS definisce 13 attributi standard

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

I Data contenono i veri dati; se sono residenti, il file si dice "immediate".

File NTFS non residenti

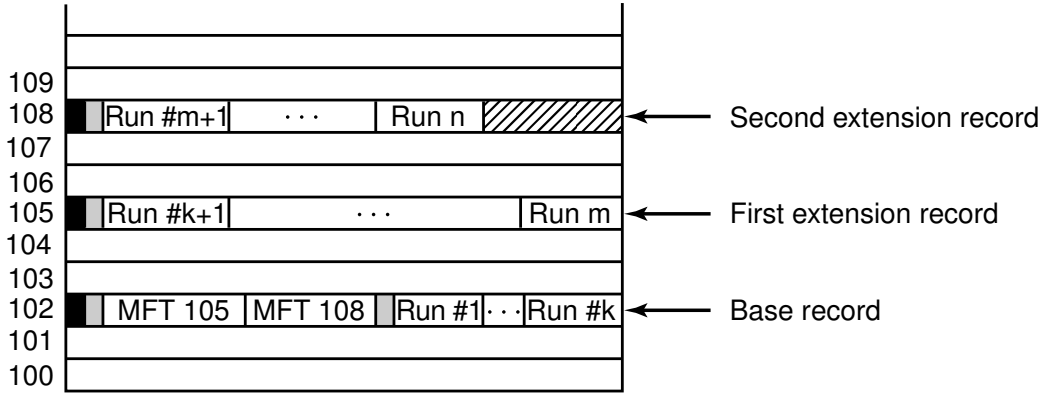
I file non immediati si memorizzano a "run": sequenze di blocchi consecutivi. Nel record MFT corrispondente ci sono i puntatori ai primi blocchi di ogni run.



Un file descritto da un solo MFT record si dice *short* (ma potrebbe non essere corto per niente!)

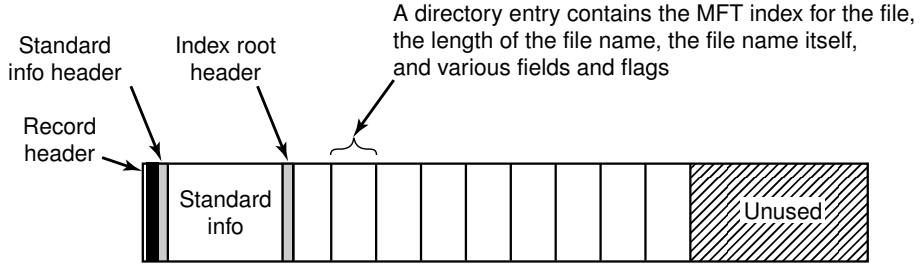
File "long"

Se il file è lungo o molto frammentato (es. disco frammentato), possono servire più di un record nell'MFT. Prima si elencano tutti i record aggiuntivi, e poi seguono i puntatori ai run.



Directory in NTFS

Le directory corte vengono implementate come semplici liste direttamente nel record MFT.



Directory più lunghe sono implementate come file nonresident strutturati a B+tree.