

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 linguaggio macchina
Oggetto	obj, o	compilato, in linguaggio macchina, non linkato
Codice sorgente	c, p, pas, f77, asm, java	codice sorgente in diversi lin- guaggi
Batch	bat, sh	script per l'interprete comandi
Testo	txt, doc	documenti, testo
Word processor	wp, tex, doc	svariati formati
Librerie	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 principale, 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

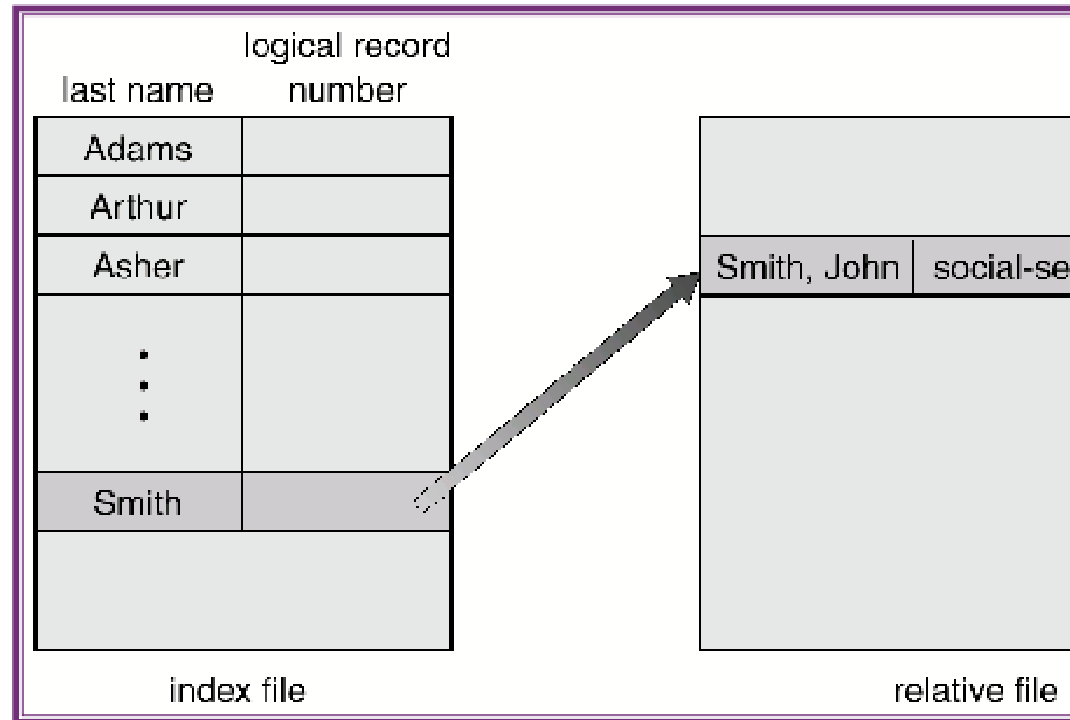
- Puntatore al file: posizione raggiunta durante la lettura/scrittura
- Contatore dei file aperti: quanti processi stanno utilizzando il file
- Posizione sul disco

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 */

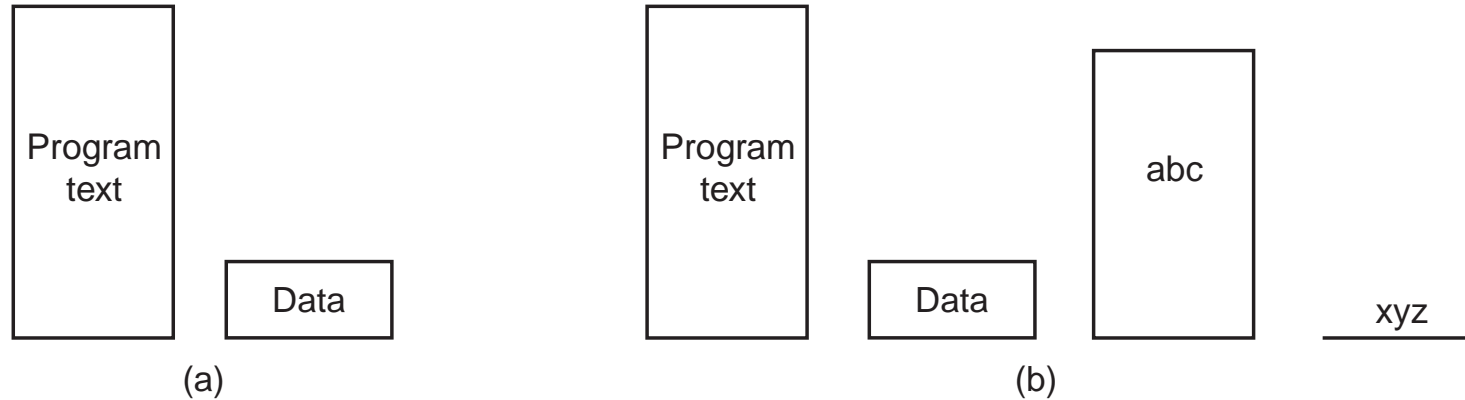
    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;                 /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);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



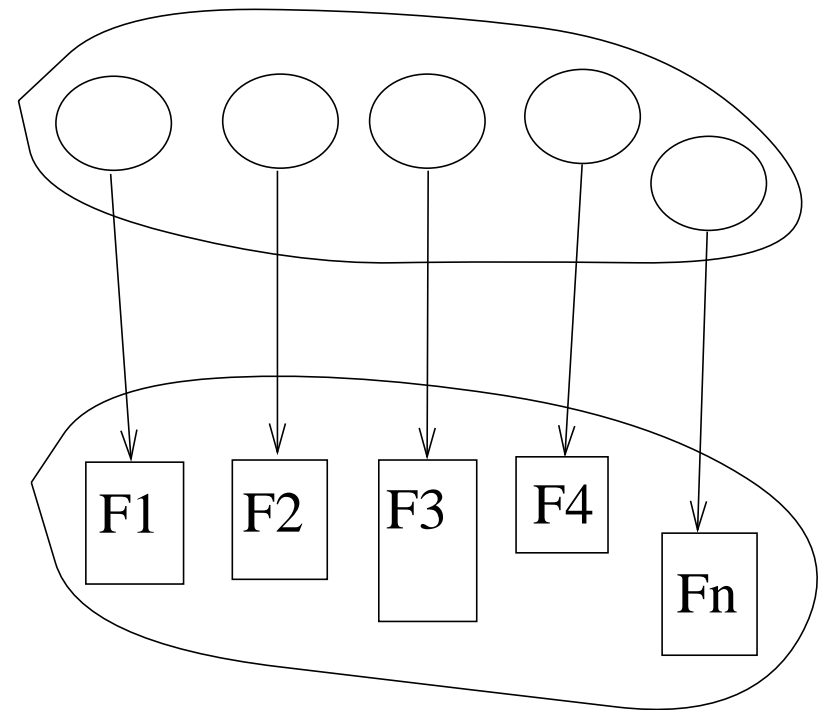
- Relativamente semplice da implementare in sistemi segmentati (con o senza paginazione): il file viene visto come area di swap per il segmento mappato
- Non servono chiamate di sistema `read` e `write`, solo una `mmap`
- Problemi
 - lunghezza del file non nota al sistema operativo
 - accesso condiviso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

Directory

Files



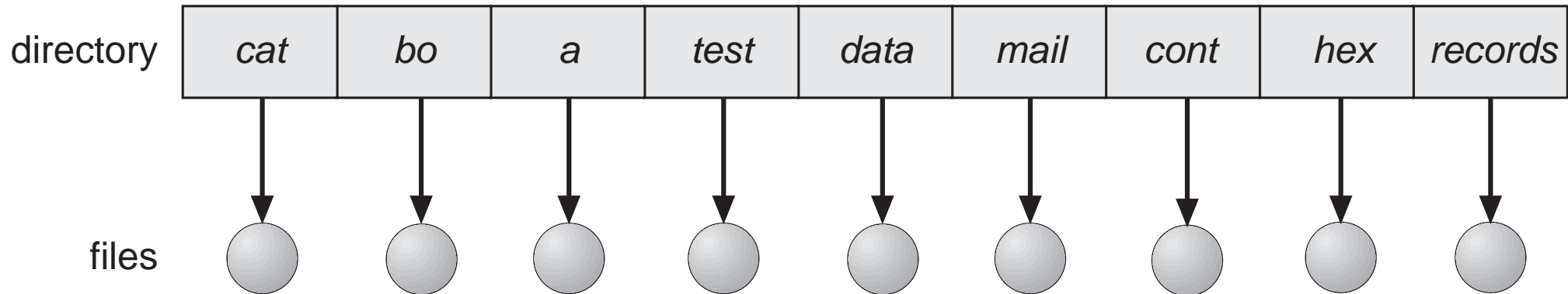
Organizzazione logica delle directory

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, ...)

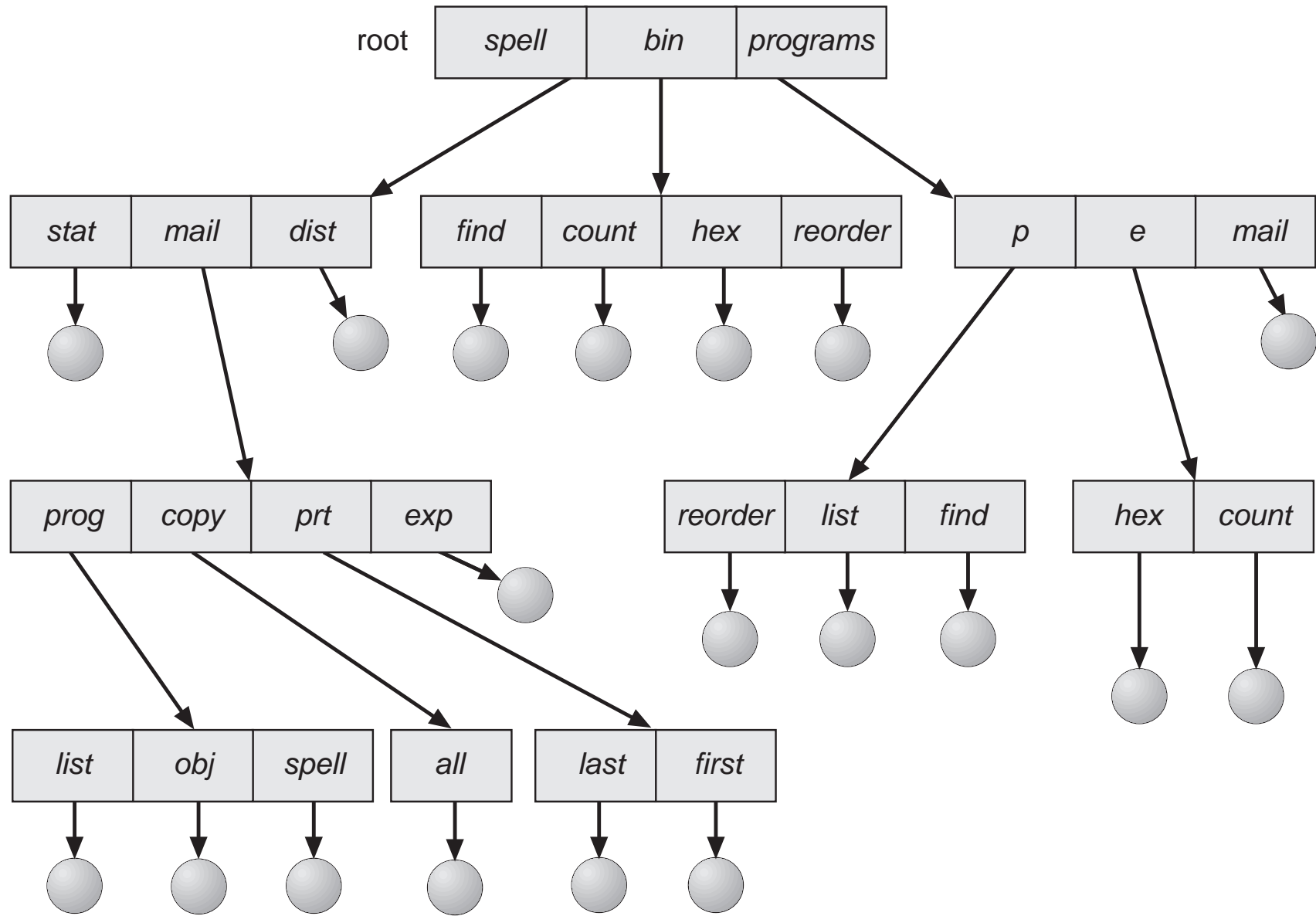
Tipi di directory: unica (“flat”)

- 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



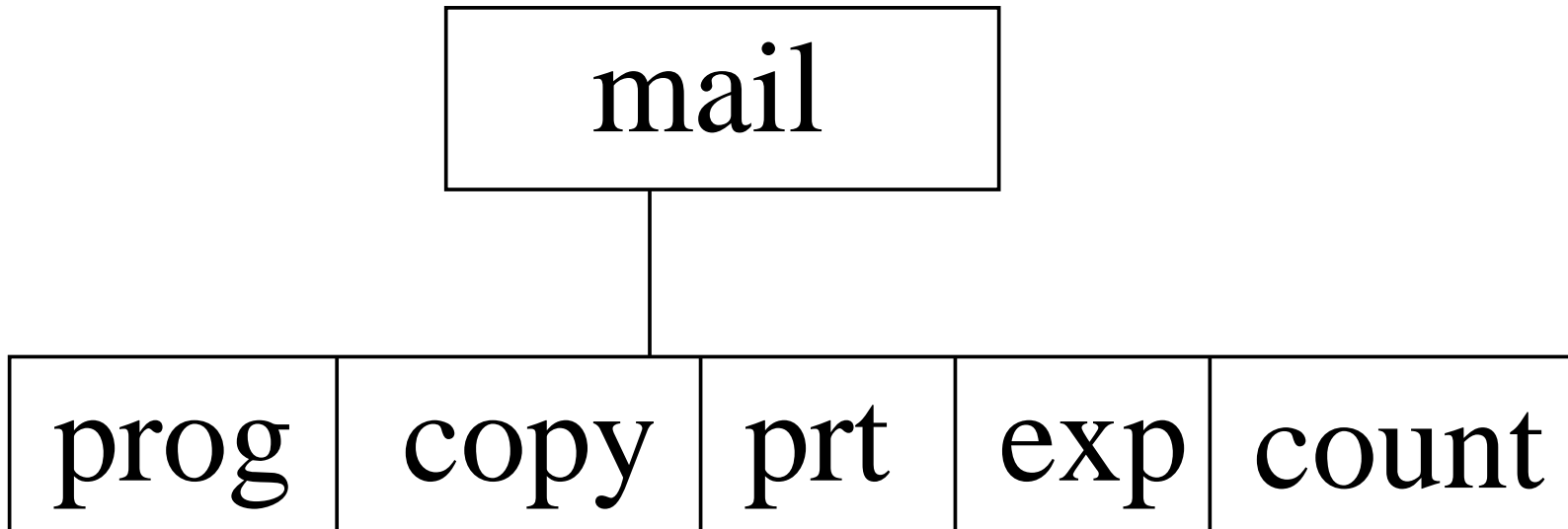
Directory ad albero (cont.)

- 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

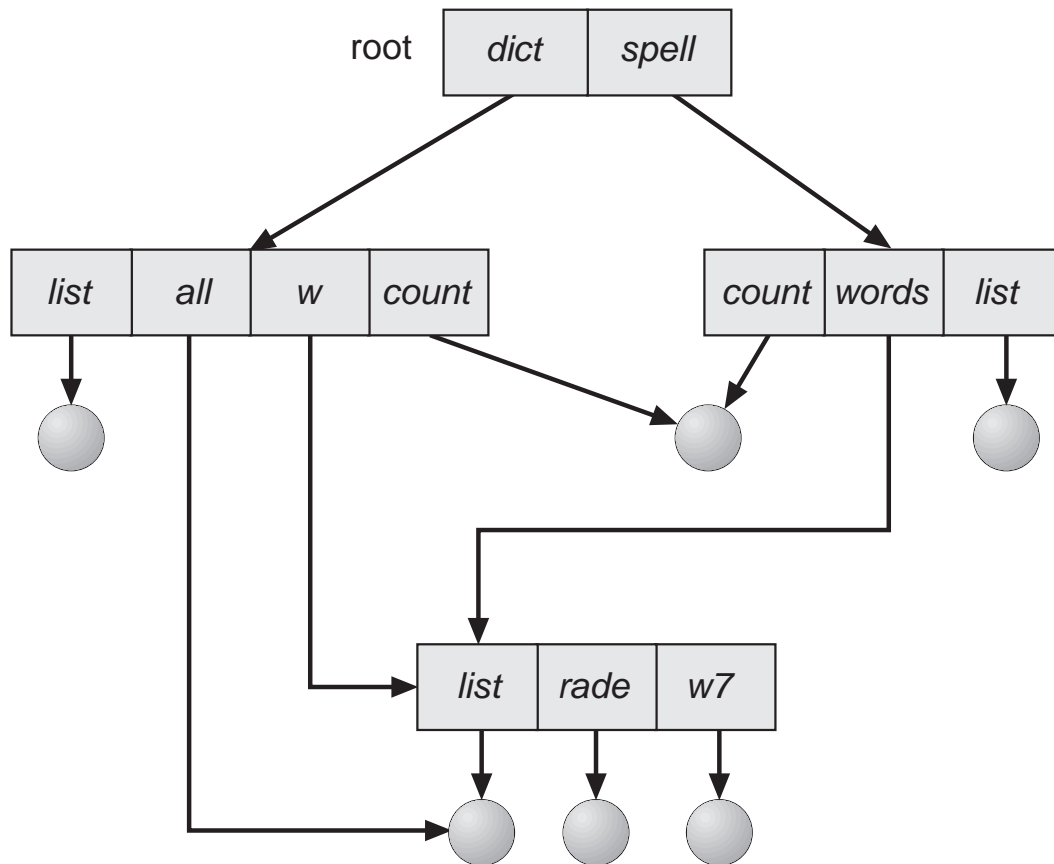


- Cancellando `mail` si cancella l'intero sottoalbero

Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory

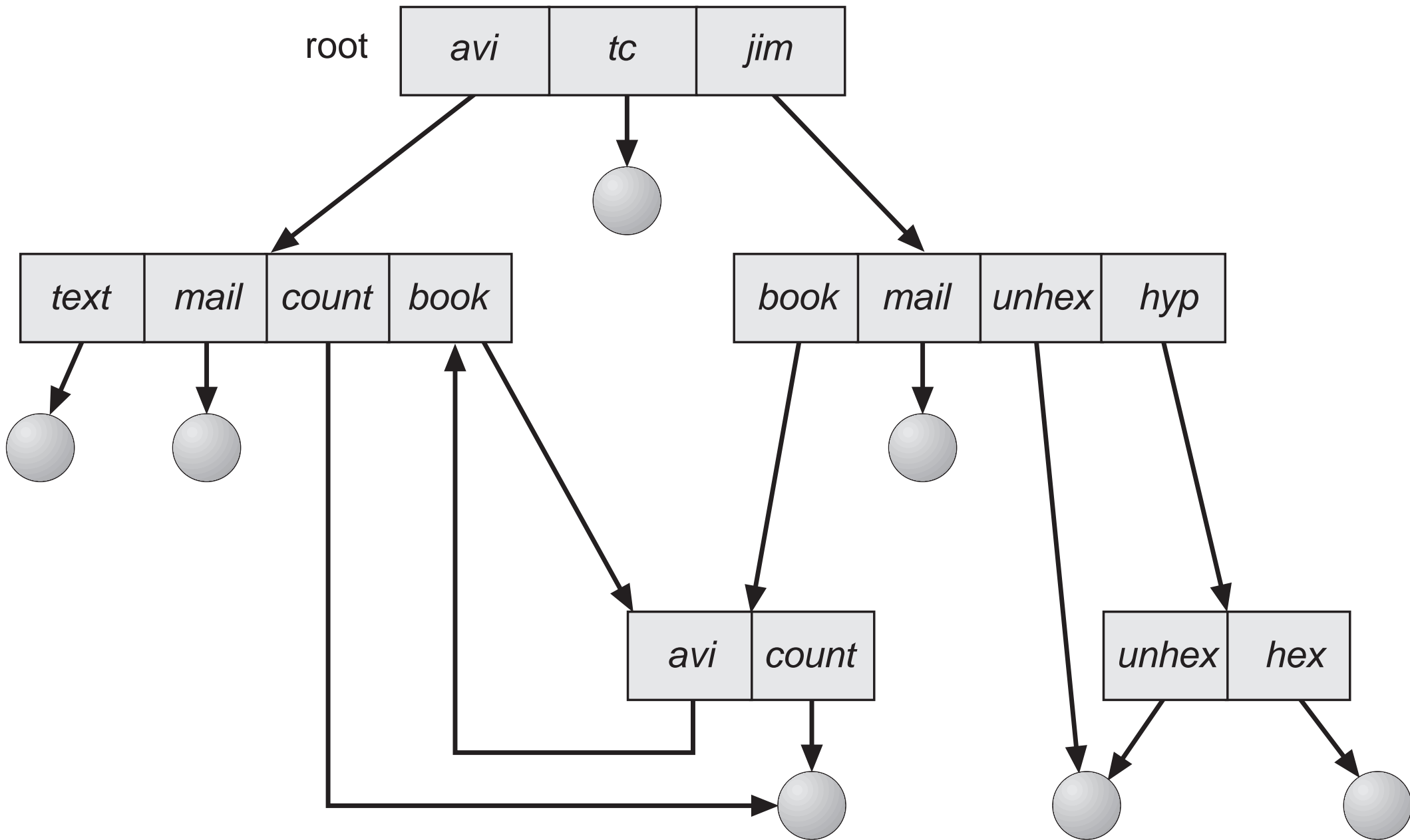
Due nomi differenti per lo stesso file (aliasing)



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	⇒	1 1 1
b)	groups access	6	⇒	1 1 0
c)	public access	1	⇒	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)*

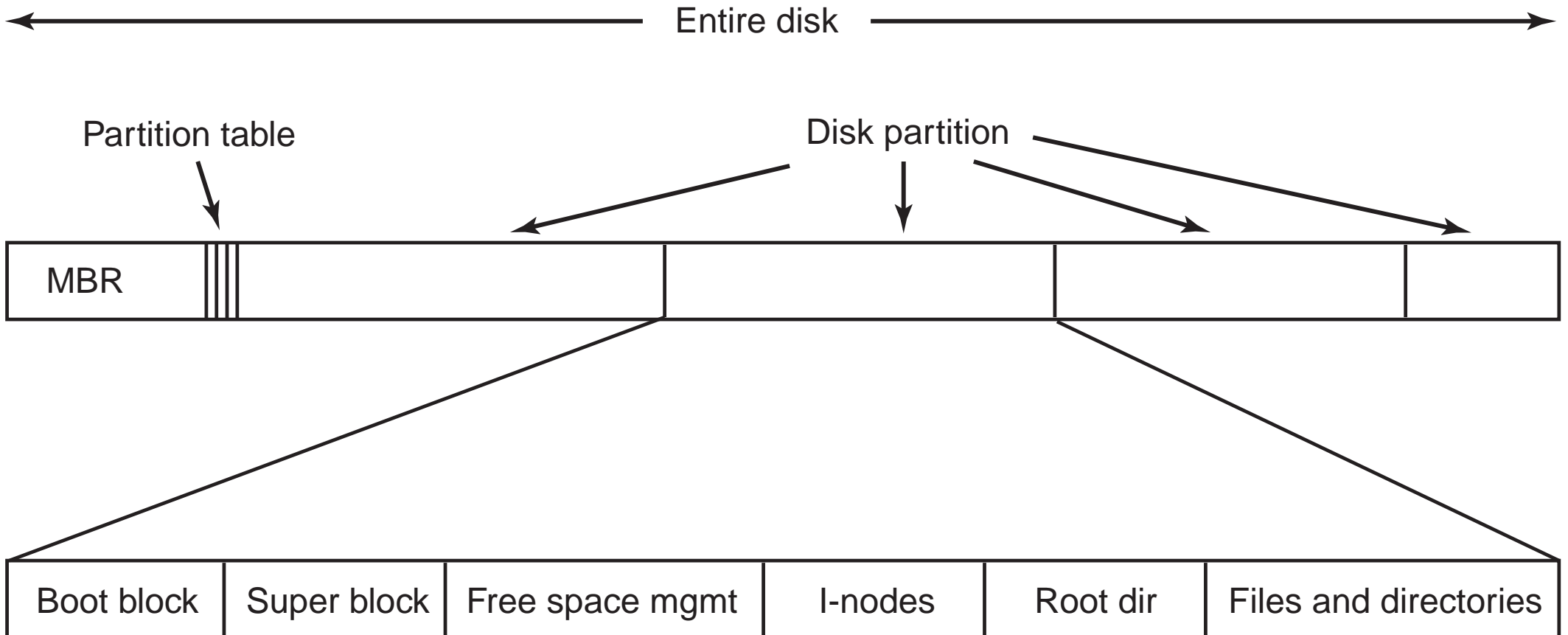
Setuid/setgid bit

- 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 è 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 così via.

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.

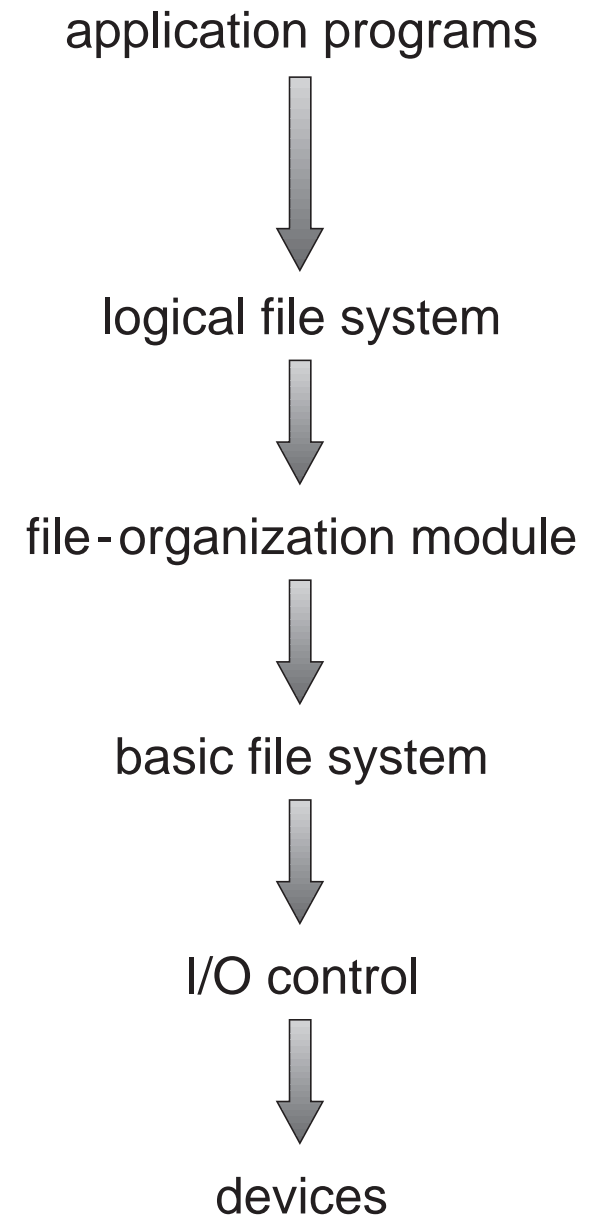


Tabella dei file aperti

- 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 $LA = \text{indirizzo logico}$

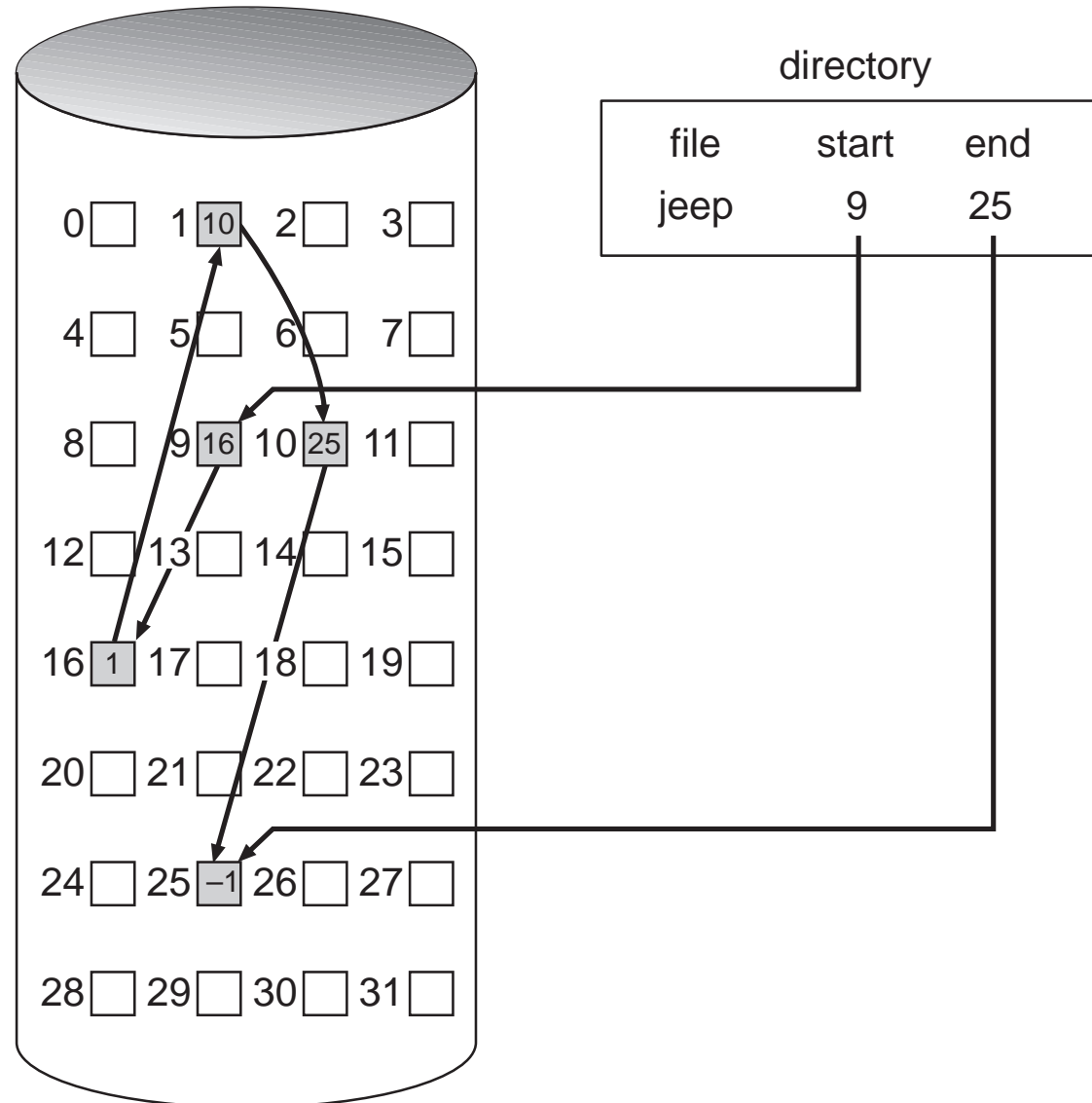
$$LA/512 \begin{cases} Q \\ R \end{cases}$$

e $Q = \text{quoziente}$, $R = \text{resto}$ allora

- Il blocco da accedere = $Q + \text{blocco di partenza}$
- 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):

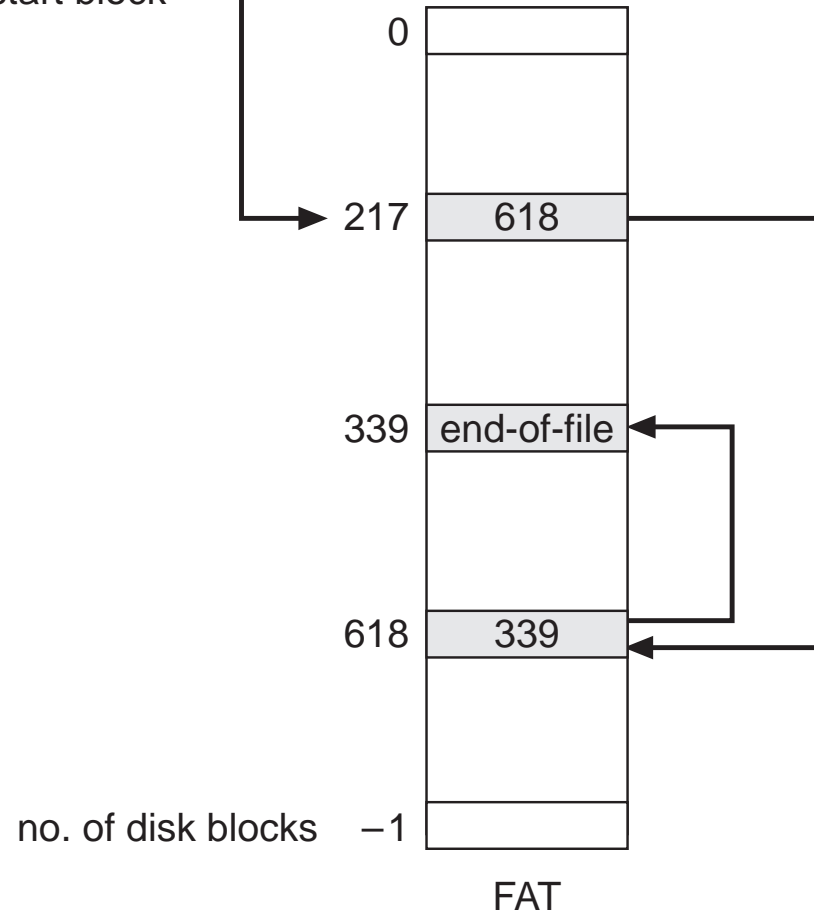
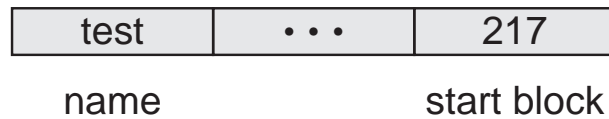
$$LA/511 \begin{cases} Q \\ R \end{cases}$$

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

Allocazione concatenata (cont.)

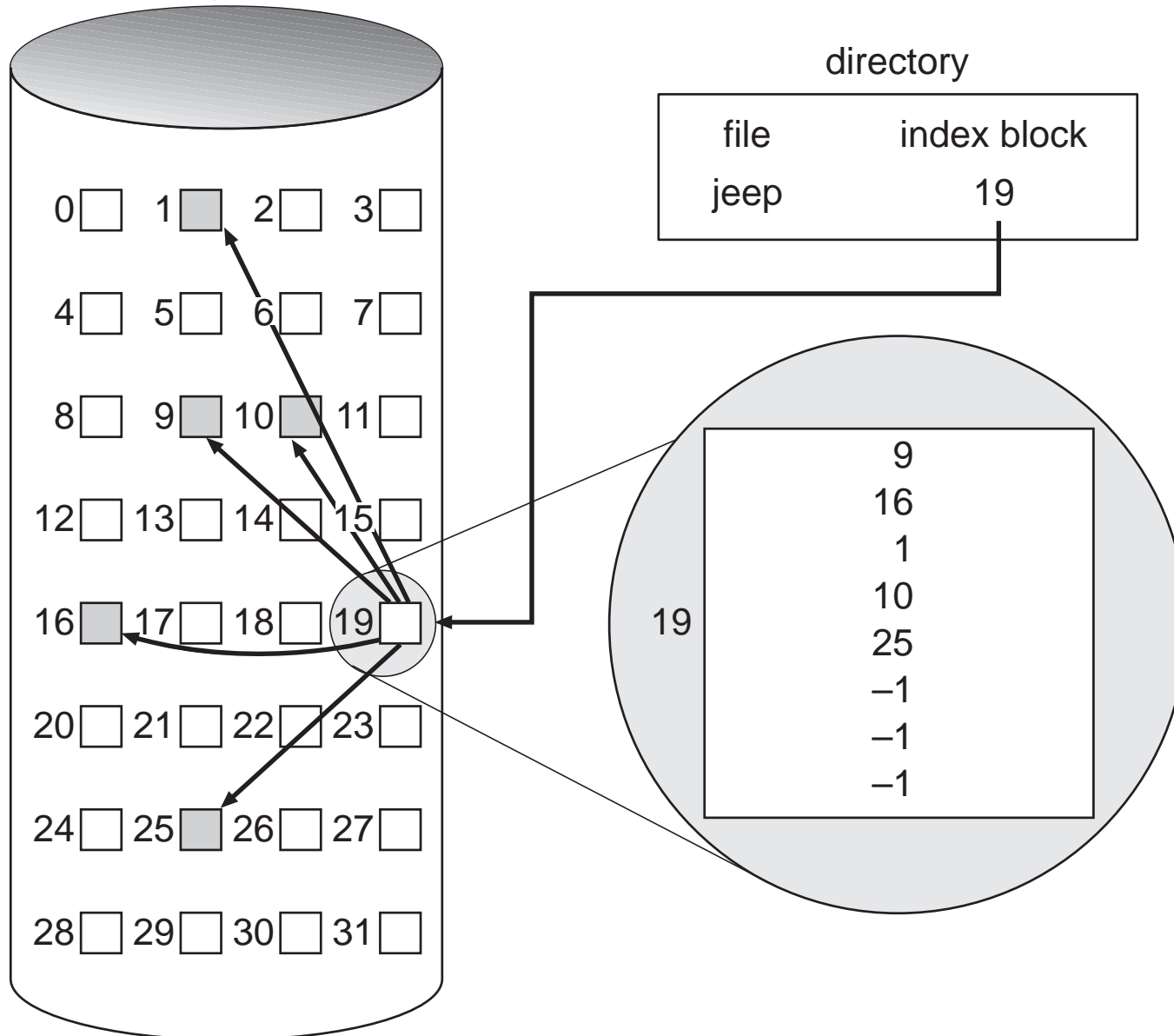
Variante: *File-allocation table (FAT)* di MS-DOS e Windows. Mantiene la linked list in una struttura dedicata, all'inizio di ogni *partizione* del disco

directory entry



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 ($512 \times 512 = 262144 = 256K$)

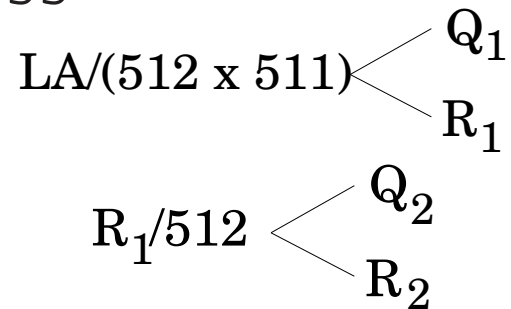
$$LA/512 \begin{cases} Q \\ R \end{cases}$$

– Q = offset nell'indice

– R = offset nel blocco indicato dall'indice

Allocazione indicizzata (cont.)

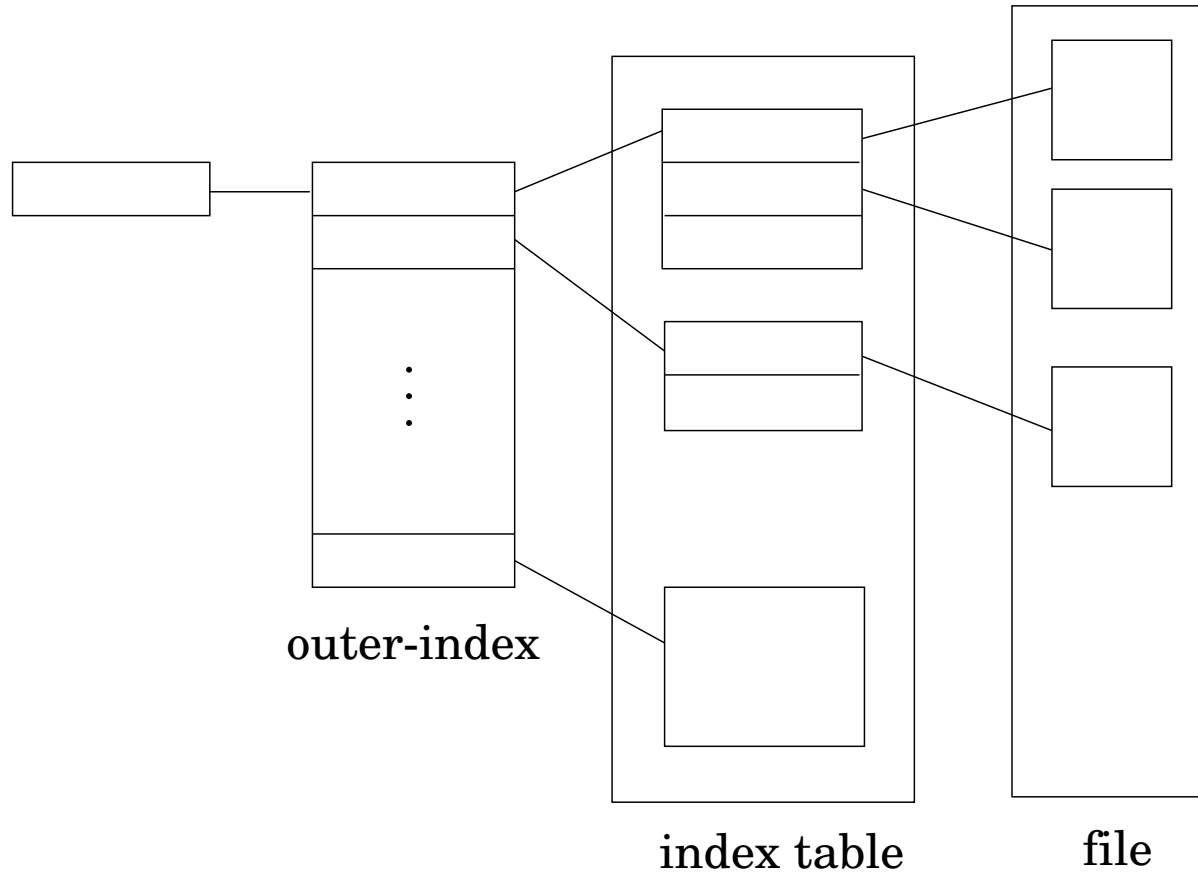
- Problema: come implementare il blocco indice
 - è una struttura supplementare: overhead \Rightarrow meglio piccolo
 - dobbiamo supportare anche file di grandi dimensioni \Rightarrow meglio grande
- Indice concatenato: l'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza, maggiore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



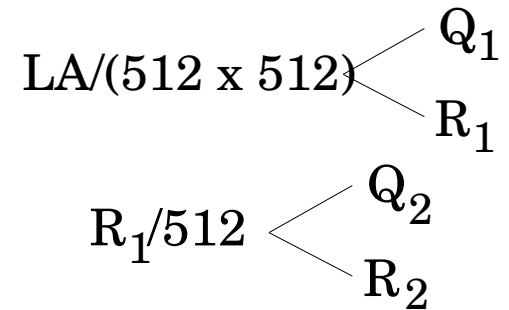
- Q_1 = blocco dell'indice da accedere
- Q_2 = offset all'interno del blocco dell'indice
- R_2 = offset all'interno del blocco del file

Allocazione indicizzata (cont.)

Indice a due (o più) livelli.



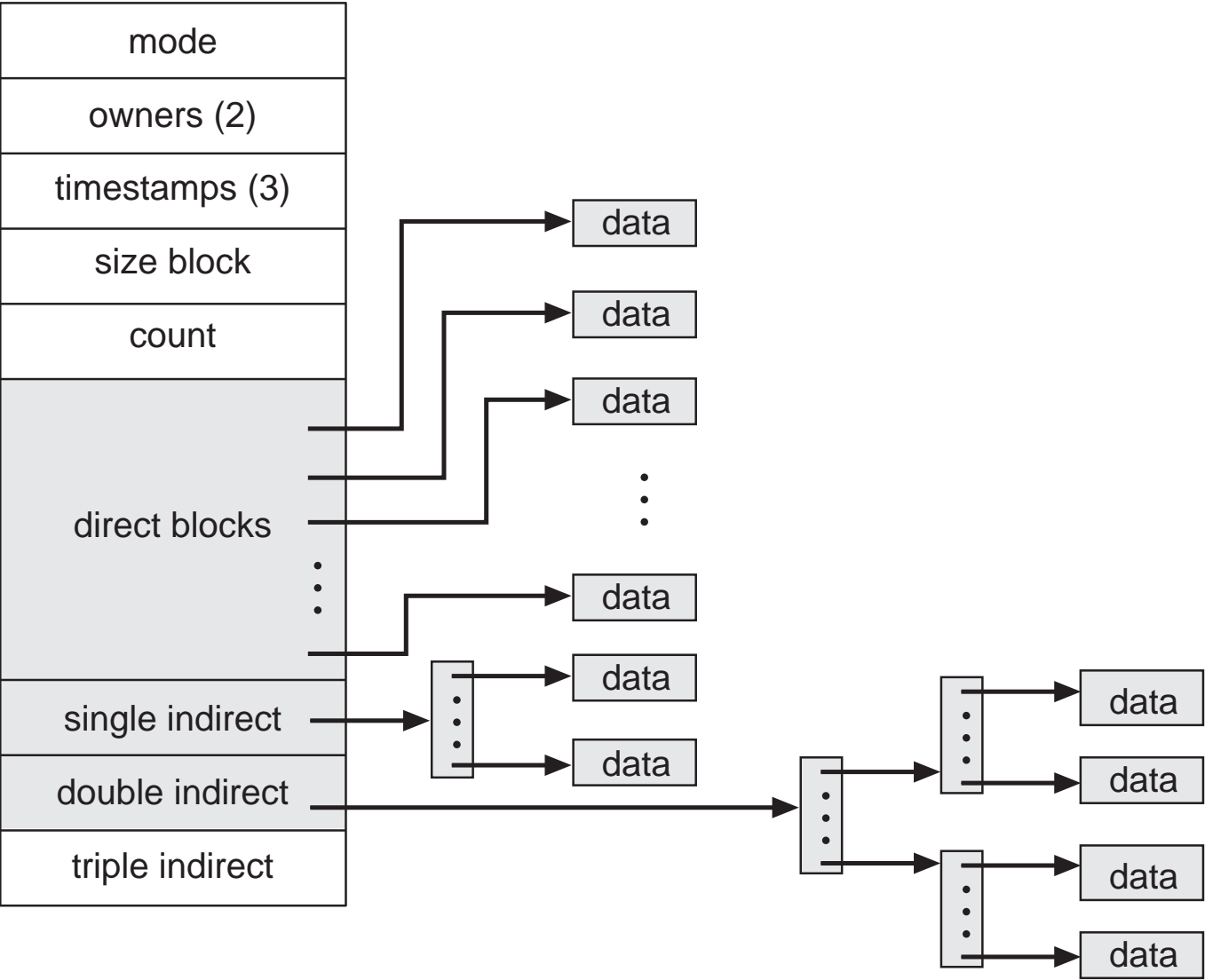
- Con blocchi da 512 parole:



- 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!)



Inodes (cont.)

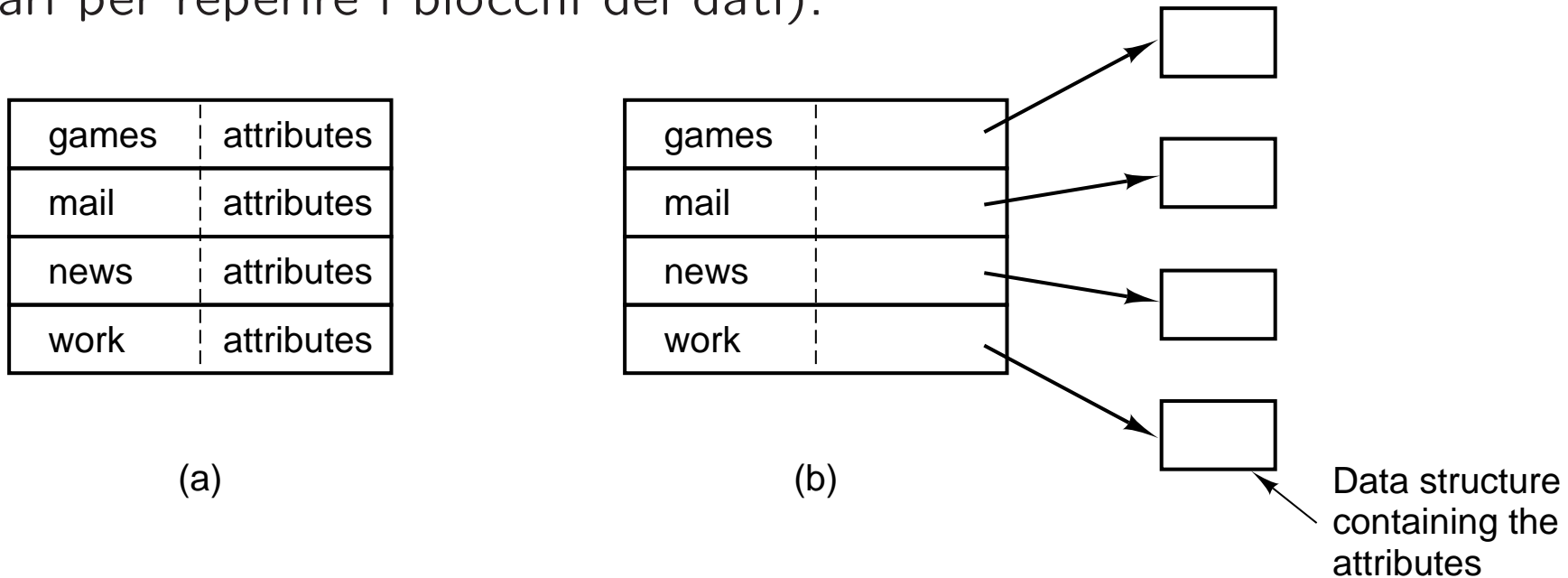
- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte (cioè indice=1 blocco ha 1024 riferimenti a blocchi)

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

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).

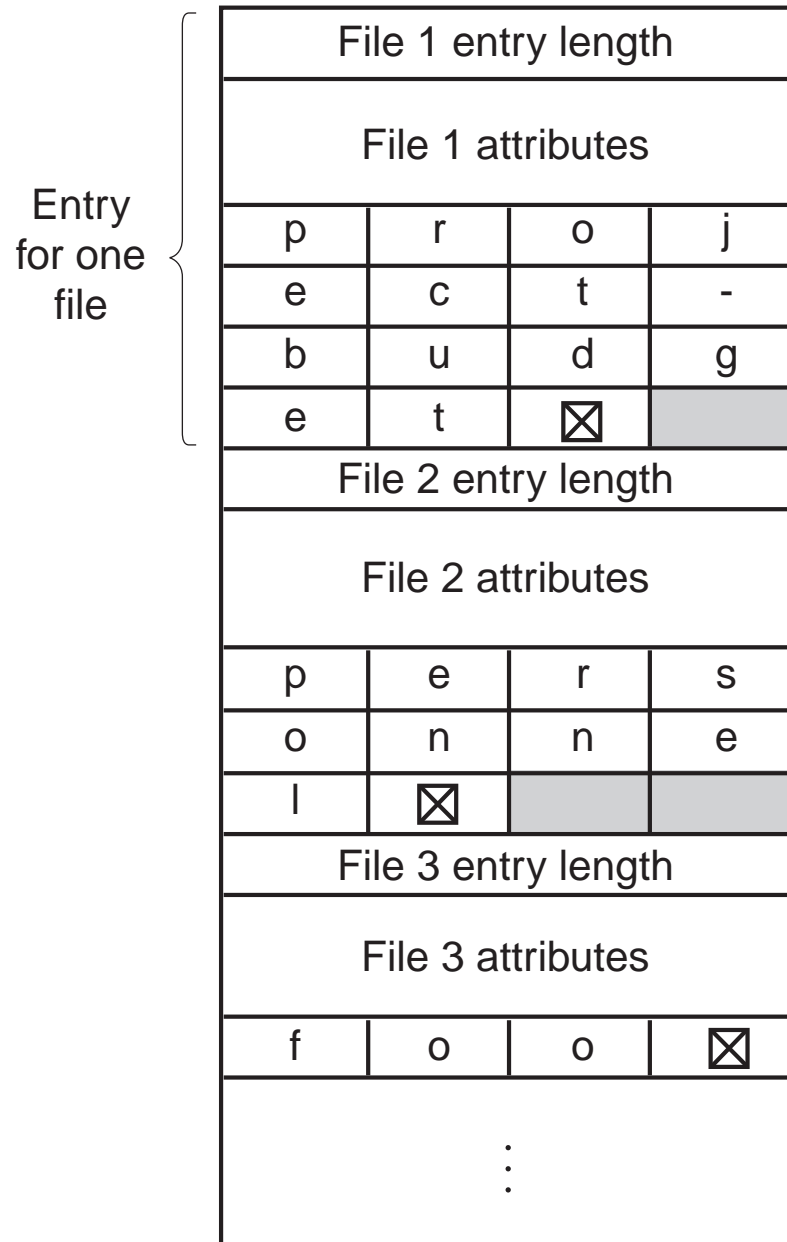


- a) Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows)
- b) Gli attributi risiedono in strutture esterne (eg. inode dei file), e nelle directory ci sono solo i puntatori a tali strutture (UNIX)

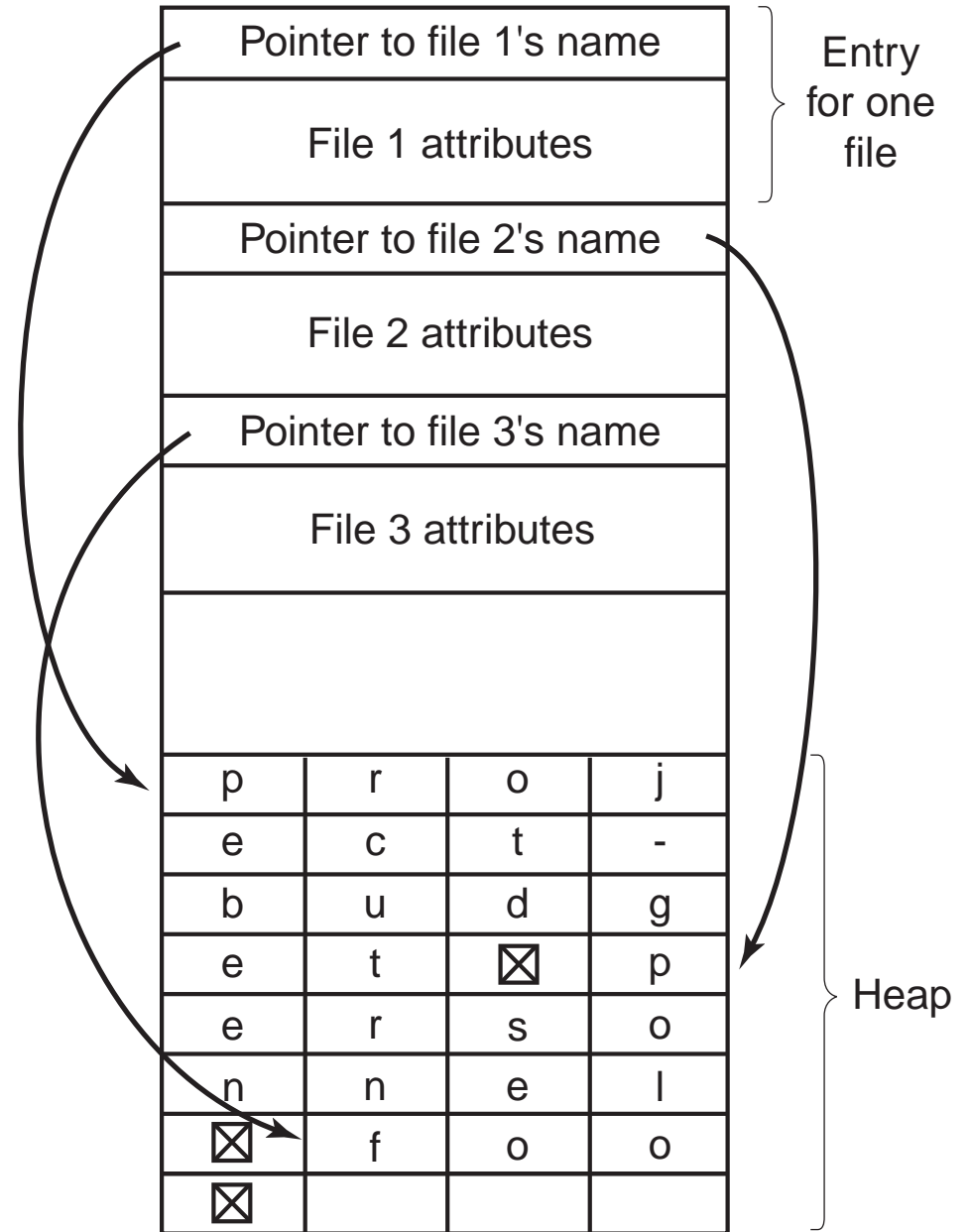
Gestione dei nomi

- Fino ad ora abbiamo supposto che i nomi dei file siano brevi e di lunghezza 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 può 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 attributi del file; tutti i nomi vengono memorizzati insieme in un *heap* alla fine della directory

Dimensioni variabili e heap



(a)



(b)

Directory: liste, hash, B-tree

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

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* (cioè 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

01011101010101111110110000001010000000101101010111100

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

- 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 smarkano tutte le directory che *non* contengono (ad una qualsiasi profondità nel corrispondente sottoalbero) file modificati
- Nota: i-node marcato → 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/giorgio/Articoli 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 contraddittorie.
- 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

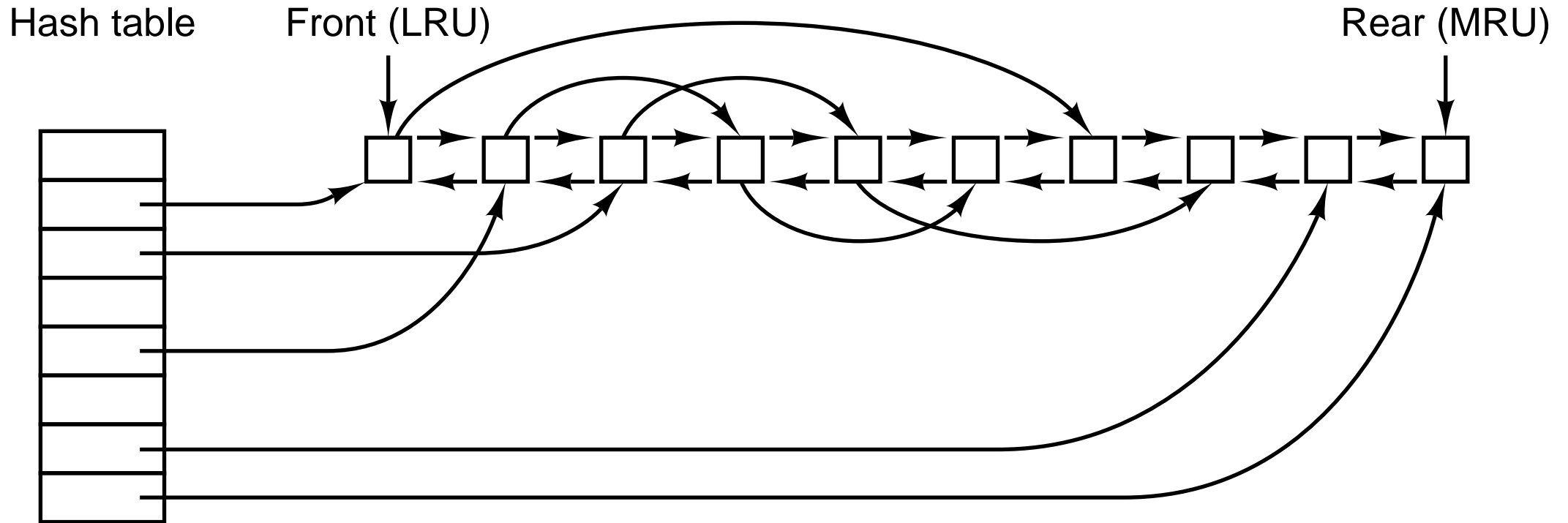
- *read-ahead*: leggere blocchi in cache *prima* che siano realmente richiesti.
 - Aumenta il throughput del device
 - Molto adatto a file che vengono letti in modo sequenziale; Inadatto per file ad accesso casuale (es. librerie)
 - Il file system può tenere traccia del modo di accesso dei file per migliorare 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 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).