Gestione del File System

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 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*.
- Internamente, il file system si appoggia alla gestione dell'I/O per implementare ulteriori funzionalità.
- 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.

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 informazioni...

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 (e.g. i magic numbers: informazioni memorizzate all'inizio di un file) attraverso programmi di sistema come `file`

```
$ file iptables.sh risultati Lucidi
iptables.sh: Bourne shell script text executable
risultati:   ASCII text
Lucidi:      PDF document, version 1.2
p.dvi:       TeX DVI file (TeX output 2003.09.30:1337)
```

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, . . .)
  - 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

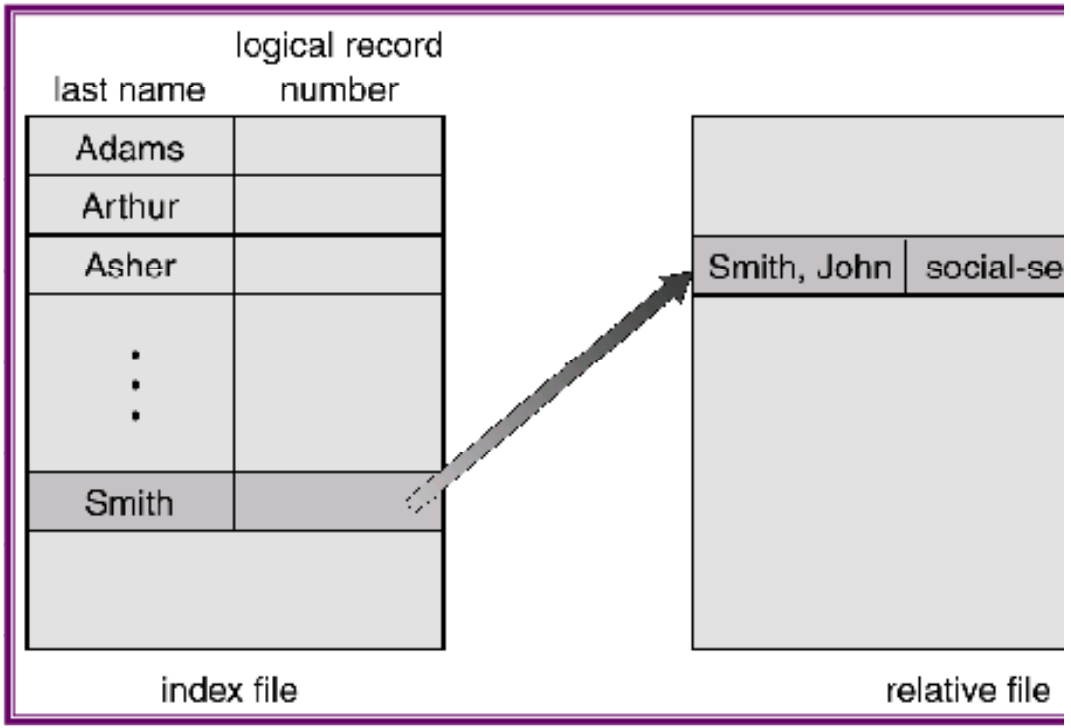
- Puntatore al file: posizione raggiunta durante la lettura/scrittura
- Contatore dei file aperti: quanti processi stanno utilizzando il file
- Posizione sul disco

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.
  - Adatto a dispositivi intrinsecamente sequenziali (p.e., nastri)
- Accesso diretto
  - Il puntatore può essere spostato in qualunque punto del file
  - 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

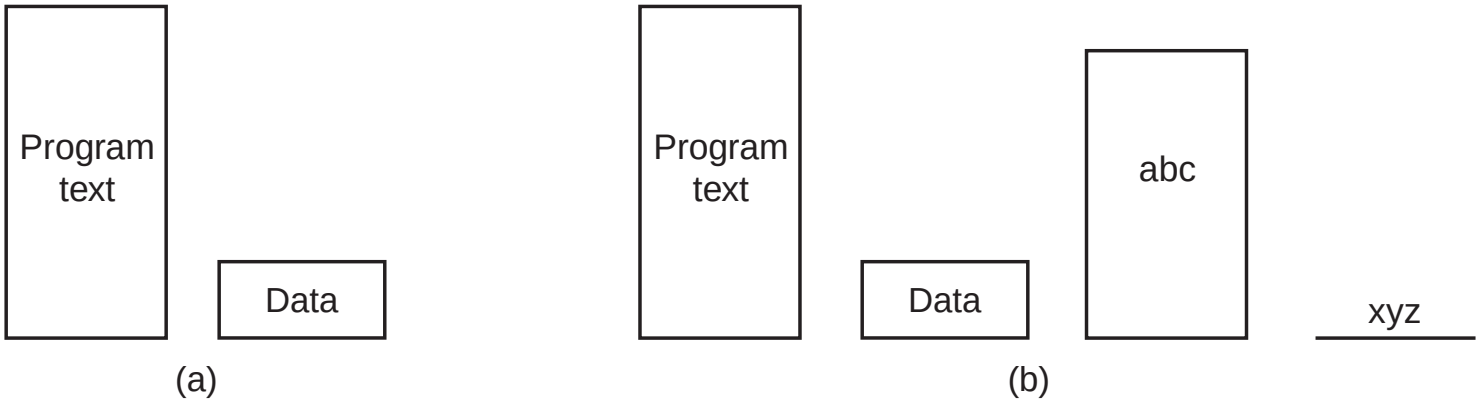


- Implementabile a livello applicazione in termini di file ad accesso diretto
- Usuale su mainframe (IBM, VMS), databases. . .

```
/* 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

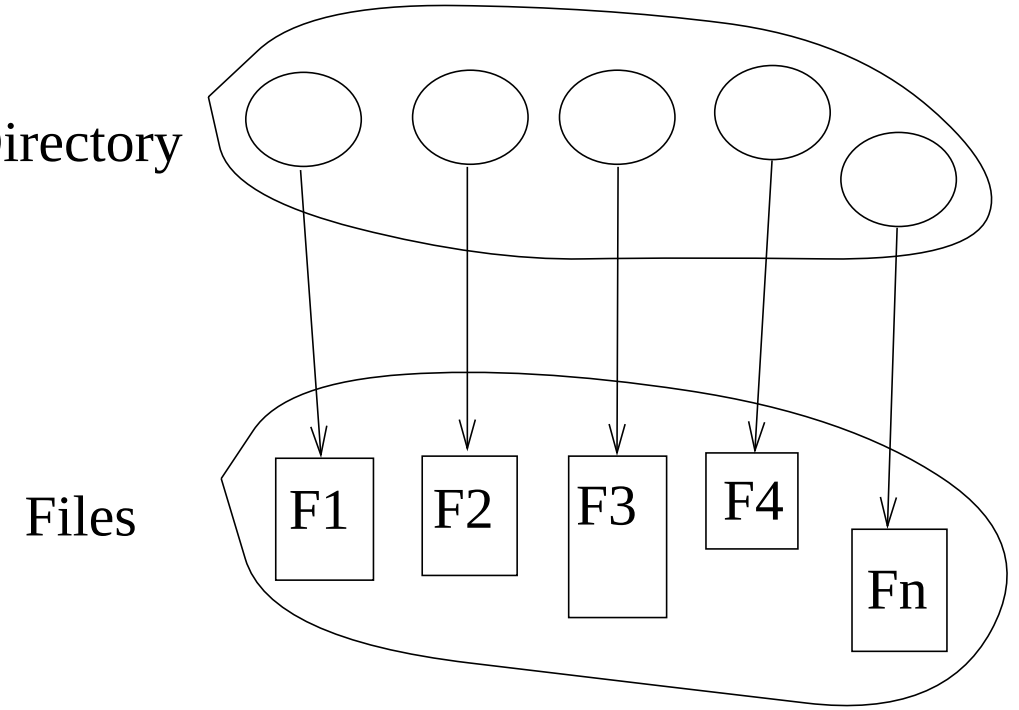
- Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria.



- 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



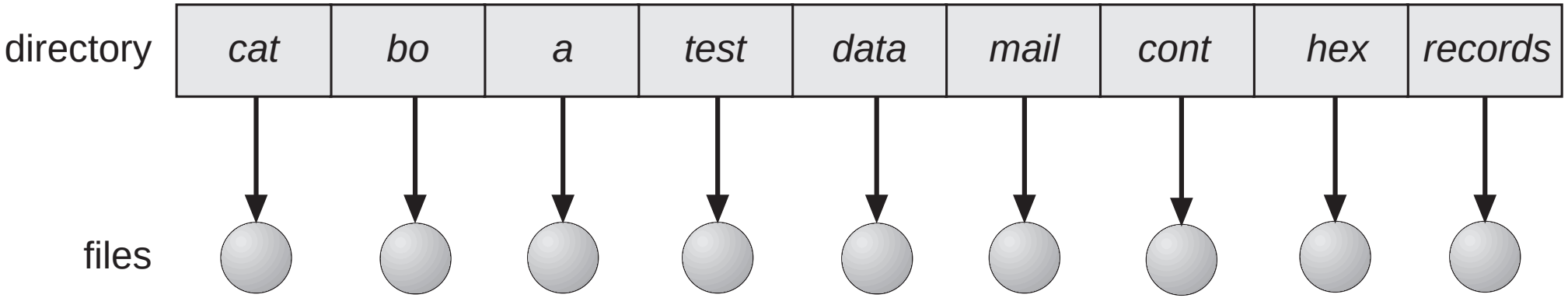
Organizzazione logica delle directory

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

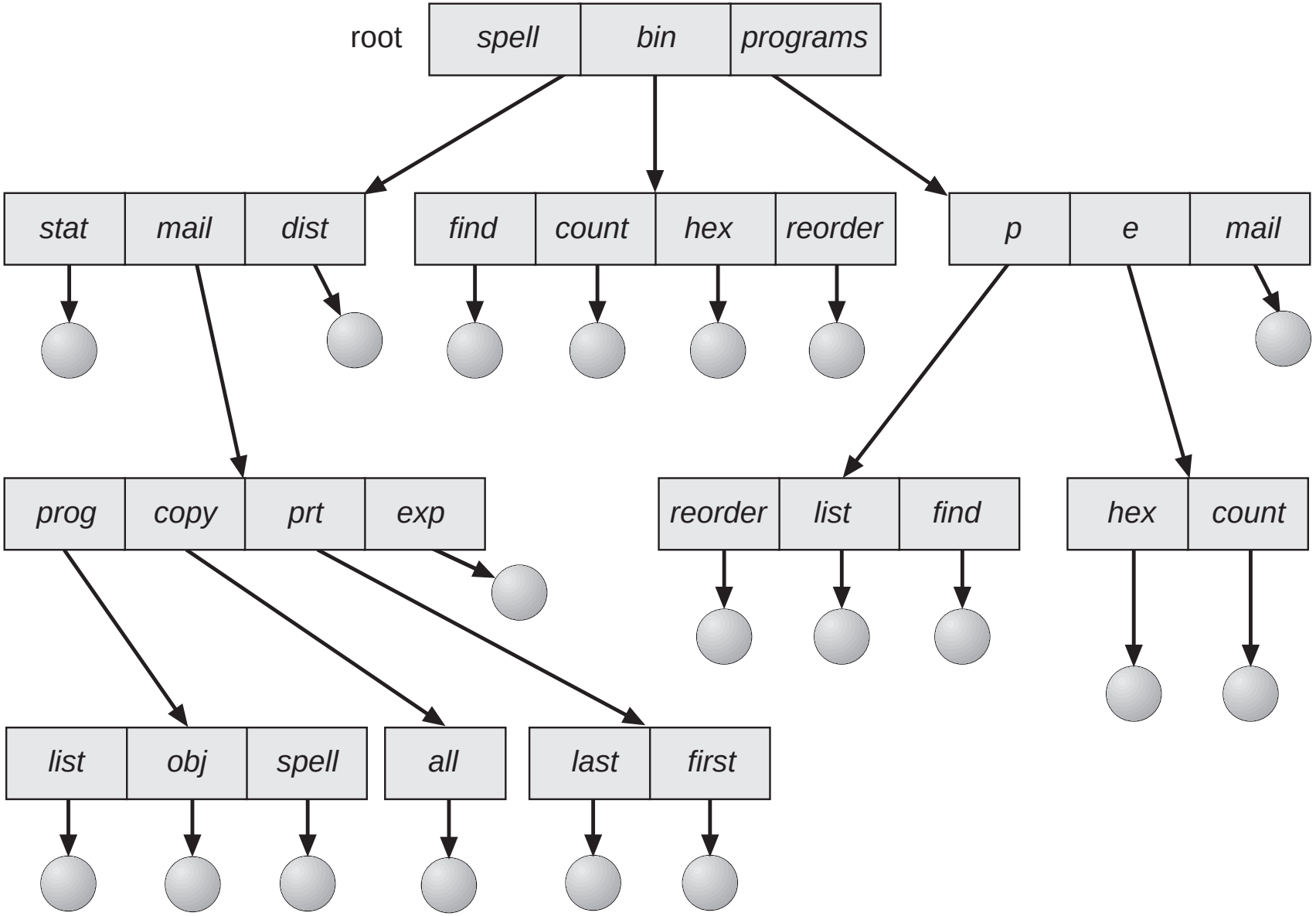


- Problema di raggruppamento e denominazione

- Obsoleta

- Variante: a due livelli (una directory per ogni utente)

Tipi di directory: ad albero



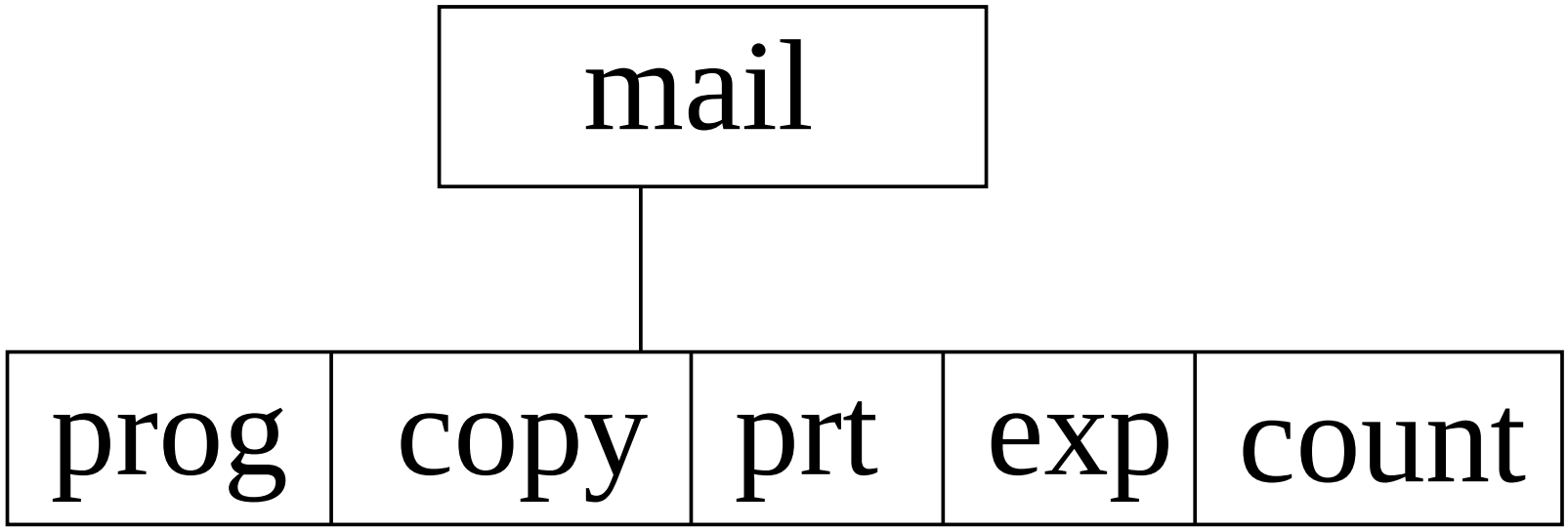
Directory ad albero (cont.)

- 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

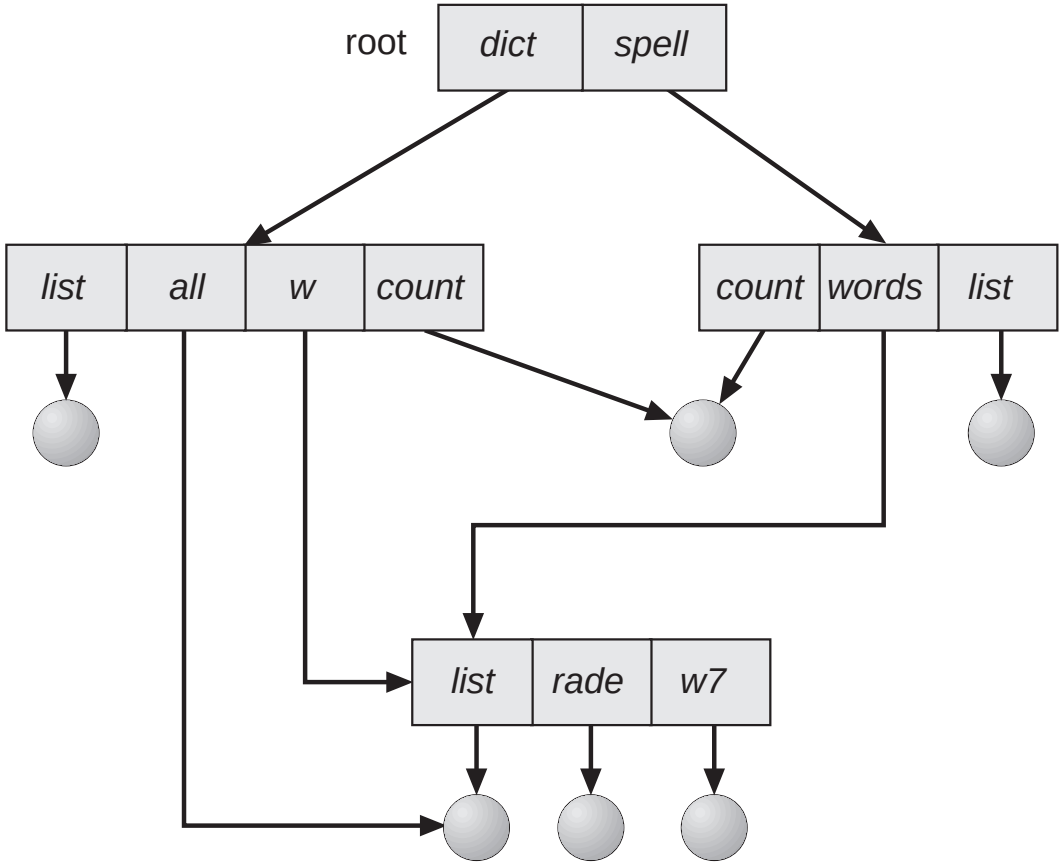


- Cancellando `mail` si cancella l'intero sottoalbero

Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory

Due nomi differenti per lo stesso file (aliasing)



Problemi con directory DAG

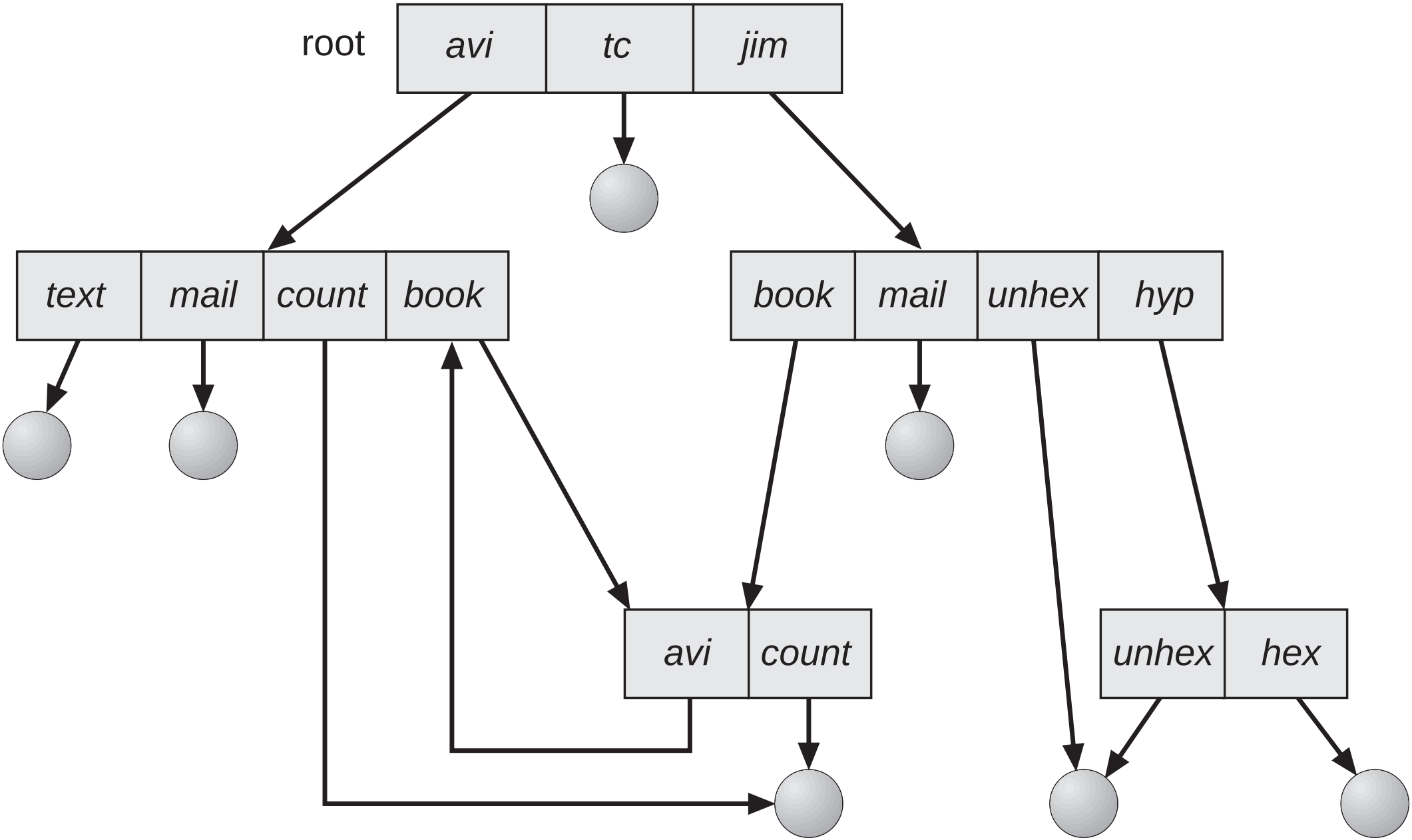
- Possibilità di puntatori "dangling".

- Soluzioni

  – Puntatori all'indietro, per cancellare tutti i riferimenti ad un file da rimuovere. Problematici perché la dimensione dei record nelle directory diventa variabile (dipende dal numero di riferimenti).

  – Contatori di riferimenti per ogni file (UNIX)

Directory a grafo



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 (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

object / domain	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.

  - *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 access control list.

- 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 user identifier (UID) e group identifier (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

- 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

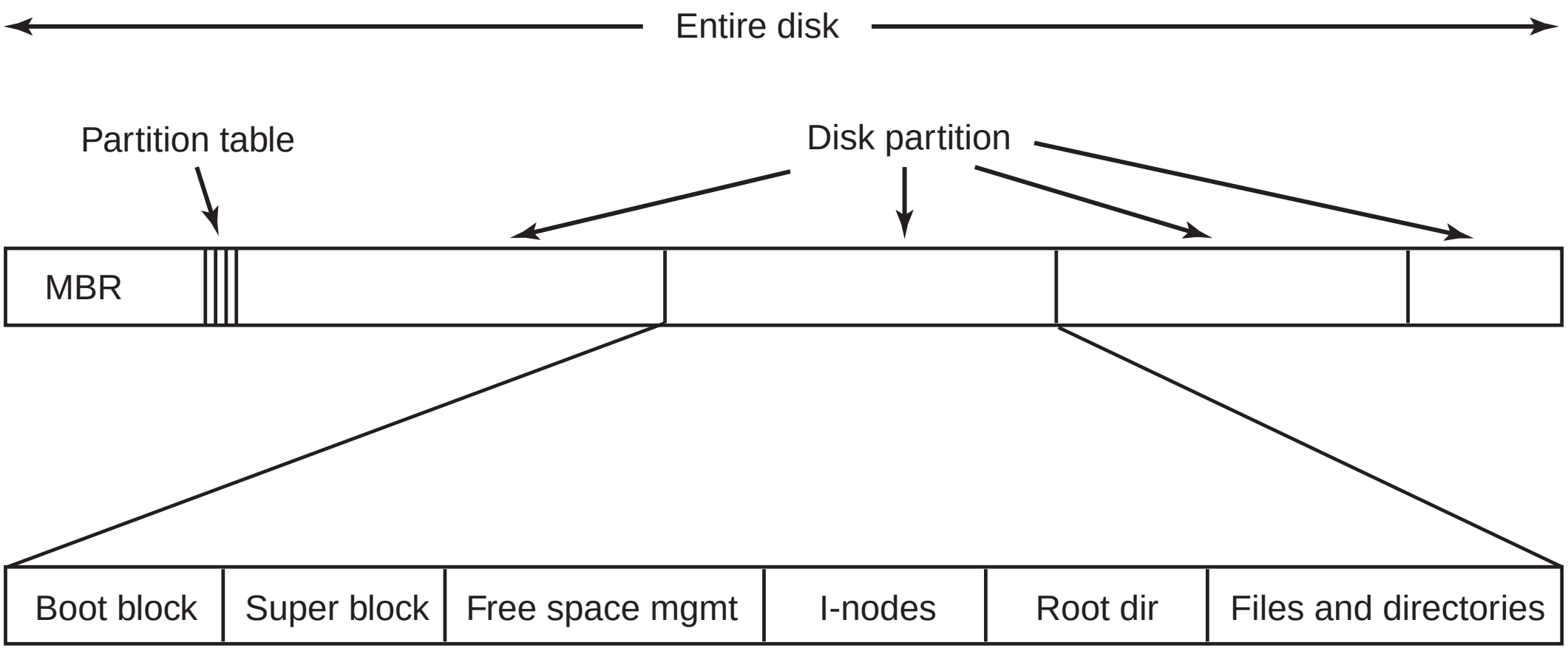
Implementazione del File System

- Il supporto utilizzato più frequentemente per la memorizzazione persistente di dati e' il disco

- Lo spazio disco viene solitamente suddiviso in *partizioni* e *blocchi* (tipicamente 512 byte)

- L'implementazione del file system deve preoccuparsi di come allocare i blocchi del disco, come organizzare i dati riepilogativi delle directory, e cosìvia.

Esempio di layout di un disco fisico



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.

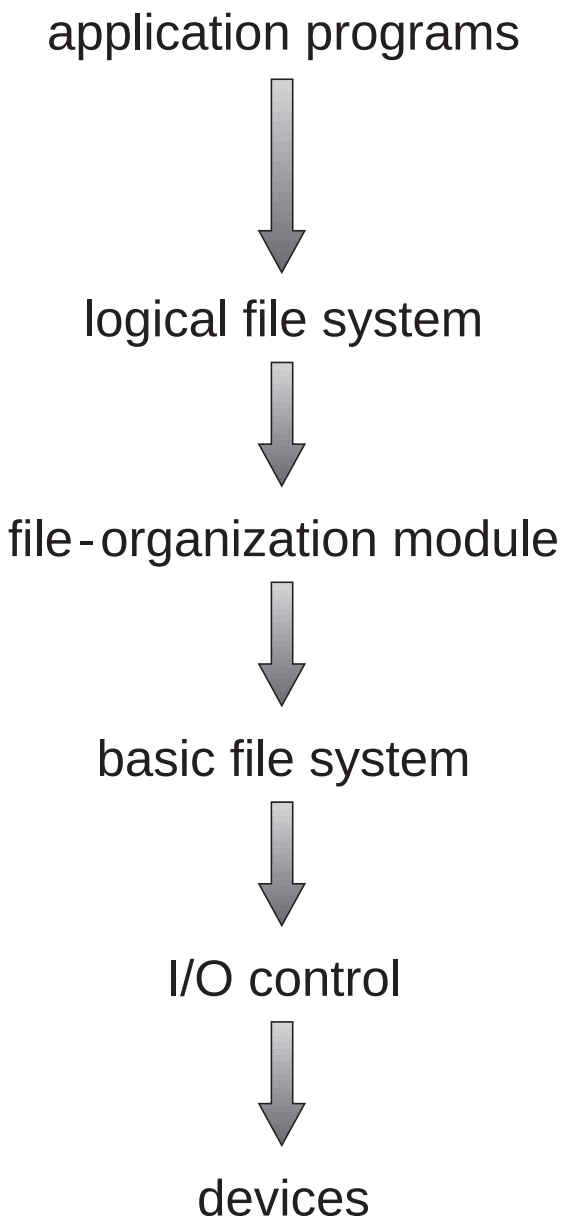


Tabella dei file aperti

- 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)
  - 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):

- Se LA=indirizzo logico

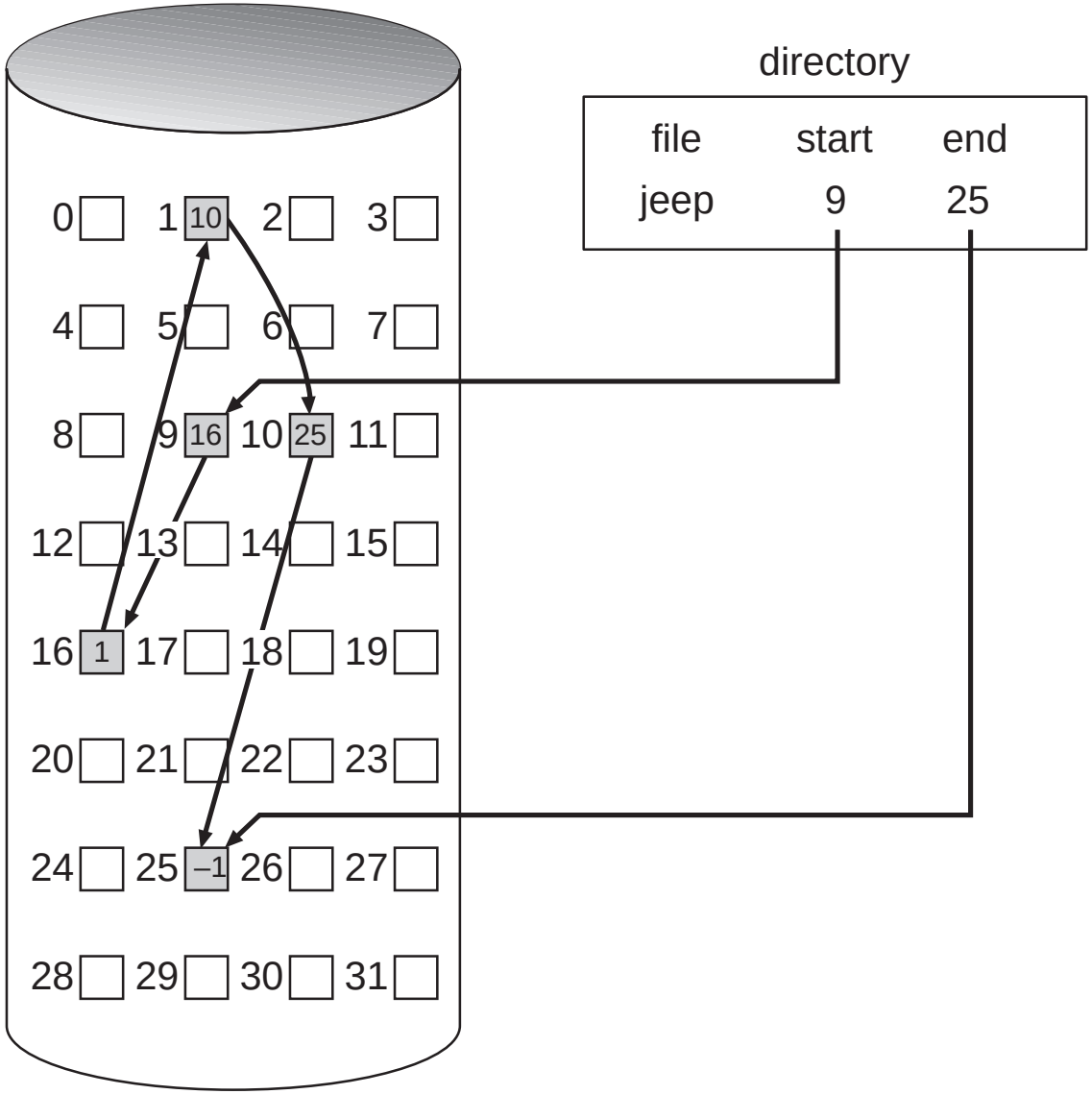
$$\mathrm{LA}/512 \begin{cases} \mathrm{Q} \\ \mathrm{R} \end{cases}$$

  e Q=quoziente, R=resto allora

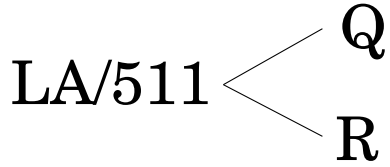
  - Il blocco da accedere = Q + blocco di partenza
  - Offset all'interno del blocco = R

Allocazione concatenata

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



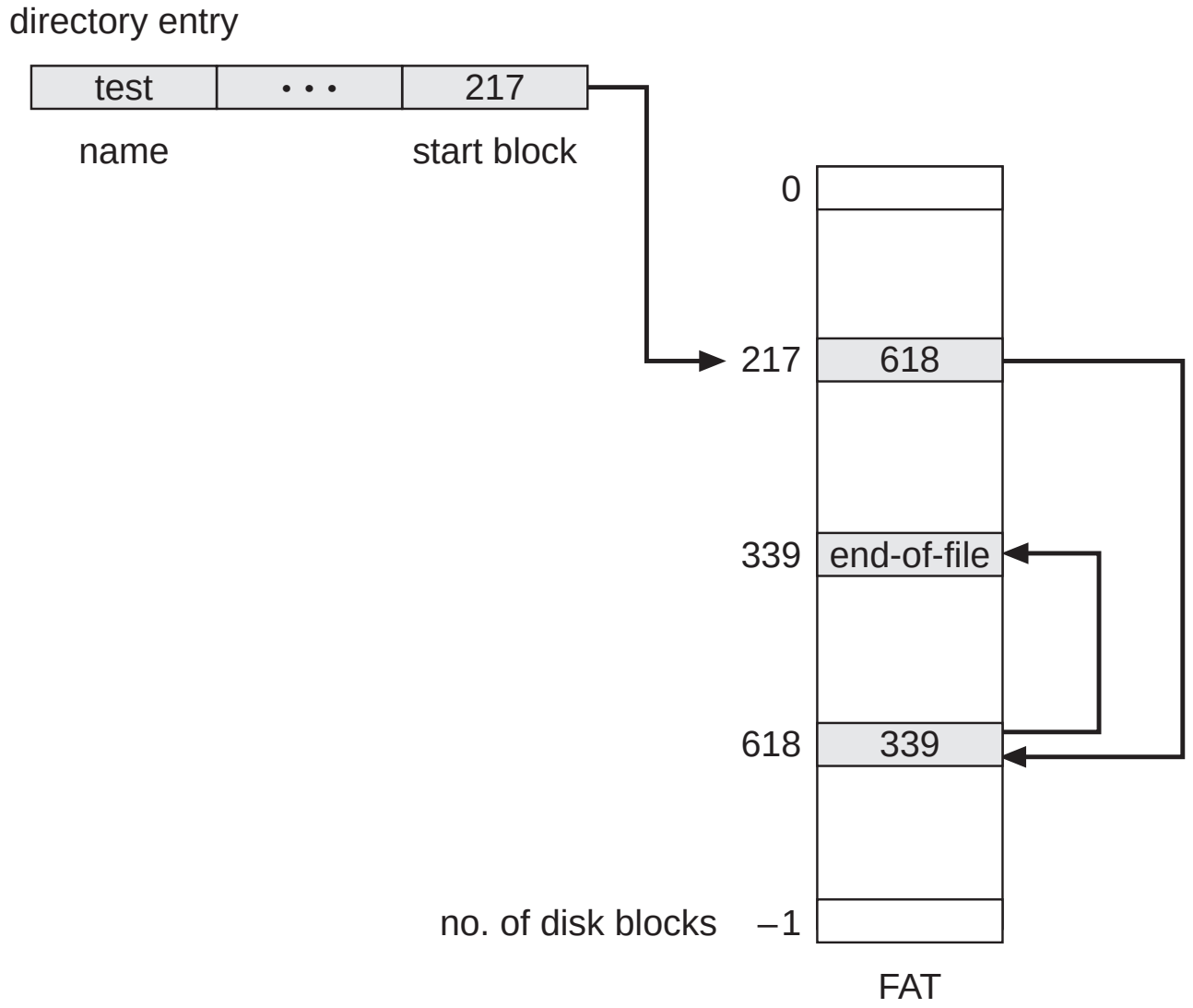
- Allocazione su richiesta; i blocchi vengono semplicemente collegati alla fine del file
- Semplice: basta sapere l'indirizzo del primo blocco
- Non c'è frammentazione esterna
- Bisogna gestire i blocchi liberi
- Non supporta l'accesso diretto
- Traduzione indirizzo logico (1 byte per il puntatore):



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

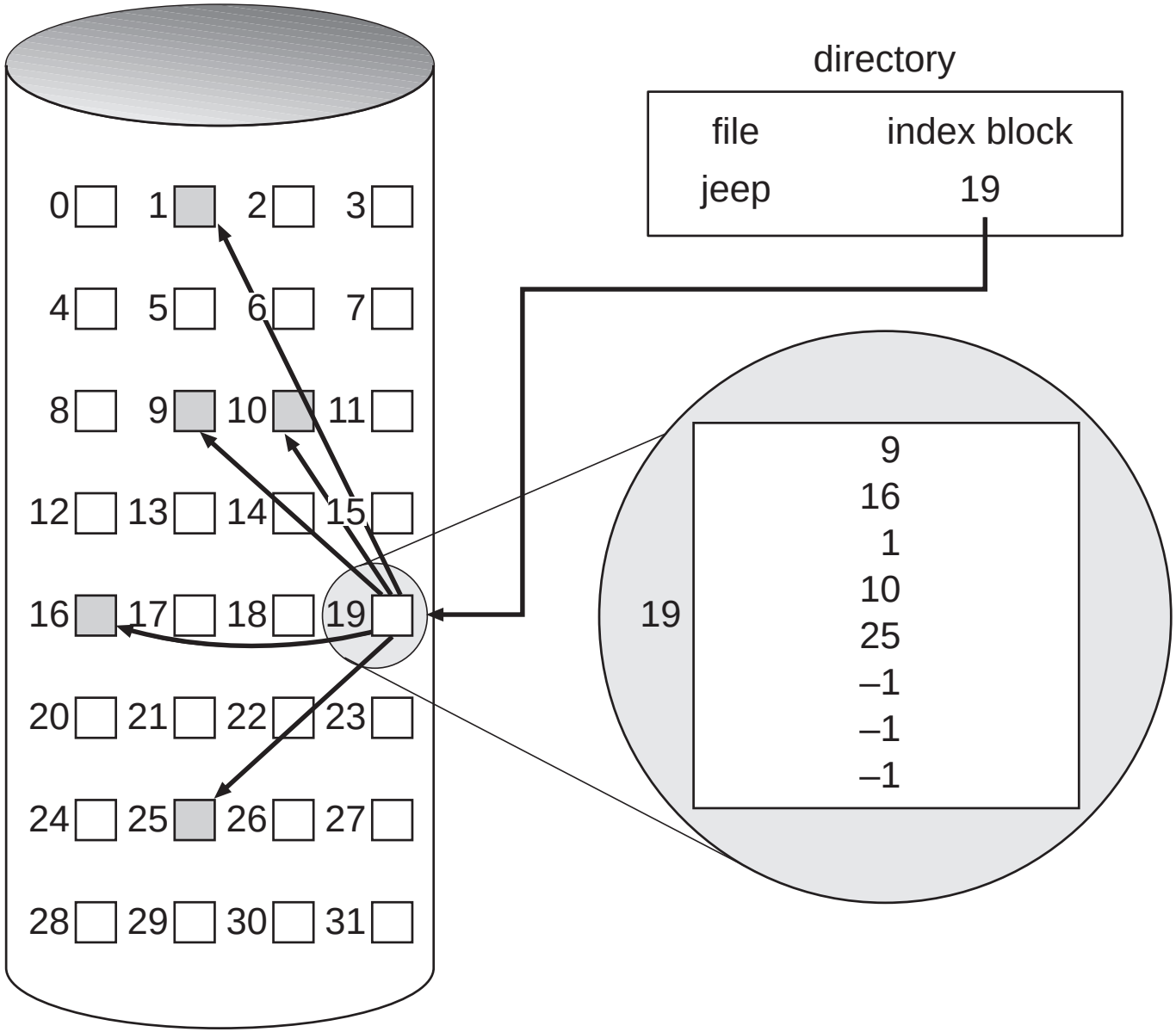
Allocazione concatenata (cont.)

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

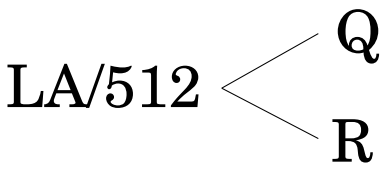


Allocazione indicizzata

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



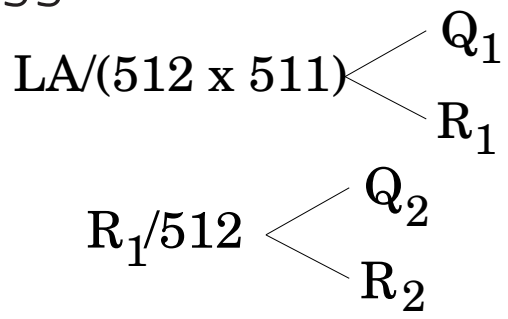
- Supporta accesso random
- Allocazione dinamica senza frammentazione esterna
- Traduzione: file di max 256K word e blocchi di 512 word: serve 1 blocco per l'indice (512x512=262144=256K)



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

Allocazione indicizzata (cont.)

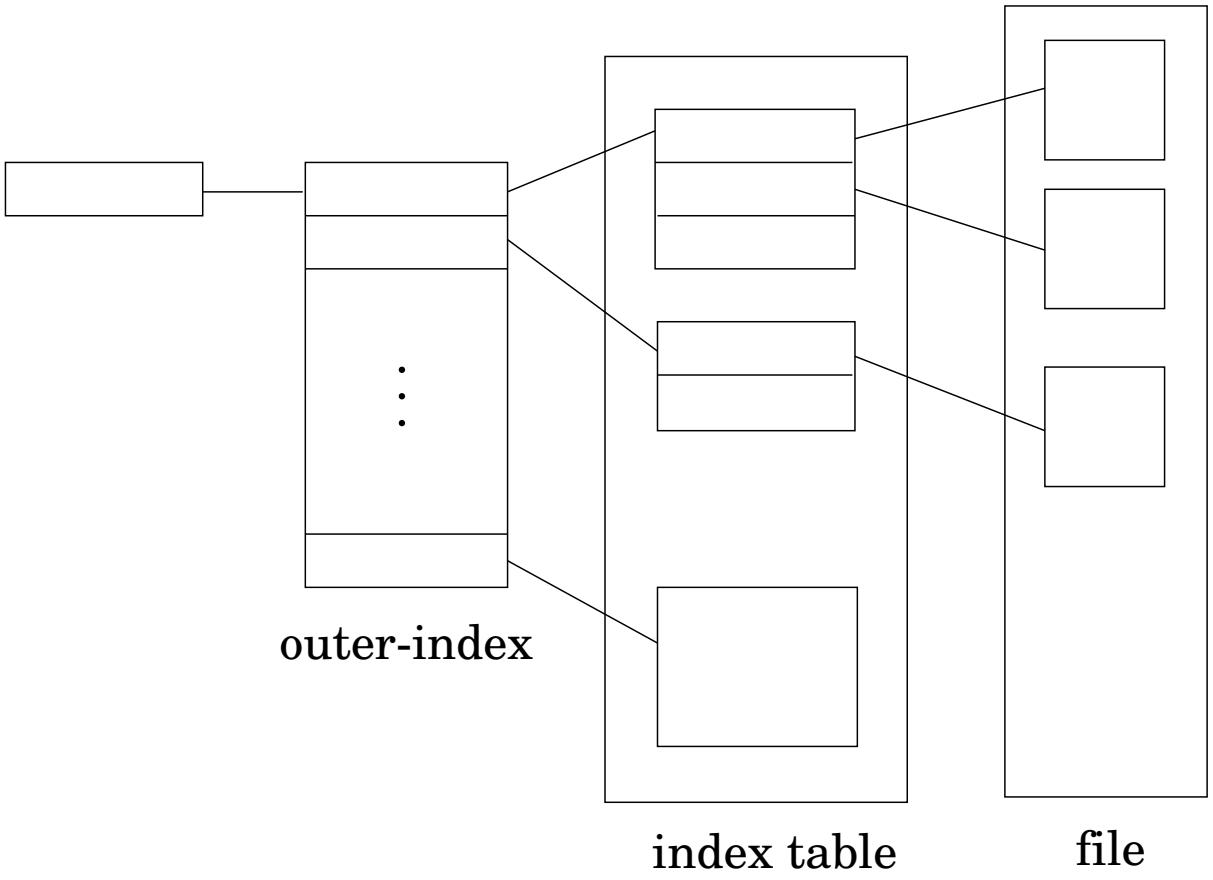
- Problema: come implementare il blocco indice
  - è una struttura supplementare: overhead $\Rightarrow$ meglio piccolo
  - dobbiamo supportare anche file di grandi dimensioni $\Rightarrow$ meglio grande
- Indice concatenato: l'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza,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



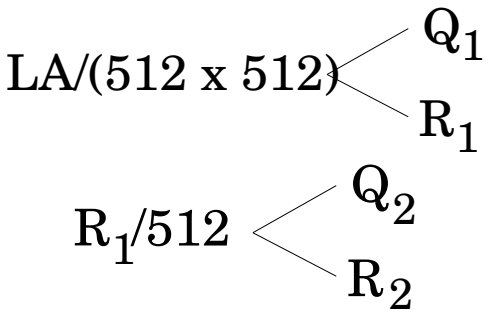
  - Q_1 = blocco dell'indice da accedere
  - Q_2 = offset all'interno del blocco dell'indice
  - R_2 = offset all'interno del blocco del file

Allocazione indicizzata (cont.)

Indice a due (o più) livelli.



- 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 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, modifica e mod. dell'inode

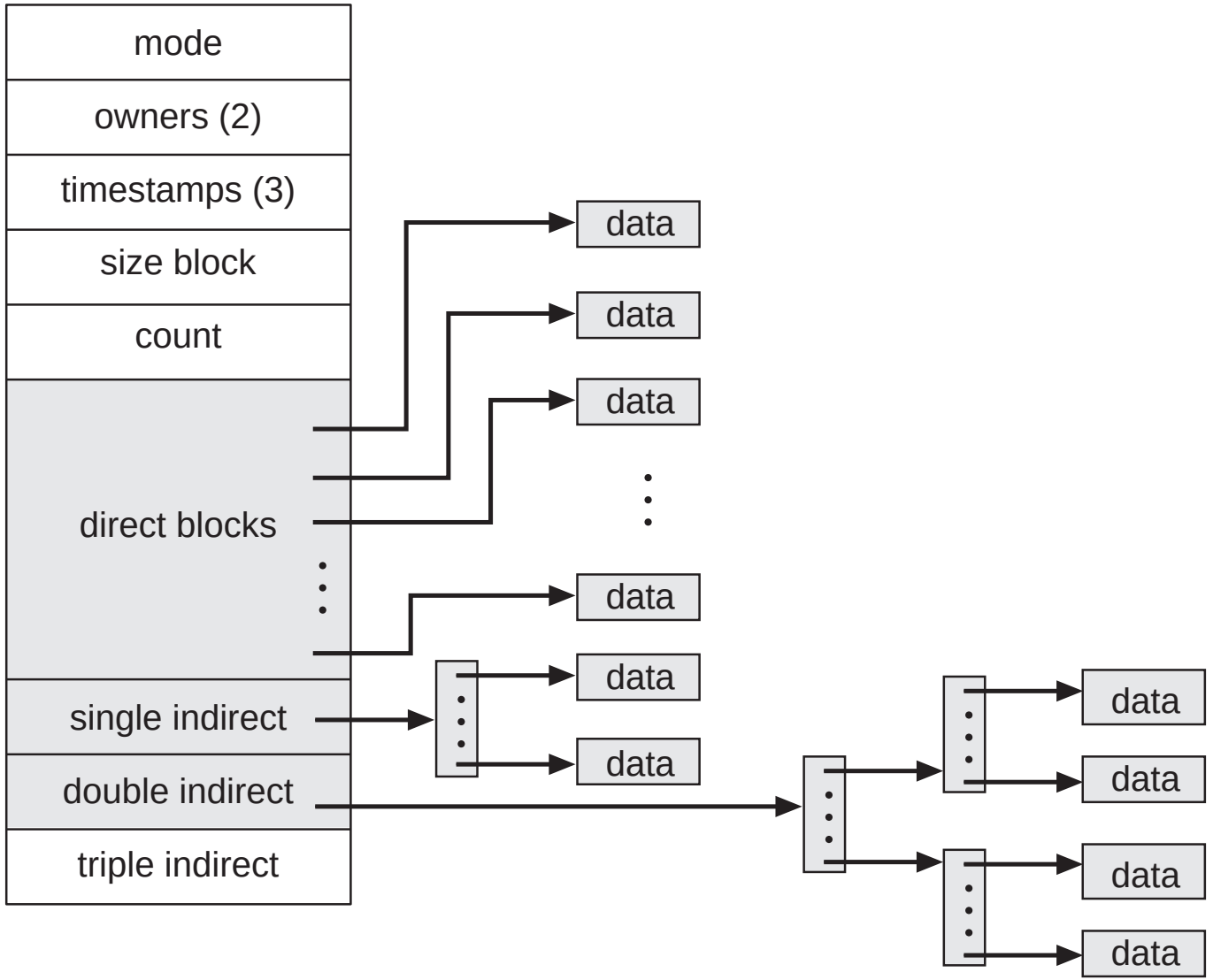
  Numero di link hard che puntano a questo inode

  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!)


mode
owners (2)
timestamps (3)
size block
count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

Inodes (cont.)

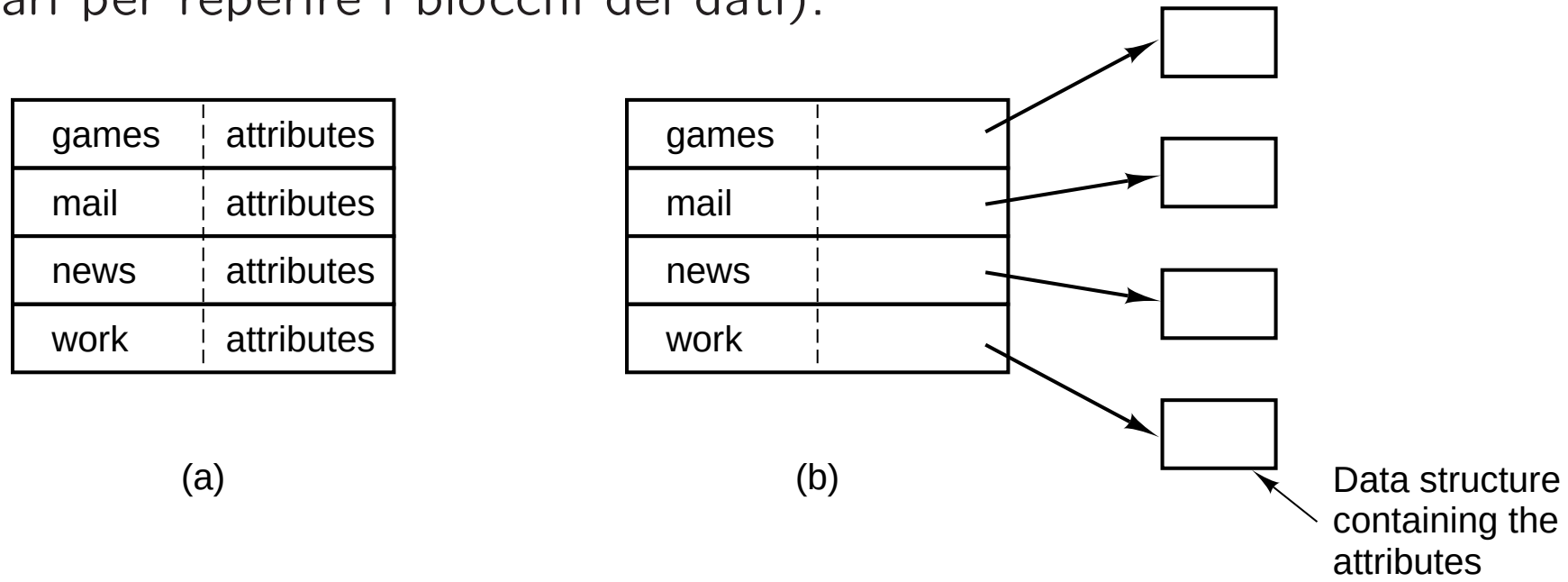
- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte (cioè indice=1 blocco ha 1024 riferimenti a blocchi)

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

  molto oltre le capacità dei sistemi a 32 bit.

Implementazione delle directory

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



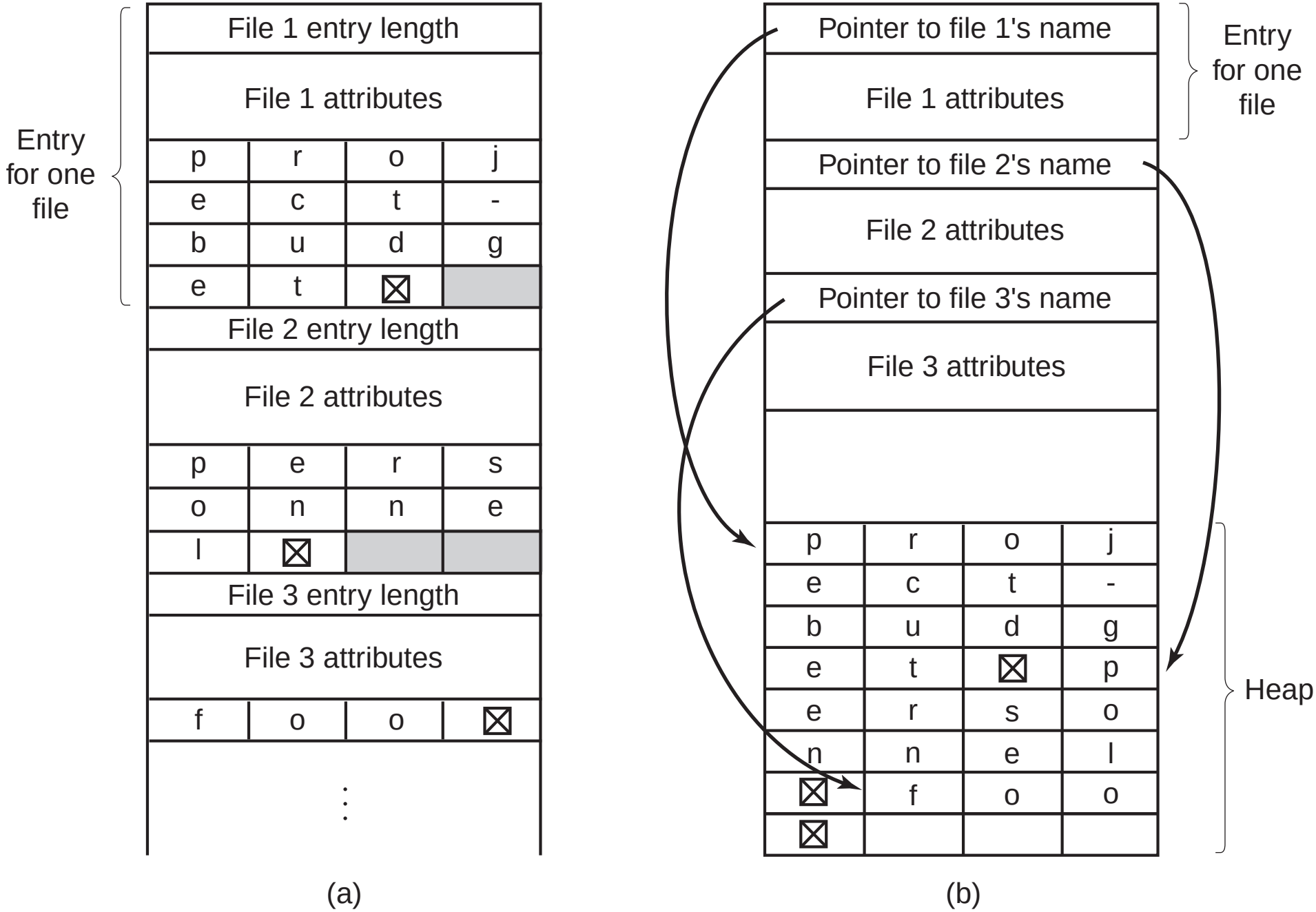
a) Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows)

b) Gli attributi risiedono in strutture esterne (eg. inode dei file), e nelle directory ci sono solo i puntatori a tali strutture (UNIX)

Gestione dei nomi

- Fino ad ora abbiamo supposto che i nomi dei file siano brevi e di lughezza fissata
- In effetti in MS-DOS i nomi erano di al più 8 caratteri + 3 di estensione
- I file system moderni supportano tuttavia nomi di file di lunghezza variabile
- Come si puo implementare?
  - Si fissa una lunghezza massima (e.g. 255) riservando sempre tutto lo spazio (elementi di una directory sempre uguali)
  - Ogni elemento di una directory contiene la sua lunghezza, attributi, e il nome del file: problema della frammentazione
  - Ogni elemento di una directory contiene un puntatore al nome del file e gli atttributi del file; tutti i nomi vengono memorizzati insieme in un *heap* alla fine della directory

Dimensioni variabili e 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

File condivisi

- In file system tipo Unix due directory diverse (e quindi due utenti diversi) possono puntare allo stesso file
  - Collegamento (hard link): i file mantengono un contatore per i riferimenti multipli (ad es. nell'i-node) in modo da evitare puntatori *dangling* (cioe' puntatori a file che non esistono più)
  - Collegamento simbolico (symbolic link): si crea un nuovo file di tipo speciale che contiene il path name del file da collegare; non crea problemi in cancellazione
    * Se si cancella il link simbolico non viene modificato il file
    * Se si cancella il file, chi prova ad usare il link otterrà un errore

Gestione dello spazio libero

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

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

  0101110101010111111011000001010000000101101010111100

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

- Comodo per operazioni assembler di manipolazione dei bit

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.

Affidabilità del file system

Perdere un file system, o parte di esso, è spesso un danno irreparabile e molto costoso.

- 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.

Come funziona il dump logico

- Il dump logico riguarda una particolare parte di file system (un suo sottoalbero) lo scopo e' riversare ad es. su nastro tutti i file modificati dall'ultimo dump. Occorre tuttavia mantenere su nastro tutte le informazioni sulla struttura dell'albero (cioe' anche directory e file non modificati)

- L'algoritmo utilizzato ad. es. in Unix effettua un *dump incrementale*

- Inizialmente si effettua un dump completo di tutto il file system

- Successivamente si salvano su nastro solo le modifiche dall'ultimo dump logico

Dump incrementale in Unix

- L'algoritmo per dump incrementale consiste di 4 fasi e utilizza una mappa di bit indicizzata sui numeri di i-node del file system considerato

  - I fase: si visita l'albero e si marcano tutti gli i-node associati a file modificati e si marcano anche tutte le directory;

  - II fase: si visita nuovamente l'albero e si smarcano tutte le directory che *non* contengono (ad una qualsiasi profondità nel corrispondente sottoalbero) file modificati

    Nota: i-node marcato $\rightarrow$ da salvare su nastro

  - III fase: si analizzano tutti gli i-node in ordine di numero e si scaricano su nastro tutte le directory *marcate* insieme ai loro attributi (proprietario, ecc)

  - IV fase: si analizzano tutti gli i-node in ordine di numero e si scaricano su nastro tutte i file *marcati* insieme ai loro attributi.

Dump incrementale in Unix

- Ad es. FS: /home/giorgio/Lucidi /home/giorgio/Articoli e Lucidi contiene un file slide modificato dall'ultimo dump
- Nella I fase: marchiamo /home /home/giorgio /home/giorgio/Lucidi /home/gi e /home/giorgio/Lucidi/slide
- Nella II fase: smarchiamo solo /home/giorgio/Articoli: /home /home/giorgio /home/giorgio/Lucidi contengono file modificati
- Nella III fase salviamo su nastro le informazioni su /home /home/giorgio /home/giorgio/Lucidi
- Nella IV fase salviamo su nastro /home/giorgio/Lucidi/slide

Ripristino da dump su nastro

- Per ripristinare un file system da nastro di dump:
  - Si crea un file system vuoto
  - Si ripristina il dump *completo* più recente: le directory compaiono prima su nastro: vengono utilizzare per creare lo scheletro del file system e poi vengono riempite con i file
  - Si procede allo stesso modo con tutti i dump incrementali fatti dopo il dump completo

Consistenza del file system

- In seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contradditorie.

- Si utilizzano dei programmi di controllo della consistenza (*scandisk*, *fsck*): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze.

- Programmi come *fsck* effettuano controlli sia per *blocchi* che per *file*

fsck: controllo per blocchi

- Nel controllo per *blocchi* si costruiscono due tabelle indicizzate sui blocco fisici

- La prima tabella tiene traccia di quante volte un blocco è presente in un file

- La seconda tabella tiene traccia di quante spesso un blocco è presente nella lista (o nella mappa di bit) dei blocchi liberi (o nella mappa di bit)

- *fsck* esegue due passi:

  – prima scandisce tutti gli i-node e recupera i numeri dei suoi blocchi: per ogni blocco incrementa il contatore nella prima tabella;

  – poi scandisce la lista dei blocchi liberi: per ogni blocco incrementa il contatore nella seconda tabella.

fsck: recovery di blocchi

- Se il file system è coerente ogni blocco avrà un 1 nella prima o nella seconda tabella

- Se un blocco non compare in nessuna delle due tabelle (ha contatore $= 0$ in entrambe): viene aggiunto alla lista libera

- Se un blocco compare più volte nella lista libera (ha contatore > 1 nella seconda tabella): si ricostruisce la lista

- Se un blocco compare più volte nello stesso file o in file diversi (ha contatore > 1 nella prima tabella): si allocano dei nuovi blocchi si fa una copia del contenuto del blocco inconsistente e si associano tale copie ai file a cui apparteneve il blocco in questione; inoltre si manda un messaggio di errore all'utente

- Se un blocco compare sia in un file che nella lista libera: si ricostruisce la lista rimuovendo il blocco dalla lista

fsck: controllo per file

- Nel controllo per *file* si utilizza una tabella indicizzata sui file che tiene traccia del numero di riferimenti al file

- *fsck* visita il file system e incrementa il contatore di ogni file che incontra in una directory durante la visita
  (nota: un file con link fisico puo' appartenere a più directory)

- Al termine della visita si confronta il contatore C associate ad un file nella tabella con il numero di riferimenti R contenuto nel suo i-node

fsck: recovery per file

- $C =$ valore del contatore associato al file f nella tabella costruita da *fsck*

- $R =$ valore del contatore di riferimenti nell'i-node del file f (modificato ogni qualvolta si crea un link fisico)

  - Se $C = R$ il file system è coerente

  - Se $R > C$ si ha un errore non grave ma un potenziale spreco di spazio: se cancello i file dalle directory in questione, R rimane > 0 e quindi l'i-node non viene rimosso.
    Recovery: si assegna R a C in modo che $C = R$.

  - Se $R < C$ si ha un errore grave. Se cancello un file da una delle directory in questione, R potrebbe diventare $= 0$ e quindi il suo i-node rimosso e i blocchi relativi rilasciati anche se esistono altre directory che puntano allo stesso file.
    Recovery: si assegna C ad R in modo che $C = R$.

Efficienza e performance di un file system

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!

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

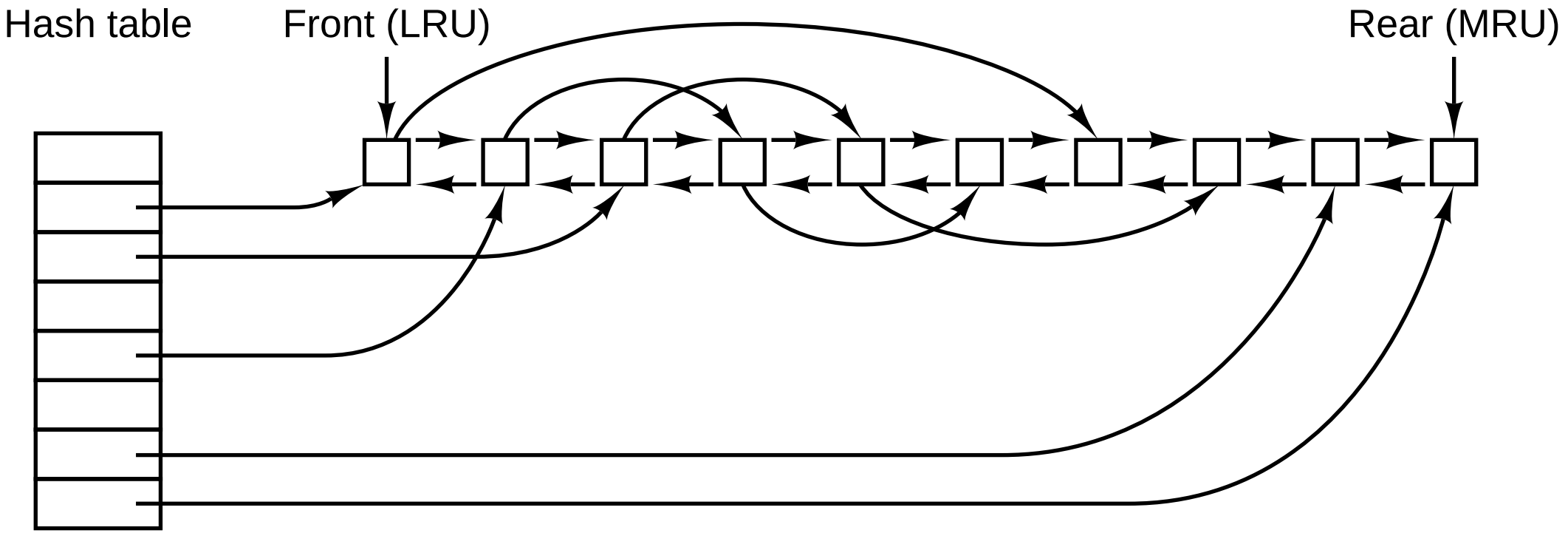
Migliorare le performance: caching

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

- sul controller: usato come *buffer di traccia* per ridurre il tempo di latenza nell'accesso al disco

- (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 (ordinata a seconda del tempo di accesso, primo blocco=usato meno recentemente) con accesso hash



- La coda può essere gestia 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:

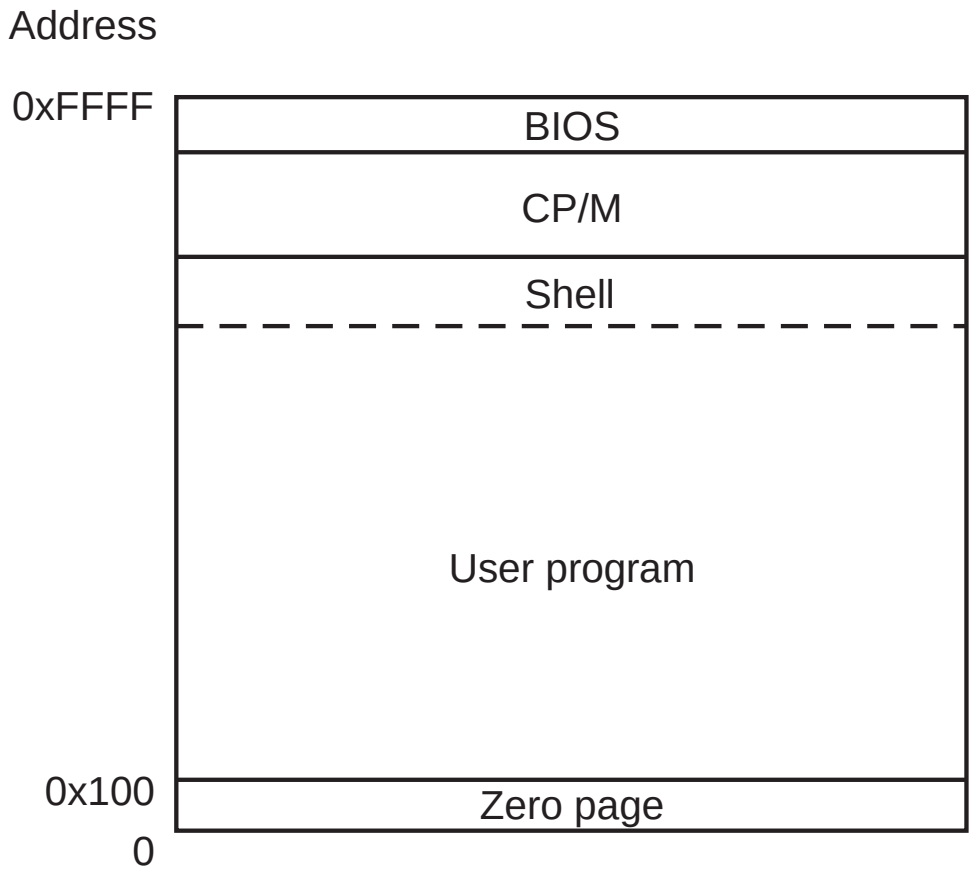
- asincrono: ogni 20-30 secondi (Unix, Windows)
- sincrono: ogni scrittura viene immediatamente trasferita anche al disco (*write-through cache*, DOS).

Esempi di File System

CP/M

- Il sistema CP/M può essere considerato l'antenato di MS-DOS
- CP/M era un sistema operativo per macchine con processori a 8 bit e 4KB di RAM e un singolo floppy disk di 8 pollici con capacità di 180 KB.
- CP/M era scarno e molto compatto e rappresenta un interessante esempio di sistema embedded
- Quando caricato nella RAM:
  - Il BIOS contiene una libreria di 17 chiamate di sistema (interfaccia harware)
  - Il sistema operativo occupa meno di 3584 byte ($<$ 4KB!); la shell occupa 2 KB

– Ultimi 256 byte: vettore interruzioni, buffer per la linea di comando



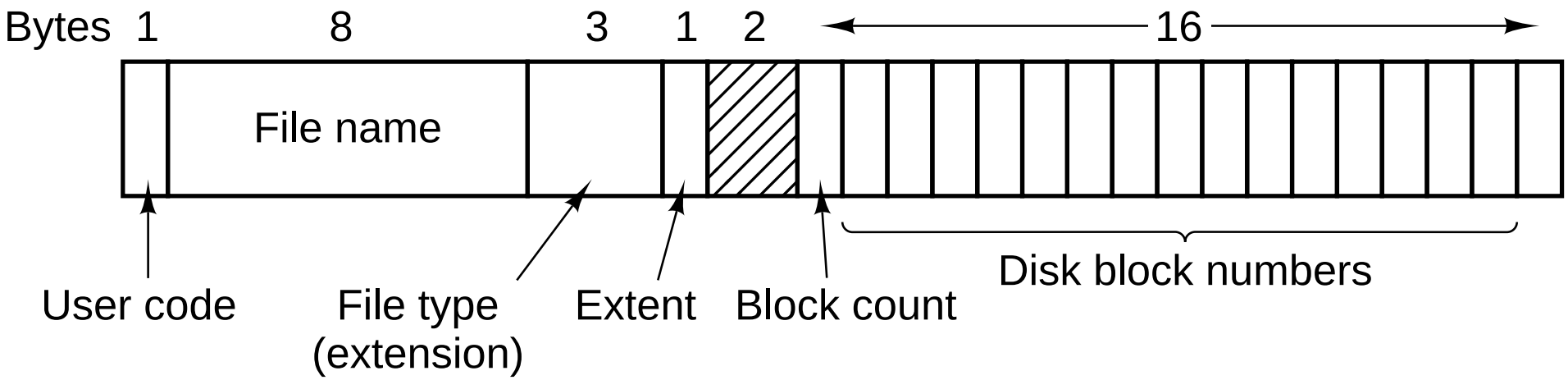
- Comandi vengono copiati nel buffer, poi CP/M cerca il programma da eseguire, lo scrive a partire dall'indirizzo 256 e gli passa il controllo

- Il programma può scrivere sopra la shell se necessario

File System in CP/M

- CP/M ha una sola directory (flat)
- Gli utenti si collegavano uno alla volta: i file avevano informazioni sul nome del proprietario
- Dopo l'avvio del sistema CP/M legge la directory e calcola una mappa di bit (di 23 byte per un disco da 180KB) per i blocchi liberi
- La mappa viene tenuta in RAM e buttata via quando si spegne la macchina

Elementi di directory in CP/M

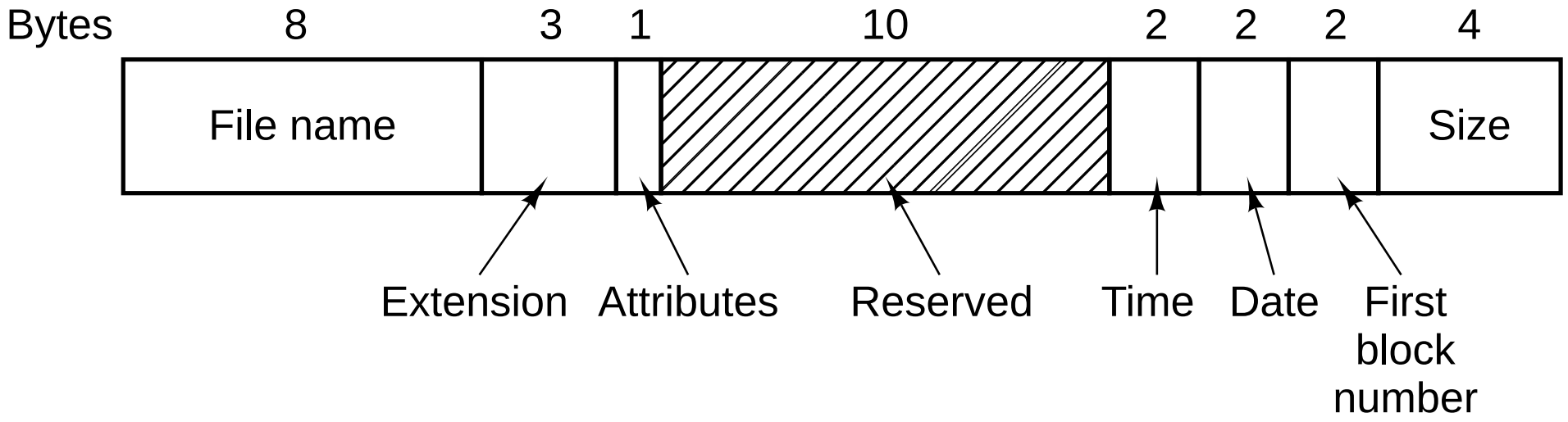


- Lunghezza del nome fissa: 8 caratteri + 3 di estensione
- Extent: serve per file con più di 16 blocchii: si possono usare più elementi di directory per lo stesso file, extent mantiene l'ordine con cui leggere i blocchi
- Contatore blocchi: numero complessivo di blocchi (non dimensione del file: serve EOF nell'ultimo blocco)
- Blocchi: dimensione da 1KB

File System in MS-DOS

- MS-DOS è una evoluzione di CP/M
- Le directory hanno struttura ad albero e non flat
- Si usa la File Allocation Table (FAT) per la mappa file blocchi e la gestione dei blocchi liberi (sono marcati in maniera speciale)
- Non è tuttavia multi utente

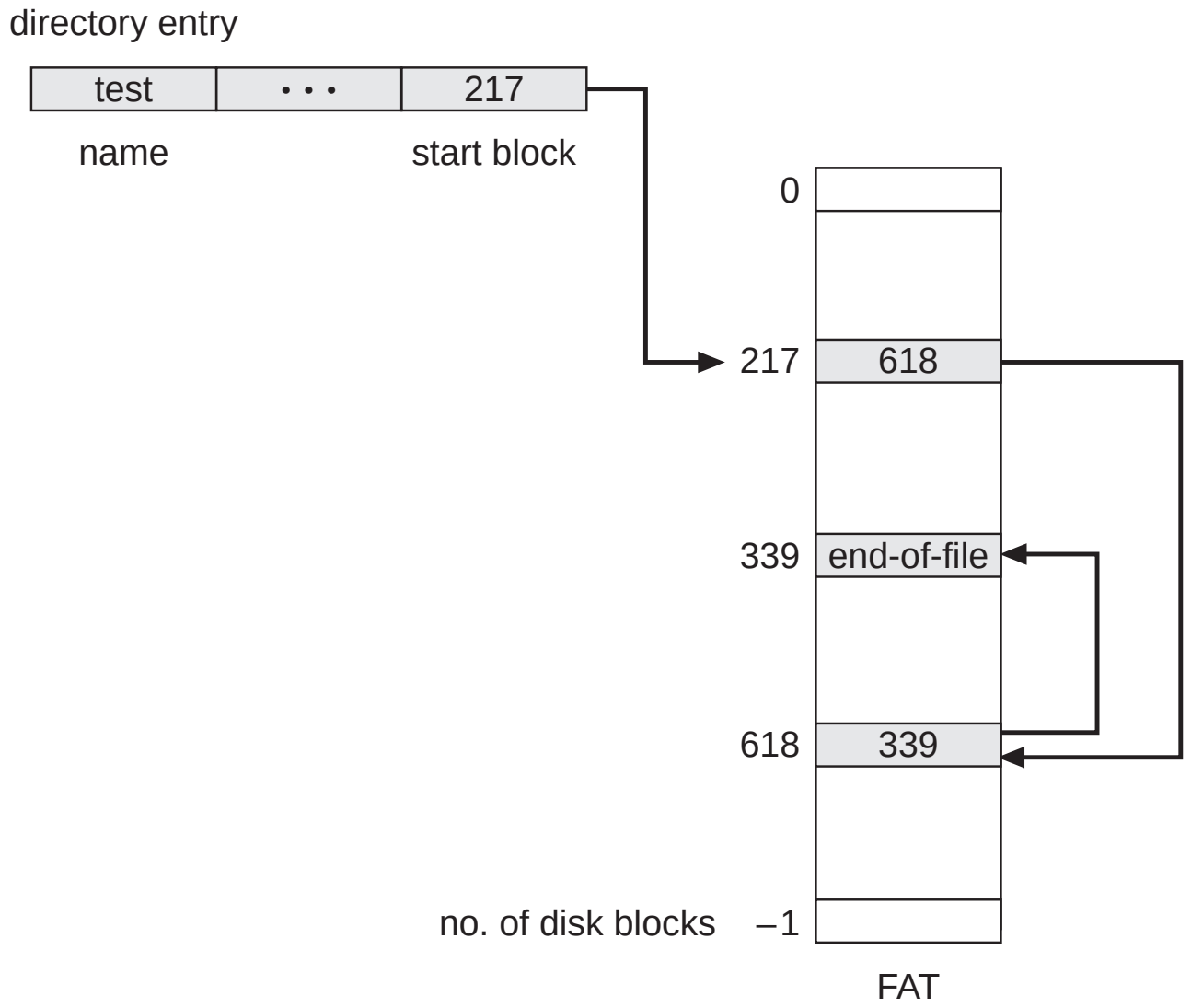
Directory in MS-DOS



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

File-allocation table (FAT)

Mantiene la linked list in una struttura dedicata, all'inizio di ogni *partizione* del disco



FAT12

- La prima versione di DOS usava una FAT-12 (cioé con indirizzi di disco di 12 bit) con blocchi da 512 byte
- Dimensione max di una partizione: $2^{12}x512$ byte ($2MB$)
- Dimensione della FAT: 4096 elementi da 2 byte
- Ok per floppy ma non per hard disk
- Con blocchi da 4KB si ottenevano partizioni da 16MB (con 4 partizioni: dischi da 64MB)

FAT16

- FAT-16 utilizza indirizzi di disco da 16 bit con blocchi da 2KB a 32KB
- Dimensione max di una partizione: $2GB$
- Dimensione della FAT: 128KB

FAT32

- A partire dalla seconda versione di Windows 95 si utilizzano indirizzi di 28 bit (non 32)

- Partizioni max: $2^{28} \times 32KB (= 2^{15})$ in realtà limitate a $2TB$ $(= 2048TB)$

- Vantaggi rispetto a FAT-16: un disco da 8GB può avere una sola partizione (su FAT 16 deve stare su 4 partizioni)

- Può usare blocchi di dimensione più piccola per coprire uguale spazio disco ($2GB$ FAT-16 deve usare blocchi da $32KB$ mentre FAT-32 può usare blocchi da $4KB$

FAT12, FAT16, FAT32

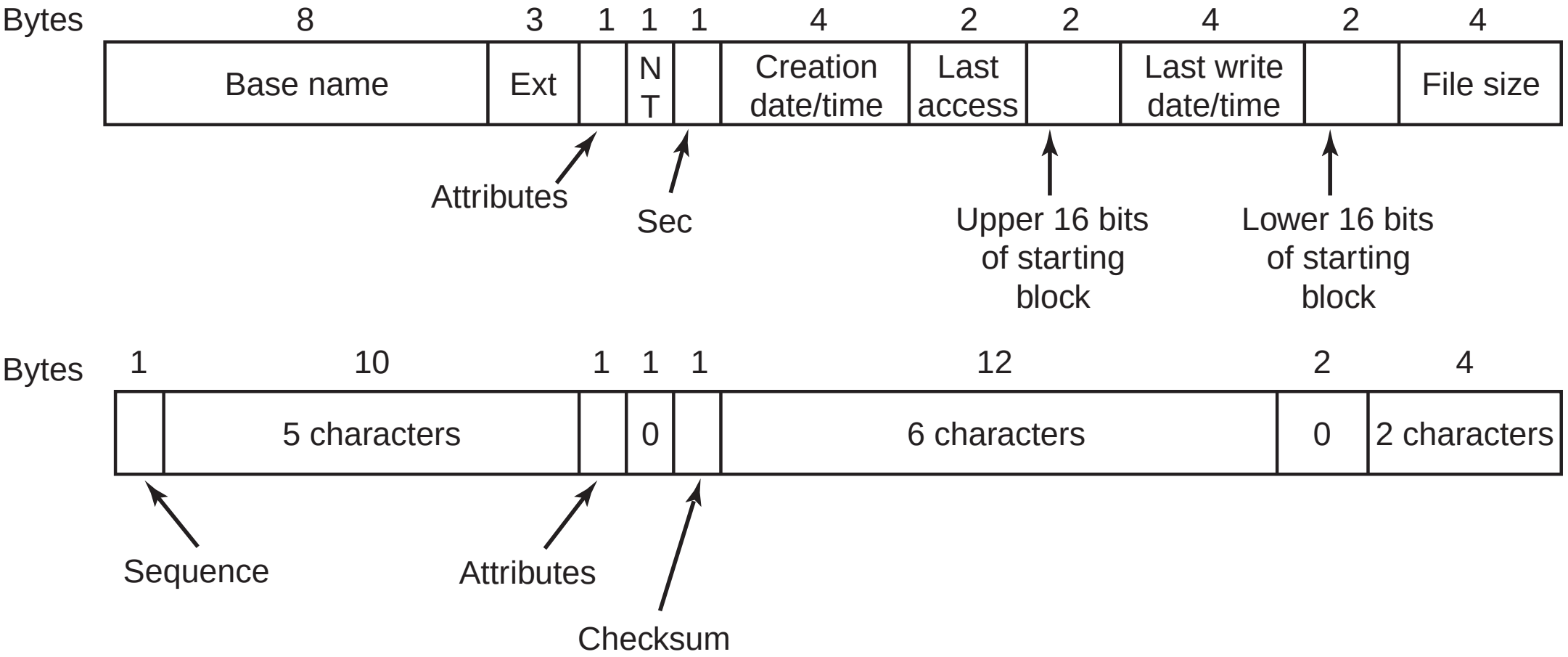
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

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

Il block size è chiamato da Microsoft *cluster size*

Directory in Windows 98

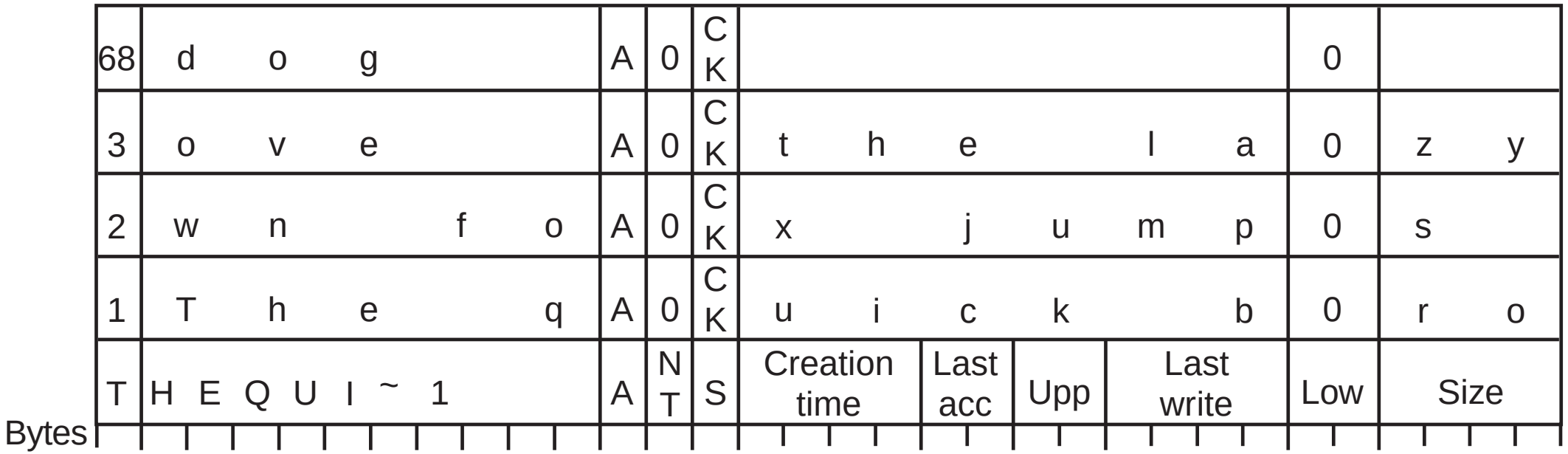
Nomi lunghi ma compatibilità all'indietro con MS-DOS e Windows 3



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

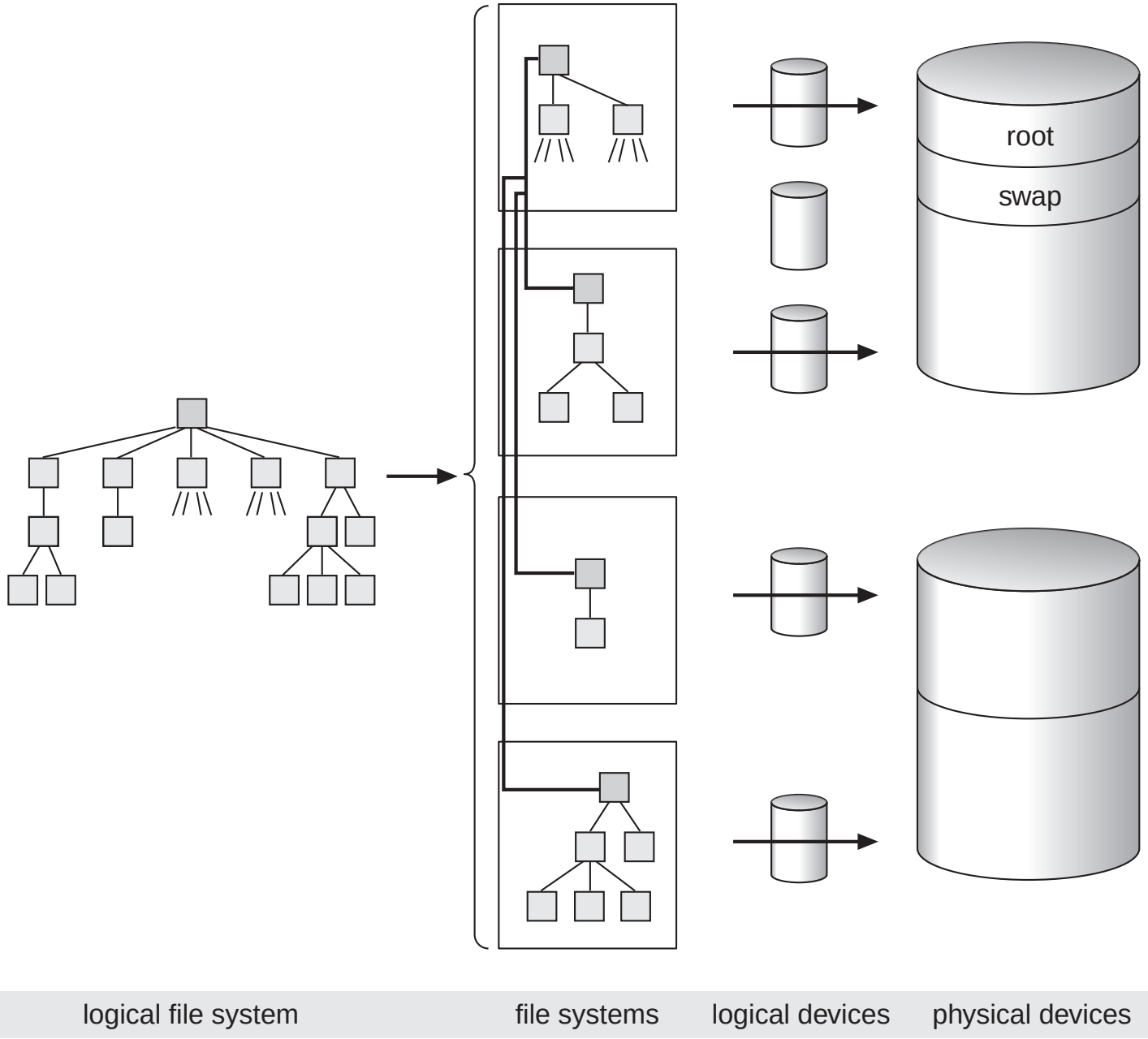
Esempio

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



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, EXT2, EXT3 e anche MSDOS); quelli preferiti sono UFS (Unix File System, aka BSD Fast File System), EXT2 (Extended 2), EFS (Extent File System)

- 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)

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 inote 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)

- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- L'inode contiene puntatori diretti e indiretti (a 1, 2, e 3 livelli)
- 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

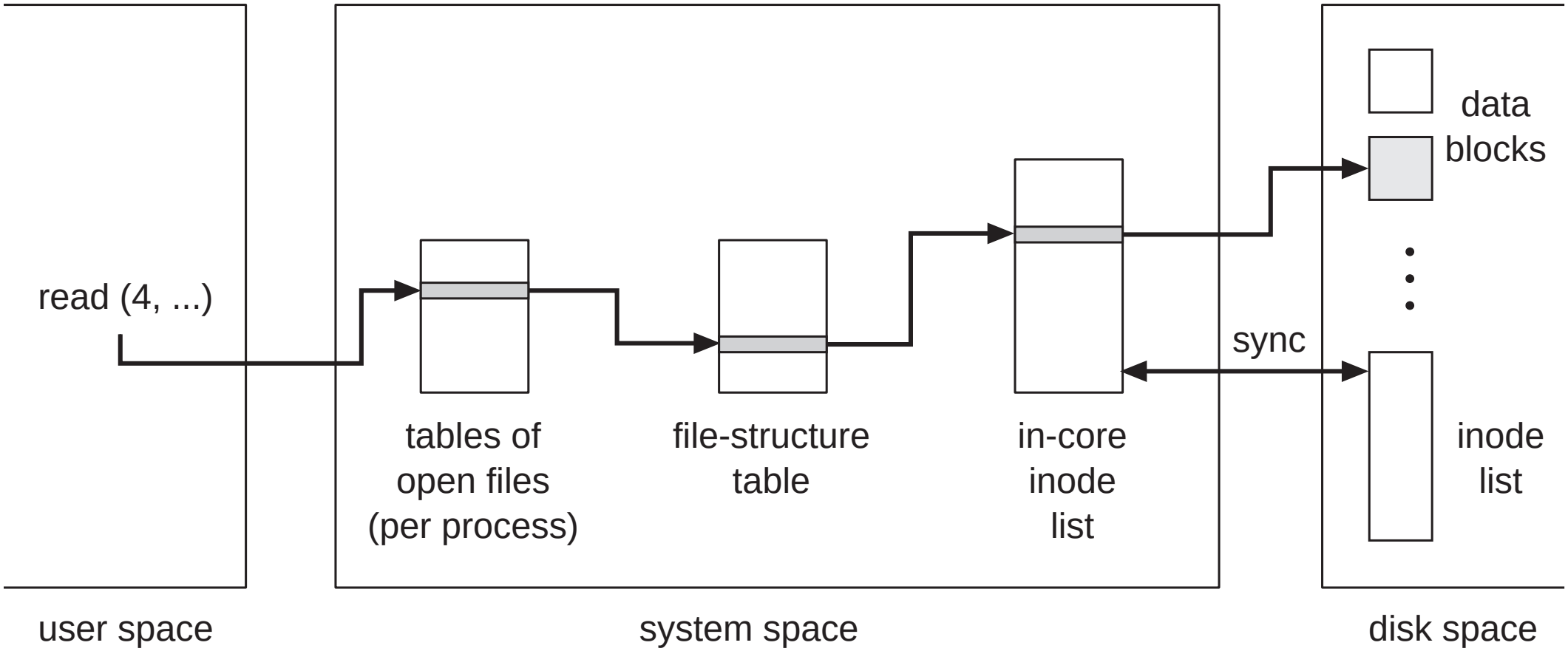
- Le system calls che si riferiscono a file aperti (read, write, close, ...) prendono un *file descriptor* come argomento

- Il file descriptor viene usato dal kernel per entrare in una tabella di file aperti del processo. Risiede nella U-structure.

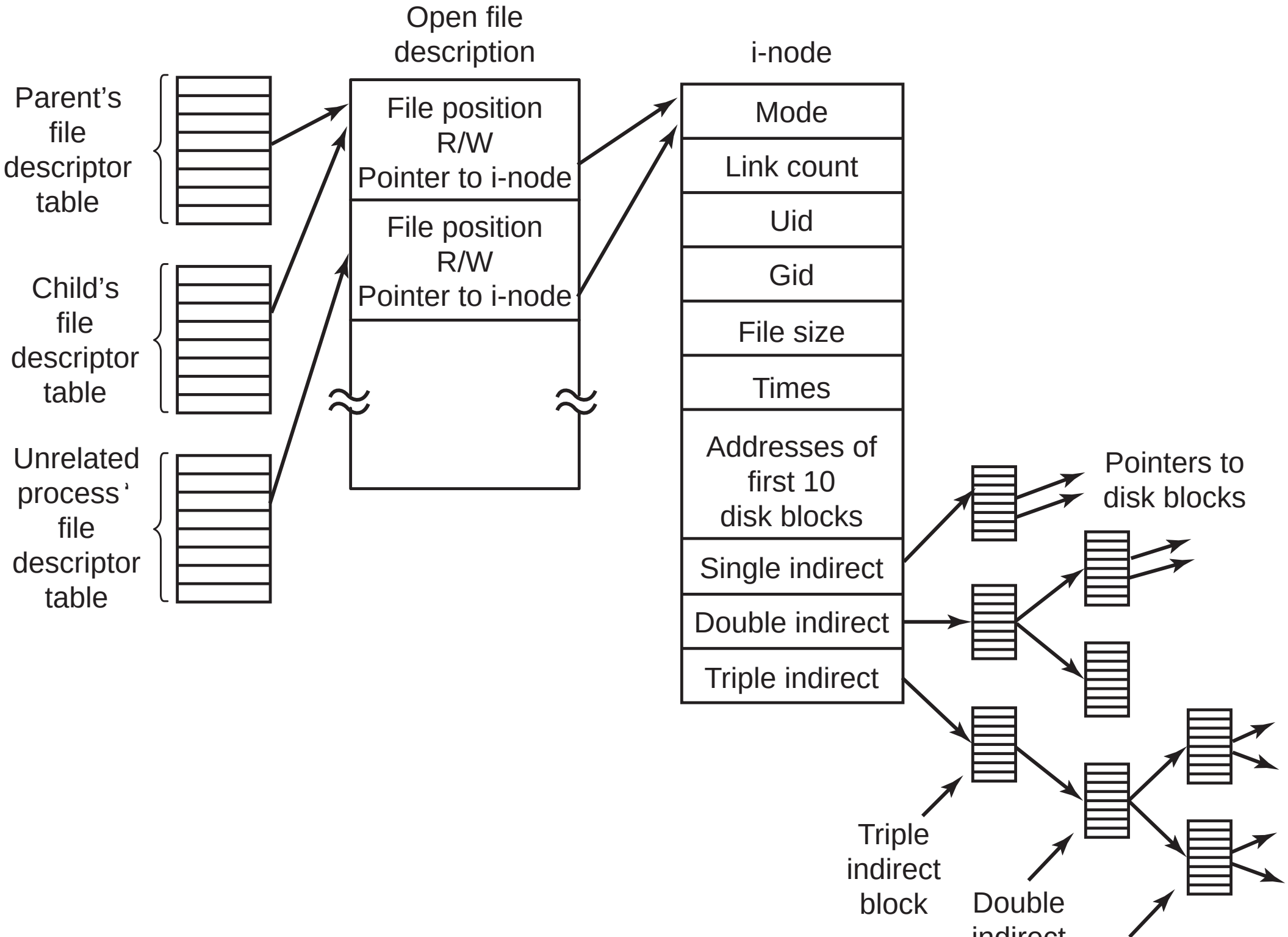
- Ogni entry della tabella contiene un puntatore ad una *file structure*, di sistema. Ogni file structure punta ad un inode (in un'altra lista), e contiene la posizione nel file.

- 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


Open file description
i-node
Parent's file descriptor table
Child's file descriptor table
Unrelated process' file descriptor table
File position
R/W
Pointer to i-node
File position
R/W
Pointer to i-node
Mode
Link count
Uid
Gid
File size
Times
Addresses of first 10 disk blocks
Single indirect
Double indirect
Triple indirect
Pointers to disk blocks
Triple indirect block
Double indirect

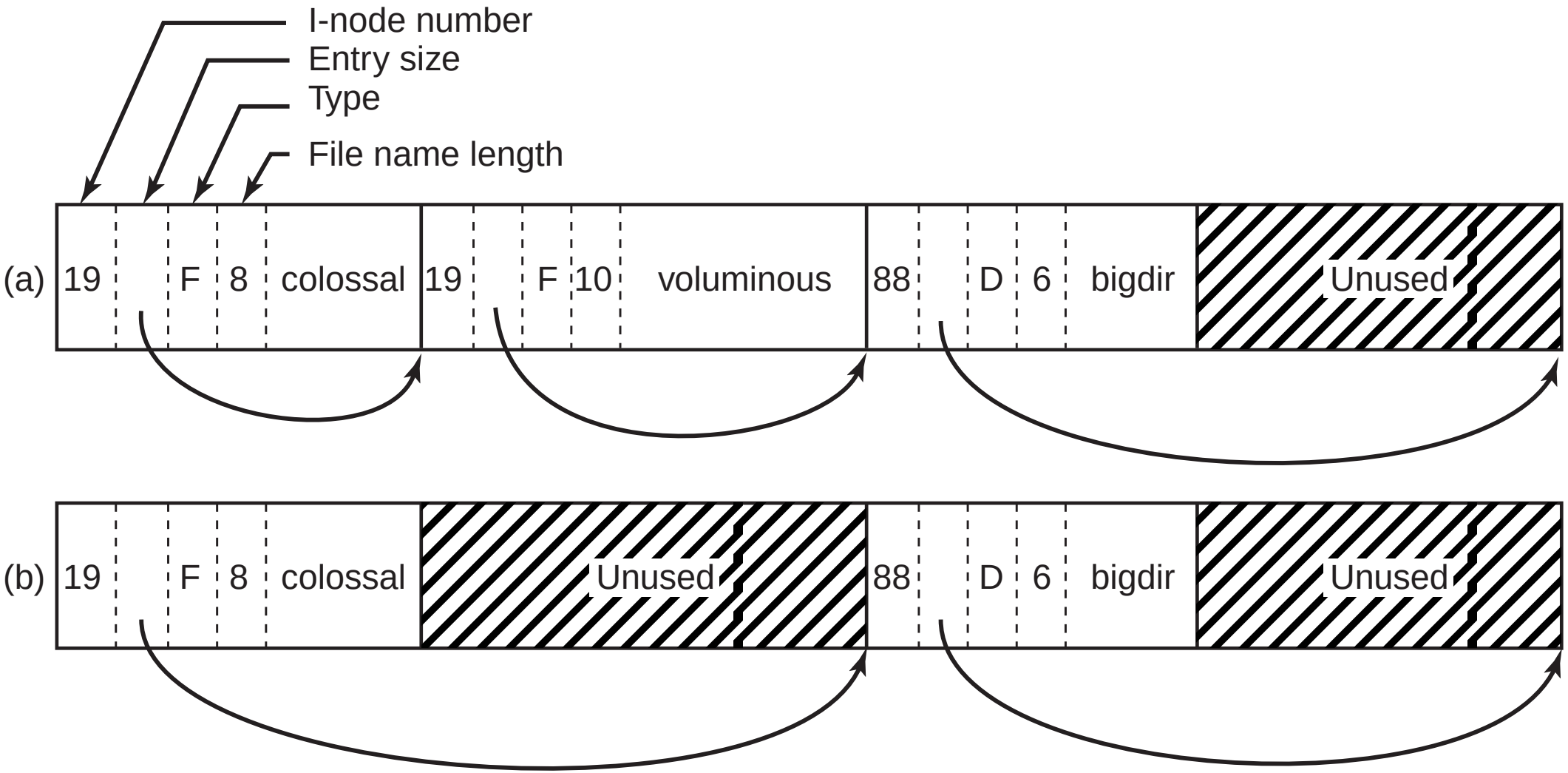
- Le chiamate di lettura/scrittura e la seek cambiano la posizione nel file

- 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

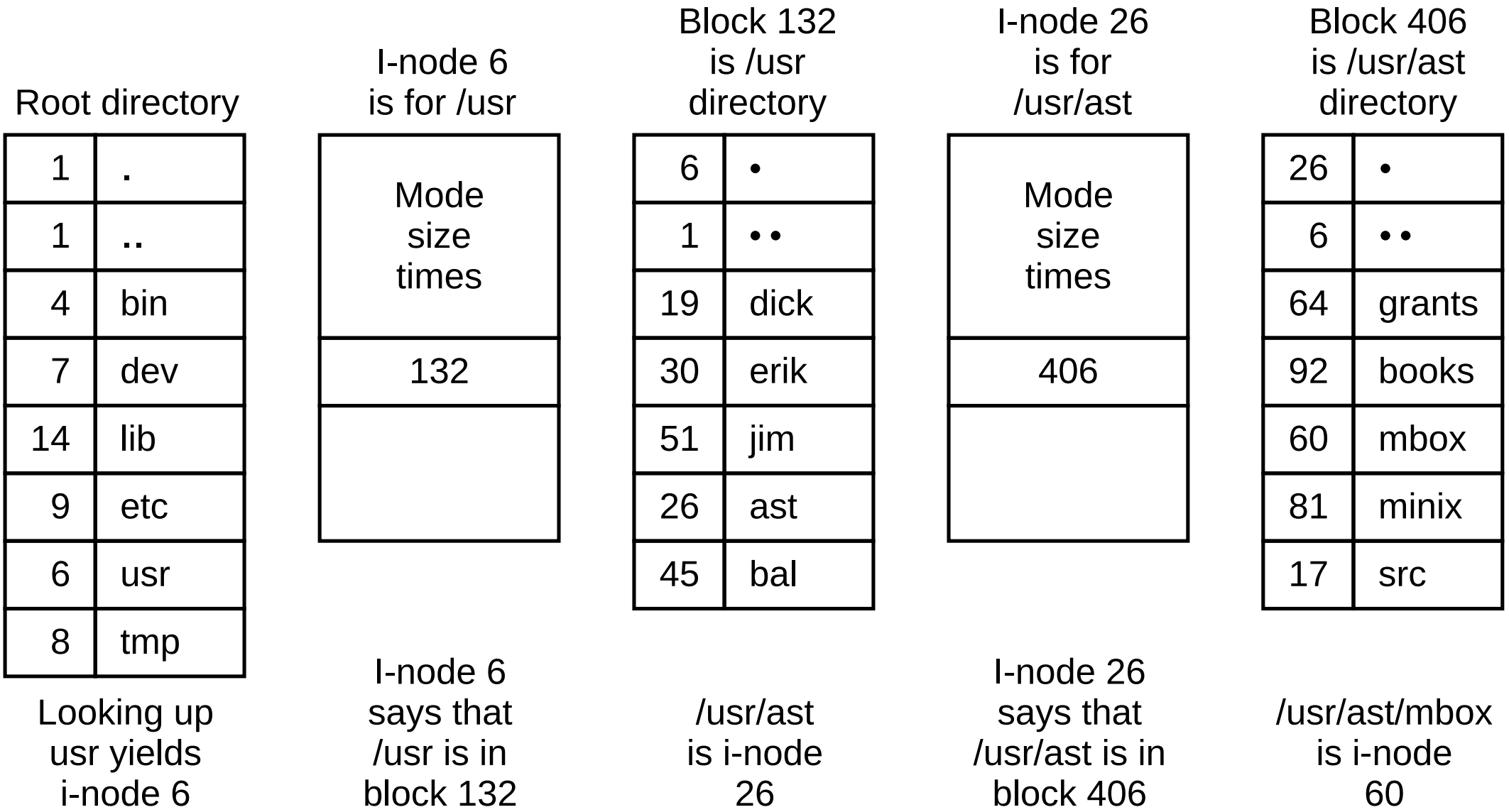
- Il tipo all'interno di un inode distingue tra file semplici e directory
- Una directory è un file con entry di lunghezza variabile. Ogni entry contiene
  - puntatore all'inode del file
  - posizione dell'entry successiva
  - lunghezza del nome del file (1 byte)
  - nome del file (max 255 byte)
- entry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


I-node number
Entry size
Type
File name length
(a)
19
F
8
colossal
19
F
10
voluminous
88
D
6
bigdir
Unused
(b)
19
F
8
colossal
Unused
88
D
6
bigdir
Unused

Traduzione da nome a inode

L'utente usa i nomi (o path), mentre il file system impiega gli inode $\Rightarrow$ 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)*


Root directory
1 .
1 ..
4 bin
7 dev
14 lib
9 etc
6 usr
8 tmp
Looking up usr yields i-node 6
I-node 6 is for /usr
Mode size times
132
I-node 6 says that /usr is in block 132
Block 132 is /usr directory
6 •
1 ••
19 dick
30 erik
51 jim
26 ast
45 bal
/usr/ast is i-node 26
I-node 26 is for /usr/ast
Mode size times
406
I-node 26 says that /usr/ast is in block 406
Block 406 is /usr/ast directory
26 •
6 ••
64 grants
92 books
60 mbox
81 minix
17 src
/usr/ast/mbox is i-node 60

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
- 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.

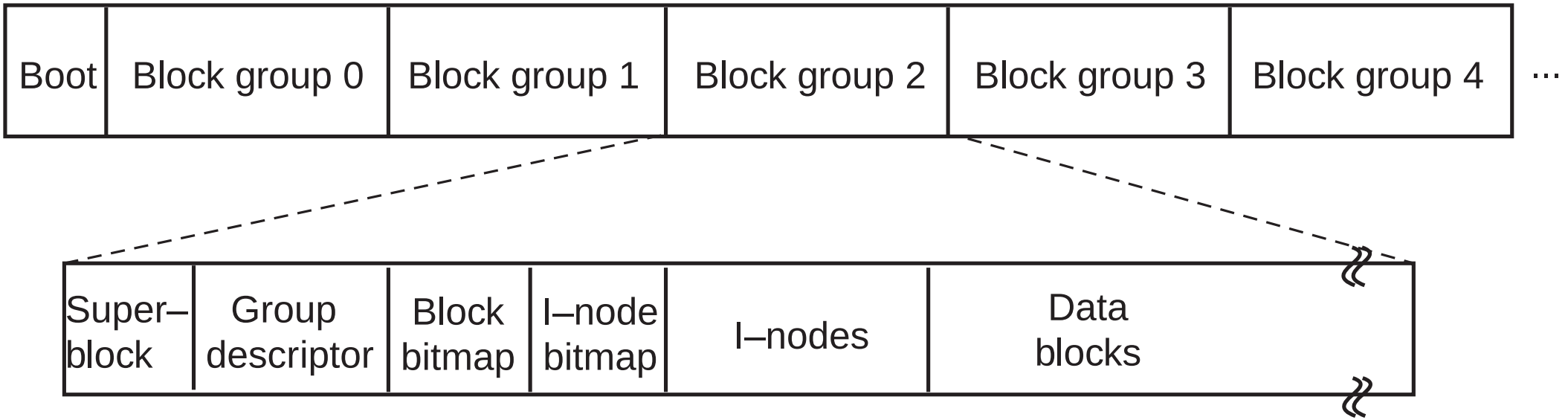
Esempio di file system fisico: Unix File System (Cont)

- 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 Berkley Fast File System (UFS)
- 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 2K, NT, XP

- Il file system NTFS è stato sviluppato *from scratch* per Windows NT
- Windows 2000 supporta sia FAT (-16 e -32) che NTFS
- NTFS supporta indirizzi di disco a 64 bit (ricordate che FAT-16 supporta indirizzi a 16 bit (max 2GB), e FAT-32 indirizzi a 28 bit (max 2TB))
- NTFS è un sistema molto più complesso del file system per MS-DOS

Caratteristiche di NTFS

- I nomi sono lunghi fino a 255 caratteri Unicode.
- I caratteri utilizzati sono in Unicode (2 byte per carattere)
- Si distinguono maiuscole e minuscole (ma le API Win32 no)
- A differenza di Unix e FAT-32, un file non è una sequenza lineare di file ma bensì si compone di attributi multipli (ad es. nome, flag, dati), ognuno rappresentato da una sequenza di byte
- L'idea di file come sequenza di attributi è stata introdotta dall'MacIntosh (MacOS)
- La lunghezza massima di una sequenza è 2^{64} byte (i.e. $18 exabyte \circ 18^{18}$)

Chiamate API Win32 del file system

- Le funzioni per il file system di API Win32 sono simili a quelle di Unix

- Ad esempio la chiamata `CreateFile` utilizzata per la creazione e apertura di un file restituisce un *gestore di file* (handle) come per il *file descriptor* di Unix

```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Implementazione di NTFS

- Diverse partizioni possono essere unite a formare un *volume logico*
- Lo spazio viene allocato in *cluster* (sequenze lineari di blocchi)
- Un blocco a dimensione variabile da 512byte e 64KB.
- La maggior parte dei dischi NTFS usa blocchi a 4KB
- La struttura principale in ogni volume è la Master File Table (MFT) che descrive file e directory

Master File Table (MFT)

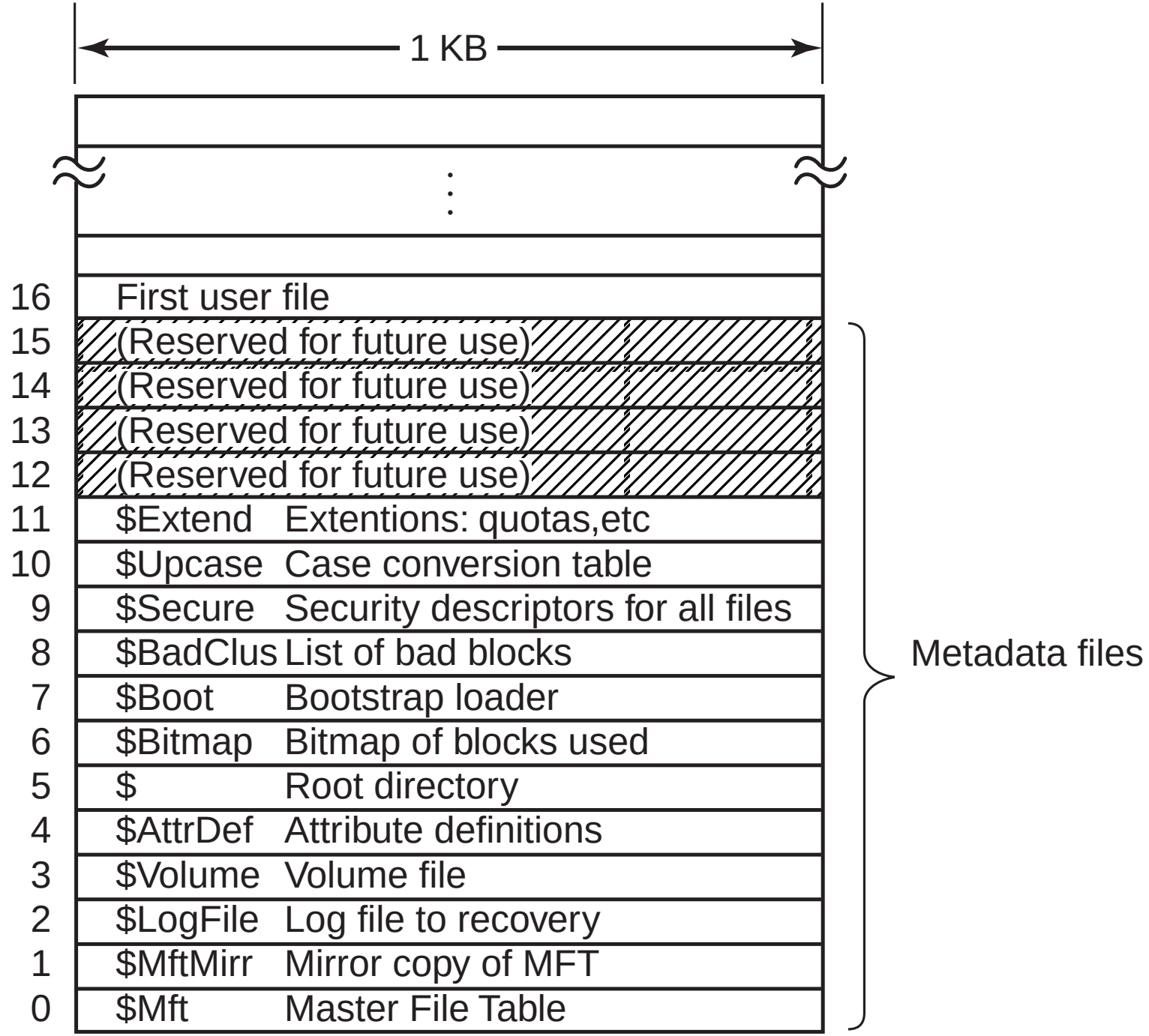
- È una sequenza lineare di record di 1KB
- Ogni record della MFT descrive un file o una directory e contiene attributi e la lista di indirizzi disco per quel file
- Per file grandi si possono usare più record per la lista dei blocchi (record base con puntatore ad altri record)
- Una mappa di bit tiene traccia dei record MFT liberi
- Una MFT è a sua volta vista come un file: può essere collocata ovunque su disco (evita il problema di settori imperfetti nella prima traccia), e può crescere fino a 2^{48} record

Struttura della MFT

- I primi 16 record descrivono l'MFT stesso e il volume (analogo al superblock di Unix).
  - Descrizione della MFT (primo record) e copia ridondante (secondo record): es. posizione dei blocchi della MFT
  - File di log: registra operazioni di aggiunta/rimozone/modifica al file system
  - Informazioni sul volume (etichetta, dimensione)
  - Informazioni sugli attributi dei file
  - Posizione della root dir (file anch'essa)
  - Attributi ed indirizzi di disco della bitmap per gestire lo spazio libero
  - Lista di blocchi malfunzionanti
  - Informazioni di sicurezza

  - Mappa dei caratteri maiuscoli (non sempre ovvia)
  - Informazioni su quote disco
  - Gli ultimi 4 record sono riservati ad usi futuri

- L'indirizzo del primo blocco della MFT viene memorizzato nel blocco d'avvio


1 KB
16 First user file
15 (Reserved for future use)
14 (Reserved for future use)
13 (Reserved for future use)
12 (Reserved for future use)
11 $Extend Extentions: quotas,etc
10 $Upcase Case conversion table
9 $Secure Security descriptors for all files
8 $BadClus List of bad blocks
7 $Boot Bootstrap loader
6 $Bitmap Bitmap of blocks used
5 $ Root directory
4 $AttrDef Attribute definitions
3 $Volume Volume file
2 $LogFile Log file to recovery
1 $MftMirr Mirror copy of MFT
0 $Mft Master File Table
Metadata files

Record della MFT

- Le informazioni sui file utente vengono memorizzate a partire dal record 16 della MFT

- Ogni record ha un'intestazione (header) seguita da una sequenza di coppie (intestazione di attributo e valore).

- Il record header contiene ad esempio uncontatore di riferimenti al file (come per i-node), ed il numero effettivo di byte usato nel record

- Ogni intestazione di attributo contiene il tipo dell'attributo e la locazione e la lunghezza del corrispondente valore

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

Attributi dei file NTFS

NTFS definisce 13 attributi standard quali

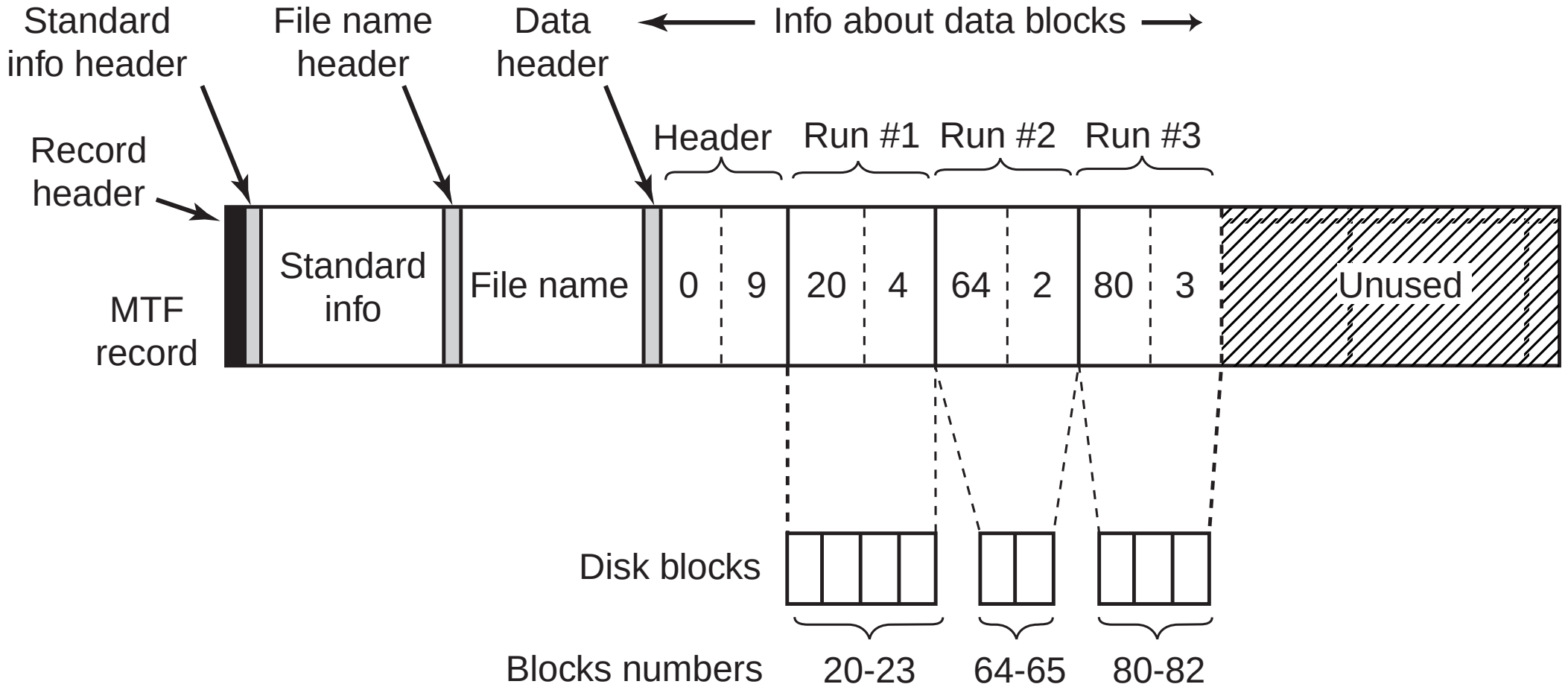
- Informazioni standard: proprietario, diritti, time-stamps, contatore di link fisici
- Nome del file (lunghezza variabile, Unicode)
- Identificatore di oggetto: nome unico per il file nel sistema
- Punti di analisi: per operazioni speciali durante il parsing del nome (ad es. montaggio)
- Dati: contiene gli indirizzi disco dei veri dati; se sono residenti, il file si dice "immediate"

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

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



- L'header del file contiene l'offset del primo blocco (e.g. 0) e il numero di blocchi totale (e.g. 9)

- Gli indirizzi dei run sono memorizzati a coppie (numero del primo blocco, numero dei blocchi del run)

- Nota: un file con n blocchi può essere memorizzato in numero di range che varia da 1 a n

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

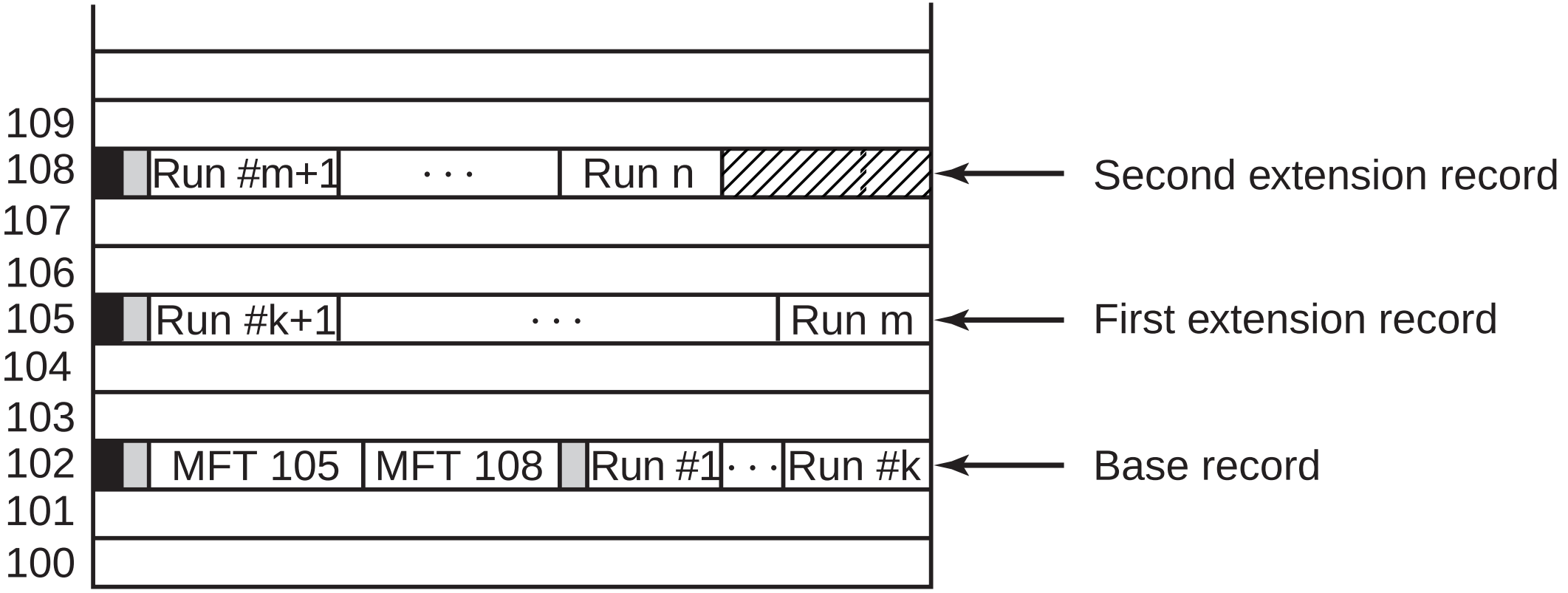
Dimensione File NTFS

- Non esiste un limite superiore alla dimensione di un file
- Ogni coppia richiede due numeri a 64 bit: 16 byte
- Una coppia può rappresentare più di un milione di blocchi disco consecutivi (ad es. 20 sequenze separate da 1 milione di blocchi da $1KB = 20KB$)
- Si usano metodi di compressione per memorizzate le coppie in meno di 16byte (si arriva fino a 4byte)

File "long"

Se il file è lungo o molto frammentato (es. disco frammentato), possono servire più di un record nell'MFT.

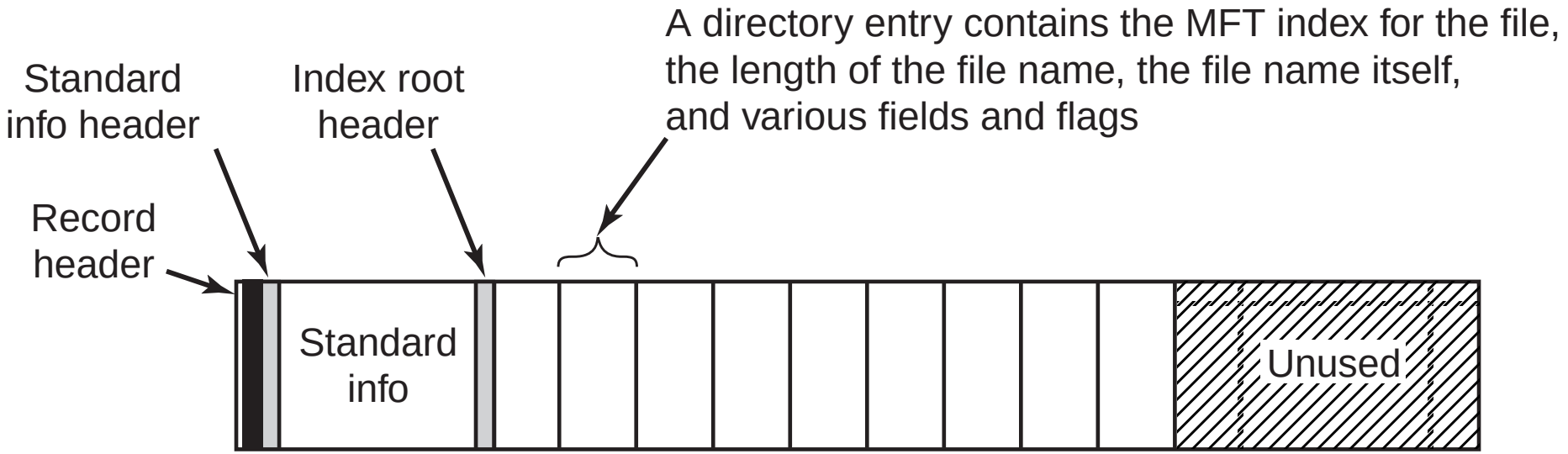
Prima si elencano tutti i record aggiuntivi, e poi seguono i puntatori ai run.



Se i puntatori ai record aggiuntivi non stanno su un solo MFT si memorizza la lista dei record con i blocchi su disco invece che nel MFT

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.

Navigazione in NTFS

- Consideriamo l'indirizzo

  C:\maria\web.html

- La directory radice \ contiene un puntatore alla lista di nomi di volumi logici (C, E, D, ecc)

- Il nome C: è un collegamento simbolico con la partizione del disco

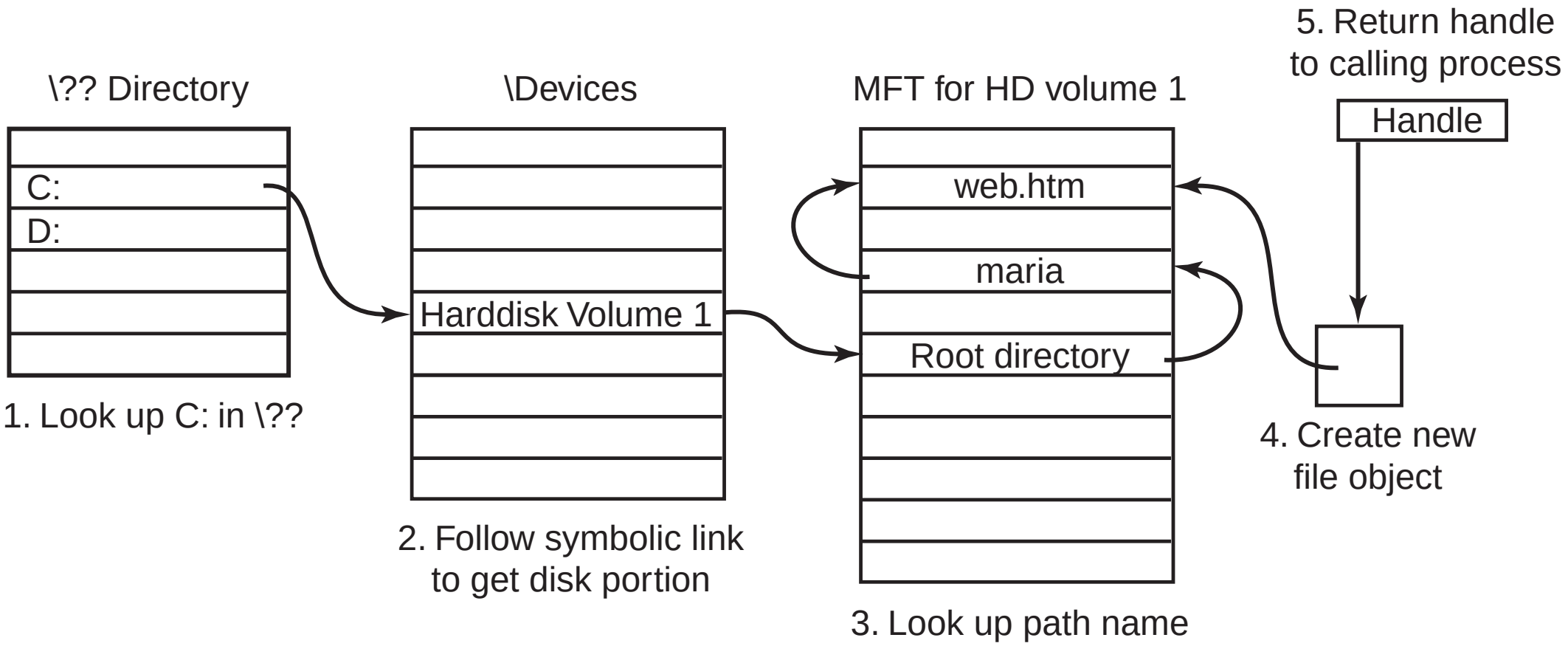
- Una volta identificata la partizione di può recuperare la corrispondente MFT

- Nel record 5 della MFT troviamo informazioni sulla directory radice del disco C

- La stringa *maria* viene cercata all'interno del record della directory radice, da tale ricerca otteniamo un indice nel MFT per la directory maria

- Quindi esaminiamo il record alla ricerca di *web.html*

- Se la ricerca ha successo si crea un nuovo *handle* (oggetto) che contiene l'indice del file nel MFT

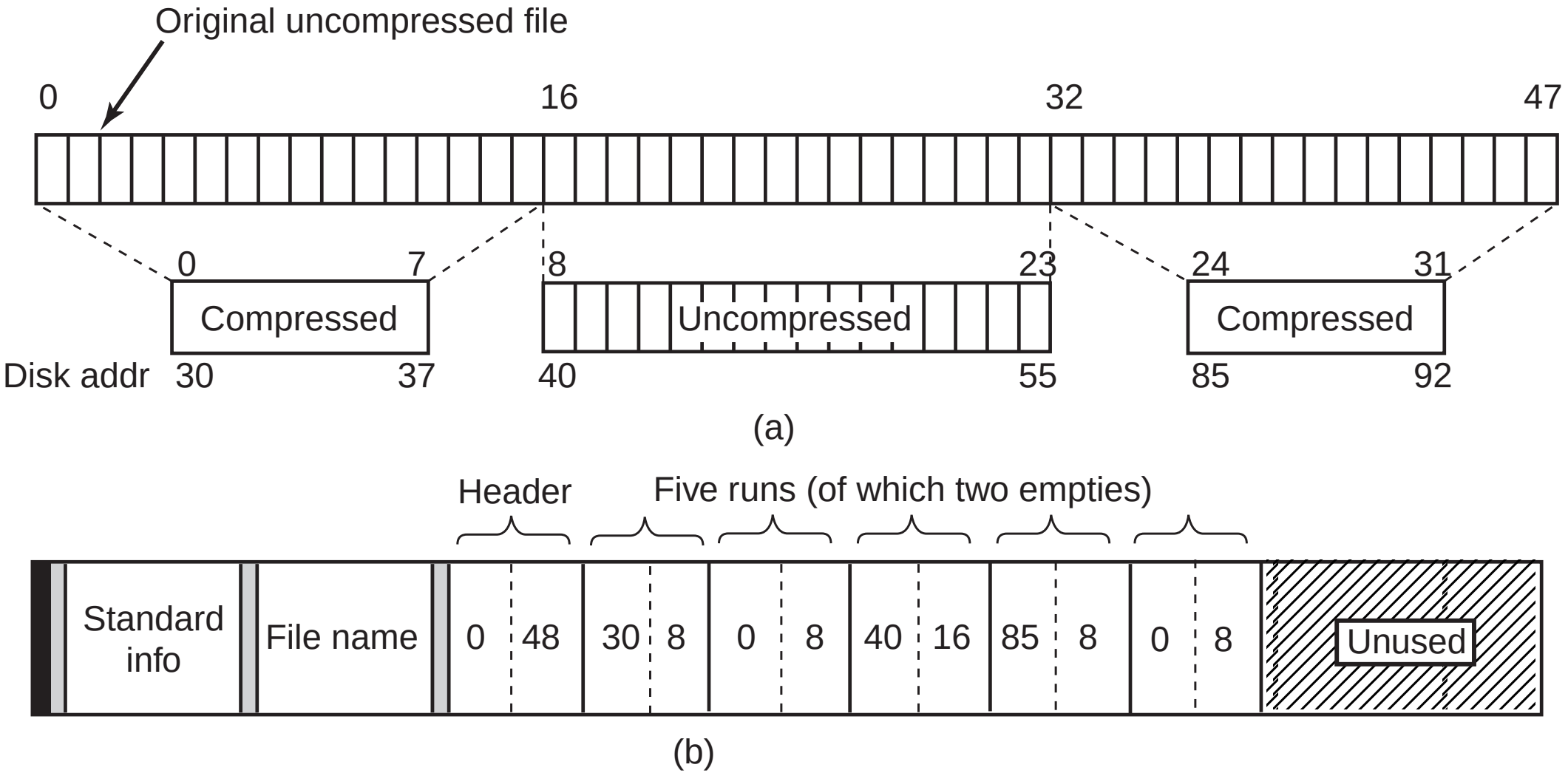

\?? Directory
C:
D:
1. Look up C: in \??
\Devices
Harddisk Volume 1
2. Follow symbolic link
to get disk portion
MFT for HD volume 1
web.htm
maria
Root directory
3. Look up path name
5. Return handle
to calling process
Handle
4. Create new
file object

Compressione file

- NTFS supporta la compressione trasparente dei file (cioè i file vengono compressi quando creati e decompressi in lettura)

- L'algoritmo di compressione lavora su gruppi di 16 blocchi: se si riescono a comprimere si scrivono i blocchi compressi e si memorizzano nel record MFT dei blocchi virtuali (con indirizzo di disco 0) per i blocchi mancanti;

- poi si prosegue con i successivi 16 blocchi



- Quando NFTS legge un file e trova due coppie consecutive $(n, m)(0, k)$ capisce che $m + k$ sono stati compressi in m blocchi e applica l'algorimo di decompressione a quella sotto-sequenza

File System CDRom

- File system particolarmente semplici in quanto progettati per supporti di sola lettura

- Ad esempio, non viene tenuta traccia dei blocchi liberi: i CDROM non vengono modificati

- Esistono vari tipi di file system: lo standard ISO 9660 specifica il tipo comunemente adottato dai supporti di lettura per CDROM

Organizzazione dei CDROM

- I CDROM hanno una spirale continua che contiene i bit in una sequenza lineare

- I bit lungo la spirale sono divisi in blocchi di 2532 byte:
  - 2048 byte sono di dati,
  - I byte rimanenti contengono header e codici di correzione

Standard ISO 9660

- File system leggibile da tutti i lettori e supportato dai principali sistemi operativi (anche MS-DOS)

- Lo standard definisce un file system per un'insieme di CDROM (fino a $2^{16} - 1$ CDROM), ognuno partizionato in volumi logici

- Ogni CDRom inzia con 16 blocchi la cui funzione non è definita dallo standard (vengono usati ad esempio per programmi di avvio)

- Di seguito si trova il descrittore di volume primario che contiene informazioni generali sul CDROM quali

  - identificatore di sistema (32 byte)

  - identificatore di volume (32 byte)

  - identificatore del distributore (128 byte)
  - identificatore del preparatore dei dati (128 byte)
  - nomi di tre file (riassunto, avviso di copyright ed informazioni bibliografiche)

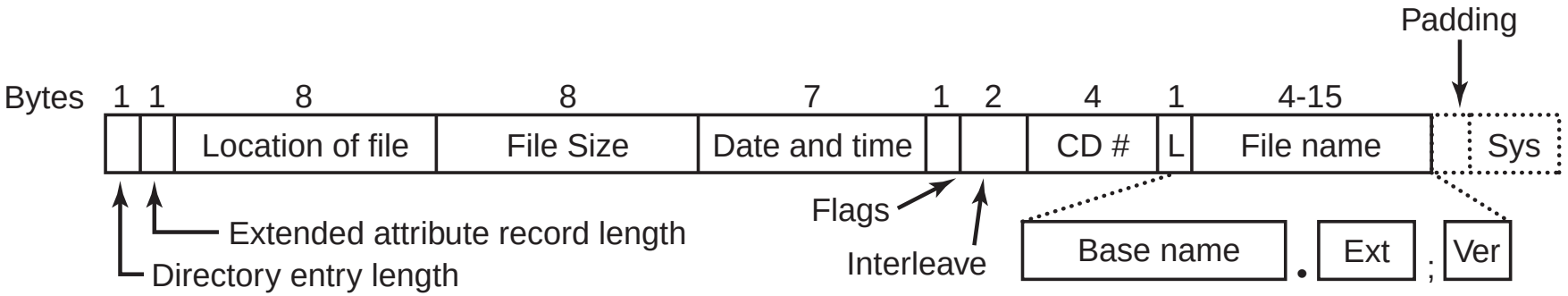
- Il descrittore contiene anche i seguenti numeri chiave
  - dimensione del blocco logico (normalmente 2048, in alcuni casi 4096, 8192)
  - numero di blocchi
  - date di creazione e scadenza

- Infine contiene un elemento di directory per la directory radice dei dati memorizzati

Directory in un CDROM

- Le directory sono composte di un numero variabile di elementi
- Gli elementi sono di dimensione variabile (tra 10 e 12 campi)
- La profondit‘a di una directory è al più di 8 livelli, mentre non c’è limite al numero di elementi in una directory
- I primi due elementi indicano la directory corrente e la directory genitore

Elementi di Directory in un CDROM



- Il primo byte identifica la lunghezza dell'elemento di directory
- Il secondo byte la lunghezza del record di eventuali attributi estesi
- Poi seguono
  - Posizione del file: i file sono identificati tramite la coppia posizione del primo blocco e lunghezza (i file sono memorizzati come blocchi contigui)
  - Data di registrazione (range 1900-2155)

- Flag: contiene bit per distinguere file da directory, bit per nascondere file, bit per abilitare uso di attributi estesi, bit per marcare l'ultimo elemento di directory
- Interlacciamento: usato solo in versioni avanzate
- Numero del CDROM sul quale è posizionato il file (l'elemento potrebbe far riferimento ad un file su un'altro CD dell'insieme)
- Dimensione del nome del file in byte
- Nome del file: nome (8 caratteri), punto, estensione (3 caratteri)
- Padding: usato per far si che ogni elemento di directory sia formato da un numero pari di byte (per allineamento)
- System use: utilizzato in modo speciale dai diversi sistemi operativi

Livelli in ISO 9660

- Lo standard definisce tre livelli
  - Livello 1: nomi con 8+3 caratteri e file memorizzati in blocchi contigui
  - Livello 2: nomi fino a 31 caratteri
  - Livello 3: nomi come nel livello 2, file formati da diverse sezioni di blocchi contigui; le sezioni possono essere condivise tra diversi file, o comparire più volte nello stesso file

Estensione Rock Ridge

- L'estensione Rock Ridge permette di rappresentare file system Unix in un CDROM

- Le informazioni aggiuntive vengono memorizzate nel campo *system use* e sono ad esempio:
  - Bit Unix per i diritti sui file, bit SETUID e SETGID
  - Nome alternativo: nome UNIX per il file
  - Campi per la rilocazione di una directory (per superare il limite di 8 livelli)
  - Time stamp contenuti negli inode di un file Unix (creazione, ultima modifica, ultimo accesso)

Estensione Joliet

- L'estensione Joliet permette di rappresentare file system Windows in un CDROM

- Le informazioni aggiuntive vengono memorizzate nel campo *system use* e specificano ad esempio:
  - Nome di file lungo (fino a 64 caratteri)
  - Insiemi di caratteri Unicode per i nomi (caratteri unicode occupano 2 byte, quindi nomi di al più 128 byte)
  - Profondità della struttura della directory maggiore di 8 livelli
  - Nomi di directory con estensione (non usato per ora)