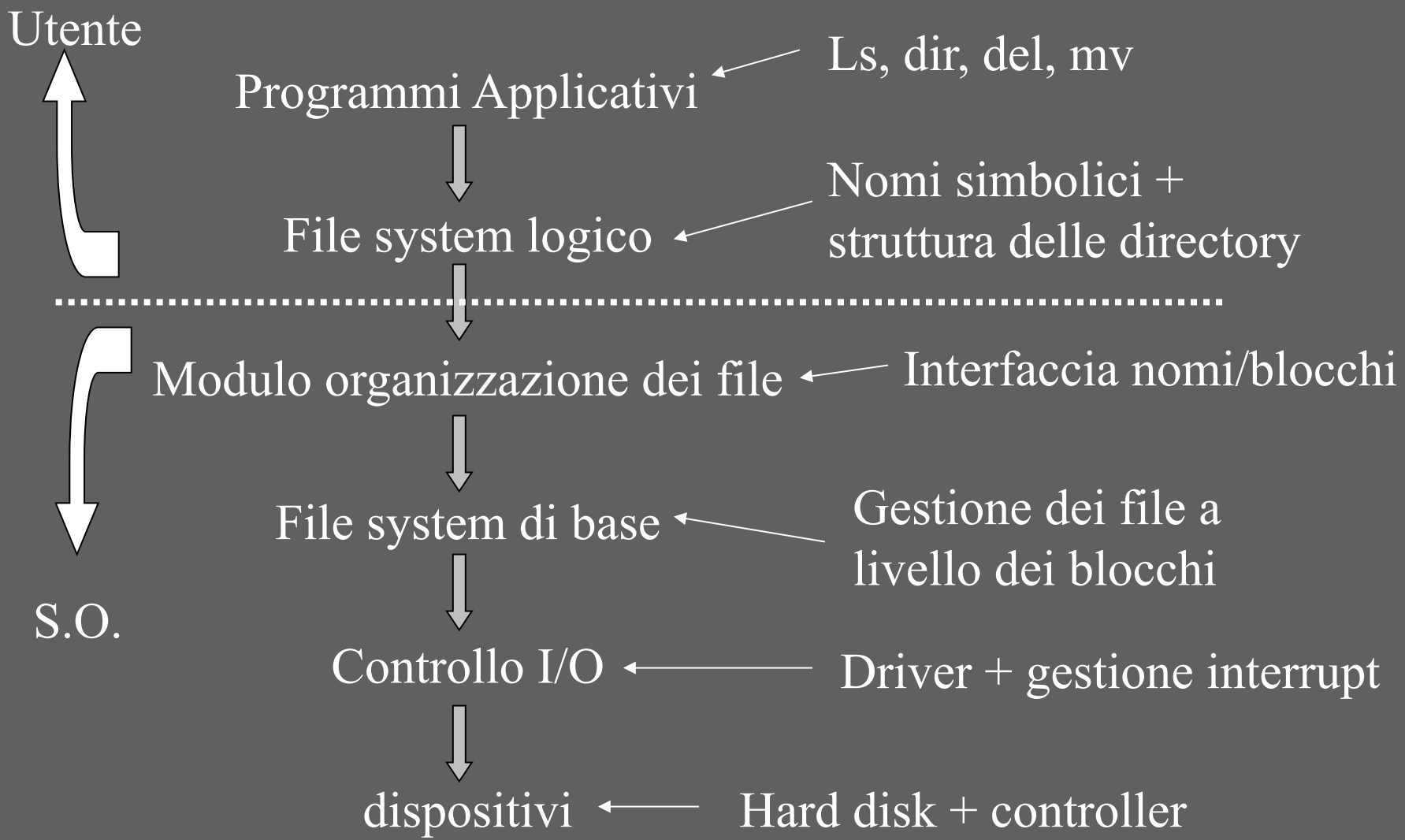
14 Realizzazione del File System

- Metodi di allocazione
- Allocazione contigua
- Allocazione concatenata e varianti
- Allocazione indicizzata e varianti
- Gestione dello spazio libero
- I link Unix

14.1 Struttura del file system (fig. 14.1)



14.4 Metodi di allocazione dei file

- (NB: In questo capitolo faremo riferimento principalmente all'hard disk come dispositivo di memoria di massa su cui sono memorizzati i file che compongono il file system)
- Come già detto nel capitolo 11, il SO vede l'HD come un enorme array in cui ogni entry è una sequenza di (ad esempio) 1024 byte (ma altri valori sono possibili, da 256 a 4096 byte)
- Il SO accede ogni blocco (letto, scritto) semplicemente comunicando al controller del disco il numero del blocco interessato, e le operazioni di lettura e scrittura su HD avvengono sempre in unità di blocchi.

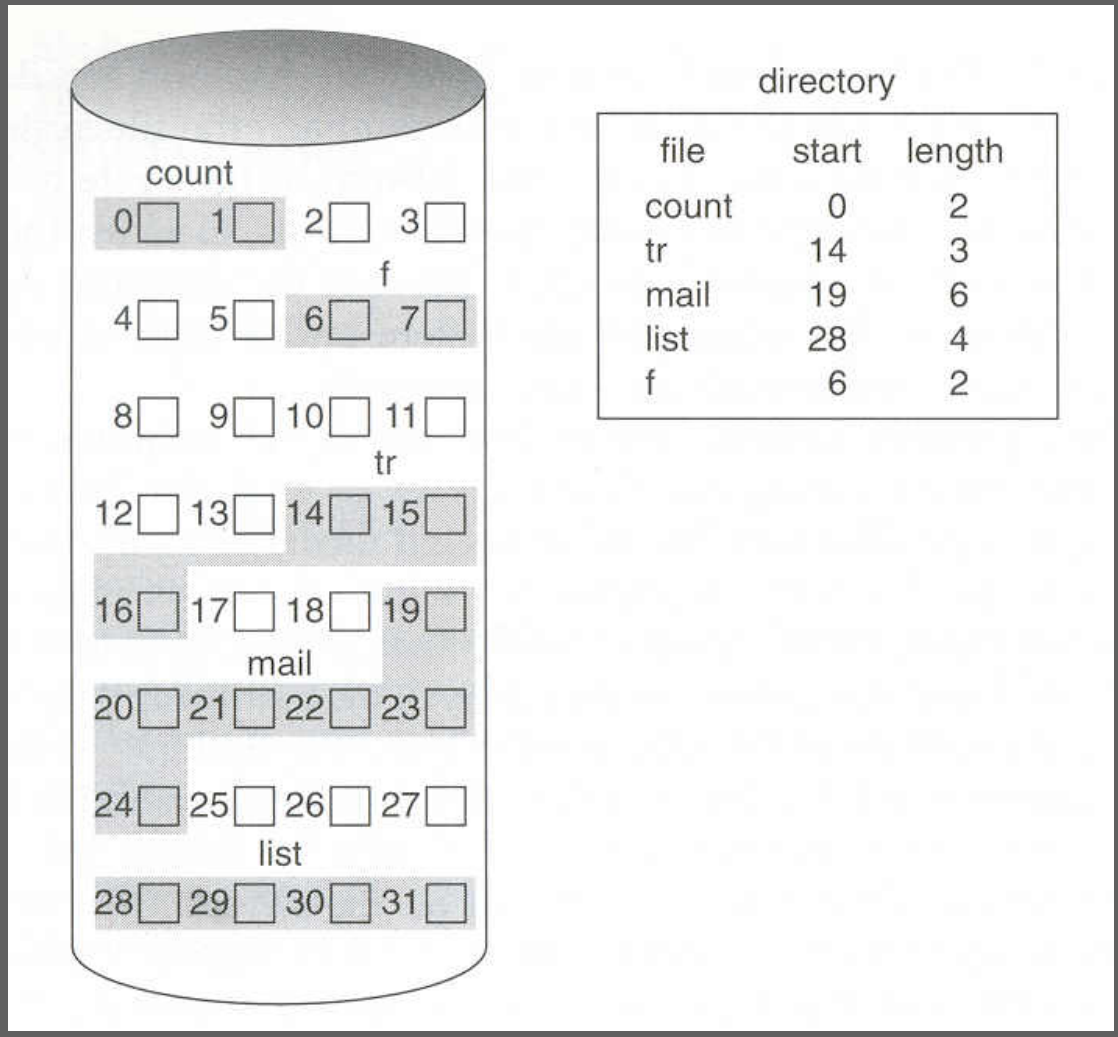
14.4 Allocazione dei file

- Il SO memorizza ogni file nei blocchi dell'HD. Se il file supera la dimensione di un blocco, allora viene distribuito su più blocchi.
- Per ogni file il SO mantiene anche una struttura interna che memorizza tutti gli attributi del file, necessari per gestire le operazioni compiute sul file (vedremo più avanti).
- Esistono fondamentalmente tre modi di base di allocare spazio sull'HD per i file:
  - allocazione **contigua**
  - allocazione **concatenata**
  - allocazione **indicizzata**

14.4.1 Allocazione contigua

- Ogni file è allocato in un insieme di blocchi dell'HD **contigui**
- Per recuperare i dati del file basta memorizzare, tra gli attributi del file:
  - il numero del primo blocco del file
  - quanti blocchi contigui sono occupati dal file (questo non è nemmeno strettamente necessario, dato che viene anche sempre registrata, tra gli attributi del file, la dimensione del file)
- Questo metodo era usato nei sistemi IBM VM/CMS

14.4.1 Allocazione contigua (fig. 14.4)



14.4.1 Allocazione contigua: vantaggi

- L’accesso ai vari blocchi del file è veloce e semplice. In particolare, l’accesso diretto ad un qualsiasi blocco del file è efficiente: noto il blocco a partire dal quale è memorizzato il file, è facile calcolare quale blocco contiene l’n-esimo byte del file
- Poche informazioni sono sufficienti a registrare completamente quale parte dell’HD è occupata dal file. Basta memorizzare il numero del blocco di partenza del file e la sua lunghezza.

14.4.1 Allocazione contigua: problemi

- Ma l'allocazione contigua presenta purtroppo molti difetti (simili agli svantaggi dell'allocazione contigua della MP)
- Per allocare un file è necessario trovargli uno spazio libero sul disco che sia costituito da una serie di blocchi **adiacenti**
- È necessario adottare una strategia di scelta del buco libero in cui memorizzare un file (first/best/worst fit)
- Il disco è soggetto **a frammentazione esterna**
- A lungo andare sarà necessaria una **ricompattazione** periodica del disco (molto più costosa della ricompattazione della MP)

14.4.1 Allocazione contigua: problemi

- Inoltre, se anche troviamo spazio sufficiente per allocare un file, che succede se la sua dimensione aumenta?
- siamo costretti a:
  - **riallocarlo** (spostarlo), ma è costoso, perché si tratta di leggere e riscrivere iblocchi dell'HD.
  - In alternativa, potremmo **sovradimensionare** il file: assegnarli diciamo (10 blocchi) anche se ne occupa solo 5 ma allora si aggrava il problema della frammentazione interna (comunque presente a causa dell'ultimo blocco)
- In definitiva, si presentano dei problemi simili a quelli visti con l'allocazione contigua in MP

14.4.2 Allocazione concatenata

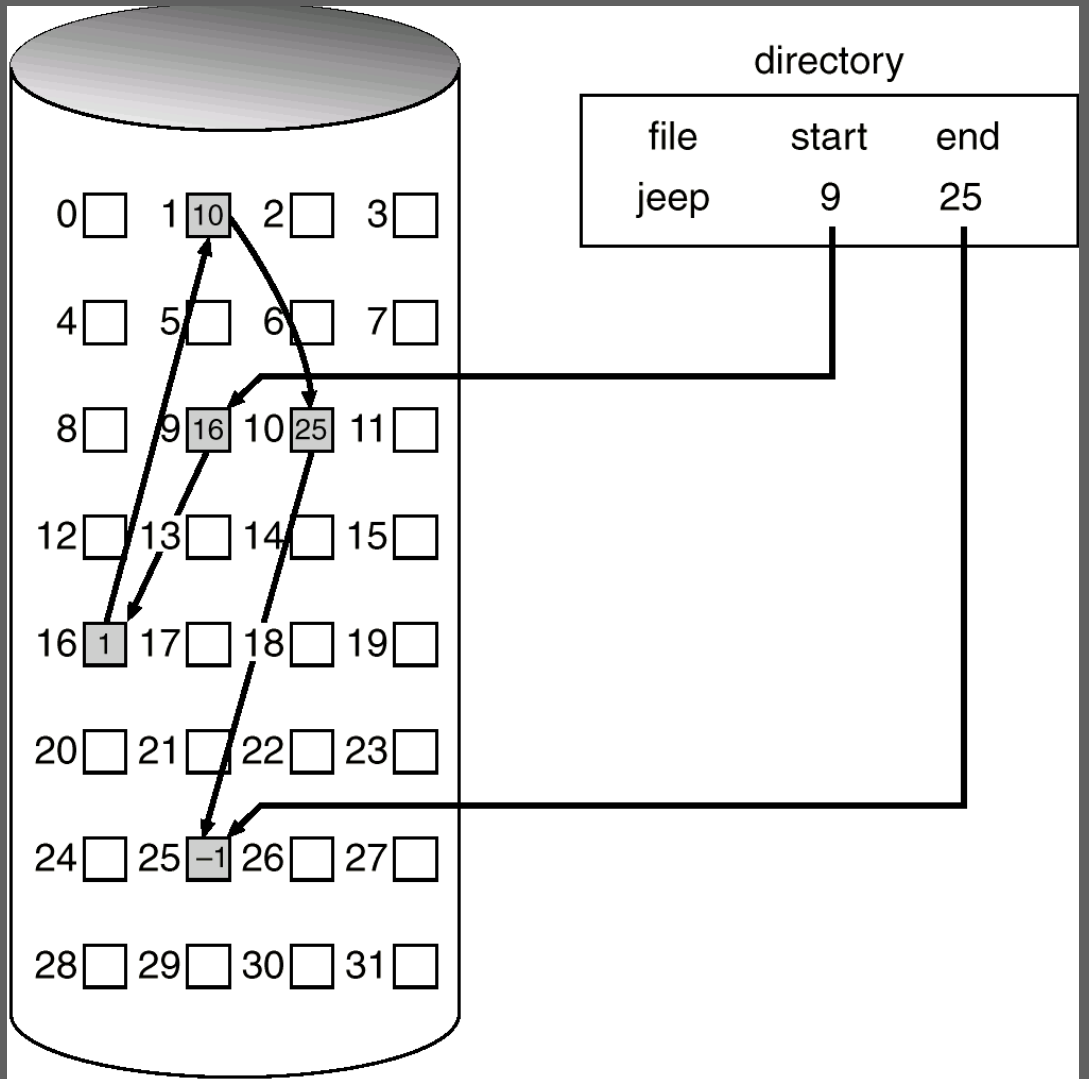
- Dato che anche in Memoria Secondaria l'allocazione contigua dà problemi, dobbiamo ricorrere a soluzioni simili a quelle viste per l'allocazione dello spazio in Memoria Primaria
- Nella **allocazione concatenata**, costruiamo una catena di blocchi contenenti i dati del file: ogni blocco conterrà (negli ultimi byte) un puntatore al blocco successivo: il numero del blocco in cui prosegue il file
- Per risalire ai vari blocchi che contengono il file basta memorizzare, tra gli attributi del file, il numero del blocco iniziale

14.4.2 Allocazione concatenata

- Opzionalmente, si può memorizzare anche il numero di blocchi usati e/o il numero del blocco finale

- Nell'ultimo blocco, nello spazio riservato per memorizzare il numero del blocco successivo viene scritto un numero negativo (o anche il valore 0), appunto per ricordarsi che quello è l'ultimo blocco del file

11.4.2 Allocazione concatenata (esempio, fig. 14.5)



14.4.2 Allocazione concatenata: vantaggi

- Non sono necessari blocchi contigui, per cui:
  - non c'è bisogno di ricompattare il disco per far spazio a nuovi file o a file che aumentano di dimensione
  - non si verifica frammentazione esterna
- Qualsiasi blocco libero può essere utilizzato per memorizzare una porzione del file

14.4.2 Allocazione concatenata: problemi

- Ma l'allocazione concatenata presenta anche alcuni problemi.
- Un problema (non troppo grave) è che in ogni blocco, gli ultimi byte sono utilizzati per memorizzare il puntatore al blocco successivo
- Nel caso di blocchi da 512 byte, con 4 byte per puntatore, circa lo 0,78% di un blocco non può essere utilizzato per memorizzare dati del file
- É uno spreco tutto sommato accettabile, dato che gli hard disk moderni offrono capacità notevoli a costi molto bassi

14.4.2 Allocazione concatenata: problemi

- Ben più grave è il fatto che **l'accesso diretto** al file può essere **altamente inefficiente:**
- in generale sono necessari n accessi al disco per accedere all'n-esimo blocco: occorre leggere il primo blocco per sapere dove si trova il secondo, leggere il secondo per sapere dove si trova il terzo, e così via fino all'n-esimo
- Inoltre, **il sistema di allocazione è poco affidabile:**
- se un blocco del file viene danneggiato, si perde tutta la parte di file a partire dal blocco danneggiato in avanti

14.4.2 Allocazione concatenata: Possibili soluzioni ai problemi

- **usare liste doppiamente concatenate:** se si danneggia un blocco possiamo recuperare i successivi ripercorrendo la catena all'indietro (ovviamente occorre memorizzare, tra gli attributi del file, anche il numero dell'ultimo blocco del file)

- **memorizzare in ogni blocco il nome del file e la posizione del blocco all'interno della catena:** scandendo tutti i blocchi del disco, possiamo recuperare quelli persi (al costo di sprecare un po' di spazio in più in ogni blocco)

14.4.2 Allocazione concatenata. Possibili soluzioni ai problemi

- Non usare blocchi singoli, ma **considerare tutto l'HD formato da cluster di blocchi adiacenti**
- Ad esempio, ciascun cluster puo' essere composto da 4 blocchi del disco adiacenti, e avere quindi una dimensione 4 Kbyte (se i singoli blocchi sono da 1024 byte).
- in pratica è come lavorare con blocchi di dimensione più grossa, per cui:
  - i tempi di accesso al file diminuiscono, perchè dobbiamo riposizionare meno volte la testina di lettura
  - meno spazio viene sprecato per i puntatori, però:
  - aumenta la frammentazione interna

14.4.2 Allocazione concatenata: la variante della FAT

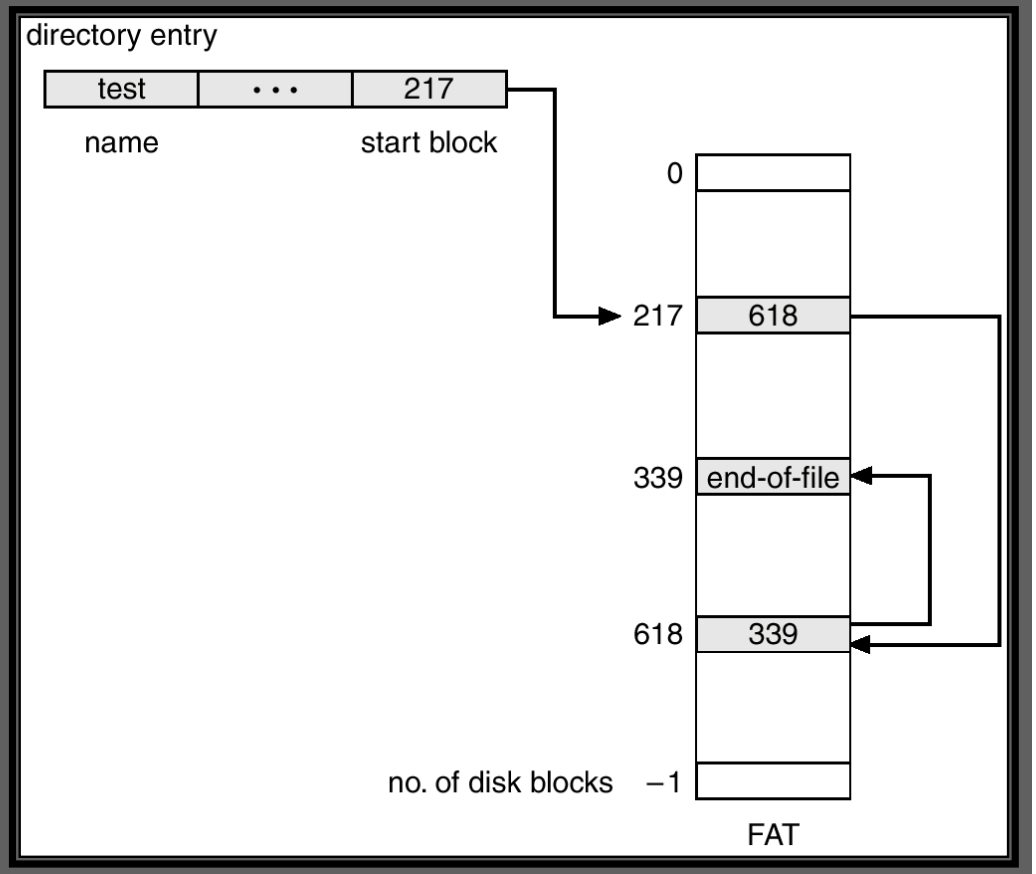
- Sebbene l'allocazione concatenata abbia gravi difetti, una sua variante molto efficiente era comunemente adottata dei sistemi ms-dos, OS2 ed è ancora disponibile nei sistemi Windows come alternativa a NTFS (che vediamo più avanti)
- **File Allocation Table (FAT):** area all'inizio del disco (in pratica è un array) in cui l'indice di ogni entry corrisponde ad un blocco, e contiene il numero di un blocco
- Le entry che contengono il valore zero corrispondono a blocchi del disco liberi

14.4.2 Allocazione concatenata: la variante della FAT

- La FAT registra lo stato di allocazione di tutti i blocchi dell'Hard Disk, e riproduce la lista concatenata di blocchi di ogni file
- Se il blocco I di un file punta al blocco N, allora nella FAT l'I-esima entry contiene il numero N
- Se l'ultimo blocco del file è il numero J, la J-esima entry della FAT contiene un marker speciale di fine file

14.4.2 un esempio di FAT

- Per far funzionare tutto, tra gli attributi del file “test” basta memorizzare il numero del primo blocco che contiene i dati del file (217).
- Per sapere quali altri blocchi contengono i dati del file “test” basta usare il numero del primo blocco come indice nella FAT, e poi seguire la catena di puntatori **all’interno della FAT** (fig. 14.6)



14.4.2 FAT: vantaggi

- Per poter usare in maniera efficiente la FAT, questa deve essere costantemente tenuta in RAM
- In questo modo, l'accesso diretto ai file migliora sensibilmente: è vero che per risalire all'n-esimo blocco di un file bisogna ancora seguire una catena di puntatori, ma invece di seguire la catena concatenata dei blocchi su disco la si può percorrere nella FAT, ossia leggere celle di RAM
- Il sistema è più sicuro in caso di perdita di un blocco, perché la catena di blocchi è riprodotta nella FAT
- La gestione dei blocchi liberi è automatica: ogni entri della FAT che contenga uno 0 rappresenta un blocco libero.

14.4.2 FAT: svantaggi

- La FAT occupa spazio in MP: è un array con un numero di entry uguale al numero di blocchi sul disco, e ogni entry deve poter contenere il numero di un blocco
- Inoltre, durante l'uso, la FAT va tenuta in RAM, sottraendo spazio ai processi
- Quindi l'uso della FAT diventa tanto più problematico quanto più aumentano le dimensioni medie degli HD (l'uso di cluster può mitigare parzialmente questo problema)
- Se la FAT viene persa non c'è più modo di accedere ai dati dei vari file, per cui deve essere frequentemente salvata sull'hard disk

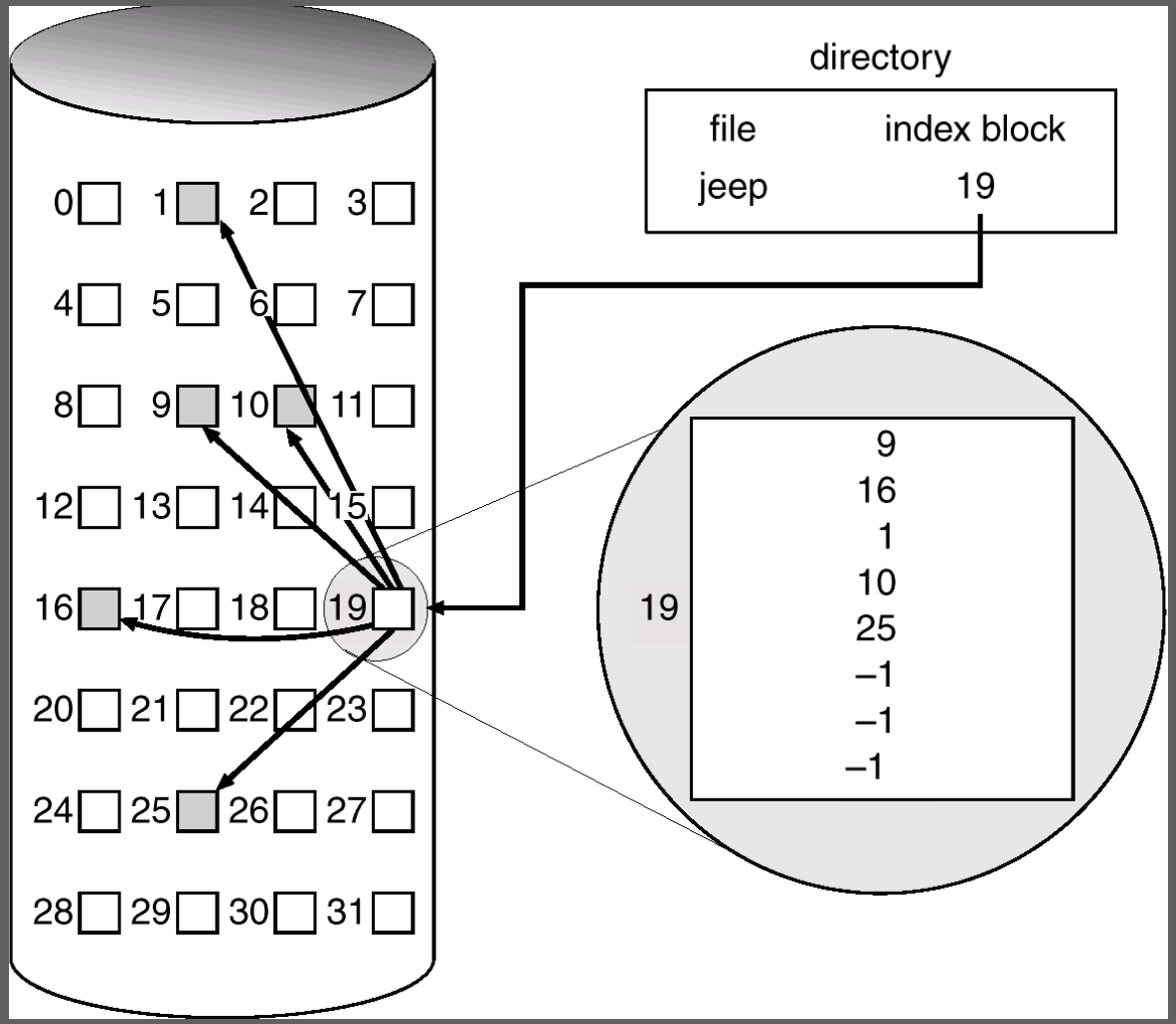
l'overhead di memoria della FAT

- Provate a calcolare quanto spazio è necessario per memorizzare la FAT di un hard disk con queste caratteristiche:
  - capacità dell'hard disk: 20 giga byte
  - dimensione di un blocco dell'hard disk: 1 kilo byte
- Qual è la percentuale di spazio del disco "sprecata" per memorizzare la FAT?
- Per risolvere l'esercizio, di quanti byte abbiamo bisogno per memorizzare il numero di un blocco dell'hard disk?

14.4.3 Allocazione indicizzata

- Teniamo direttamente traccia di tutti i blocchi in cui è contenuto un file scrivendo il loro numero in un altro blocco del disco (detto **blocco indice**).
- Per recuperare i dati del file basta memorizzare, tra gli attributi del file, il numero del suo blocco indice
- osservate che questo approccio è simile alla gestione della memoria primaria nei sistemi paginati: il blocco indice funziona come una sorta di page table

14.4.3 Allocazione indicizzata (esempio, fig. 14.7)



11.4.3 Allocazione indicizzata: vantaggi e svantaggi

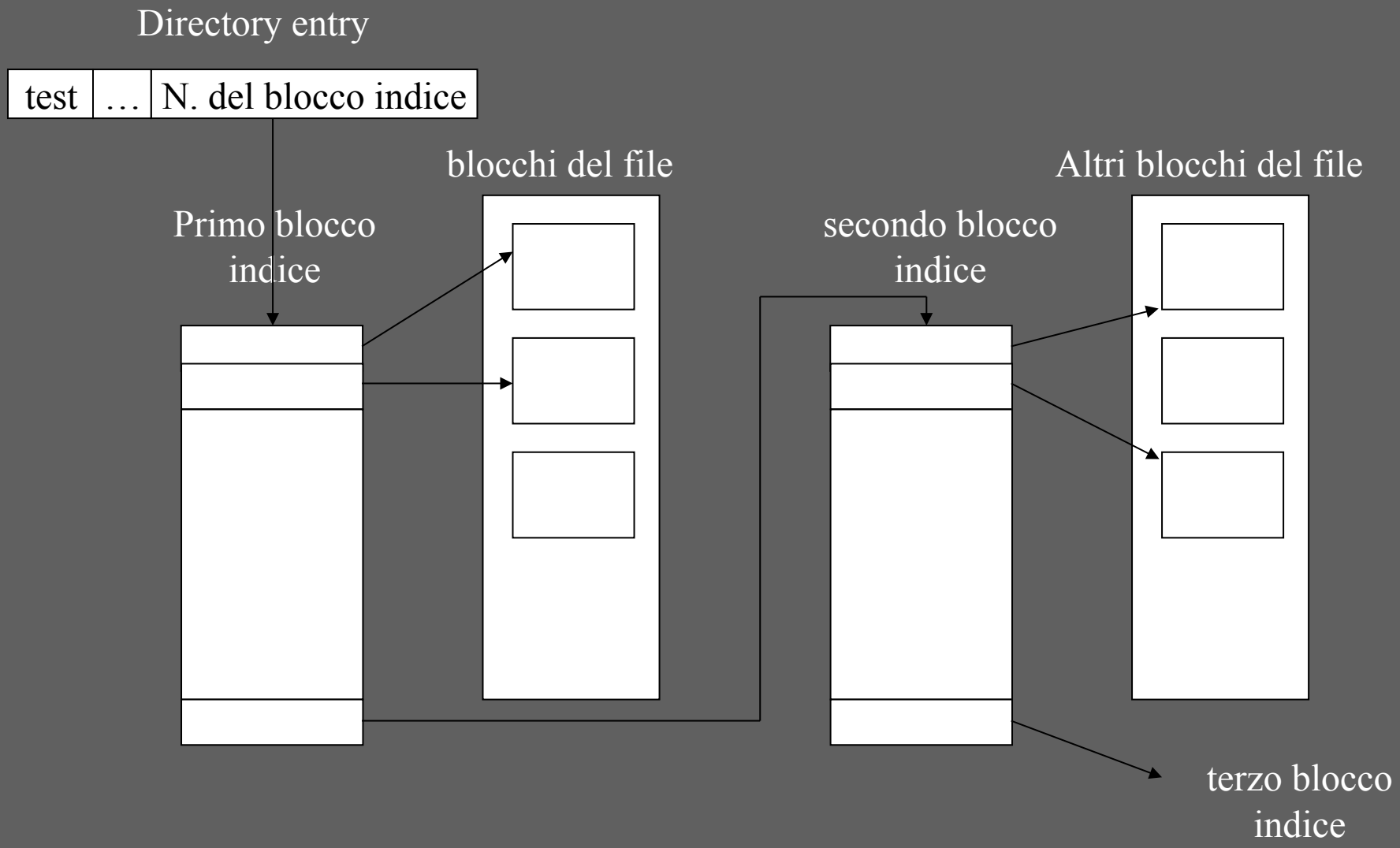
- Come per l'allocazione concatenata, non sono necessari blocchi contigui, per cui non si crea frammentazione esterna
- L'accesso diretto ad un qualsiasi byte del file è efficiente: una volta portato in RAM il blocco indice è facile calcolare quale blocco dell'elenco contiene il byte desiderato
- Tuttavia, uno svantaggio fondamentale è che un intero blocco indice deve sempre essere utilizzato per memorizzare il numero dei blocchi di dati del file: se il file è piccolo il blocco indice è quasi tutto vuoto, e sprecato...

14.4.3 Allocazione indicizzata: problemi

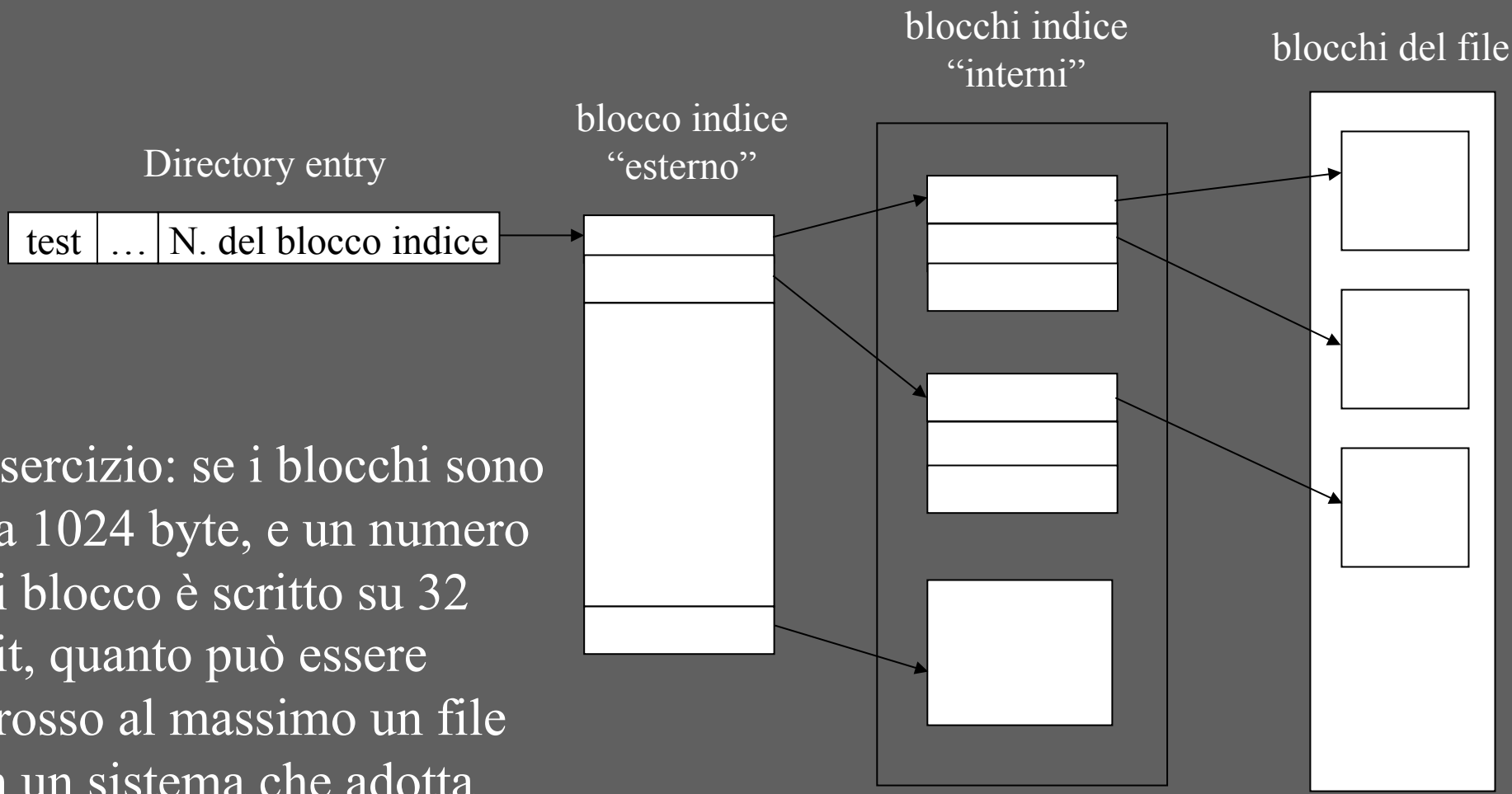
- E' comunque necessario risolvere un problema: che succede se un blocco indice non è sufficiente per memorizzare i numeri di tutti i blocchi dati del file?
- Ci sono due soluzioni si base possibili:

1. l'ultima entry del blocco indice punta ad un secondo blocco indice (**schema concatenato**)
2. il blocco indice contiene solo puntatori ad altri blocchi indice (**schema a più livelli**)

14.4.3 Allocazione indicizzata: lo schema concatenato



14.4.3 Allocazione indicizzata: lo schema a più livelli



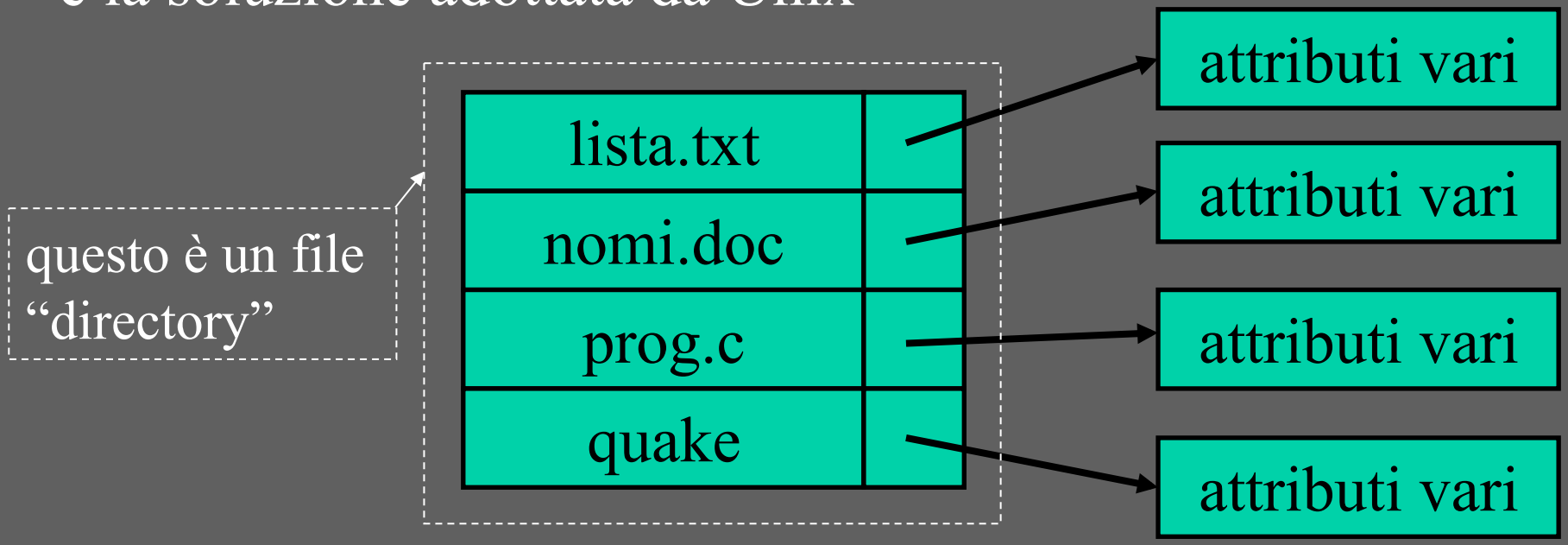
Esercizio: se i blocchi sono da 1024 byte, e un numero di blocco è scritto su 32 bit, quanto può essere grosso al massimo un file in un sistema che adotta questo schema?

14.4.3 Gli i-node Unix

- Una variante della allocazione indicizzata è usata in Unix: ad ogni file è associato un i-node (index-node) che contiene gli attributi del file, e l'elenco dei blocchi di dati del file
- Gli i-node sono gestiti direttamente dal SO e sono memorizzati permanentemente sull'hard disk in una porzione riservata al SO, di solito nella prima parte dell'hard disk stesso.
- Semplificando un po', possiamo pensare che i primi blocchi dell'hard disk vengano appunto usati per contenere ciascuno l'i-node di un file. Il numero del blocco indice (ossia dell'i-node) di un file viene scritto a fianco del nome del file nella directory che lo "contiene".

13.3 Le Directory

- (Ricordate questa slide?) Alternativamente, possiamo inserire, a fianco del nome di ogni file solo un puntatore ad una struttura interna, anch'essa memorizzata in modo permanente sull'hard disk, e gestita direttamente dal SO, in cui sono contenute tutte le informazioni su quel file: questa è la soluzione adottata da Unix

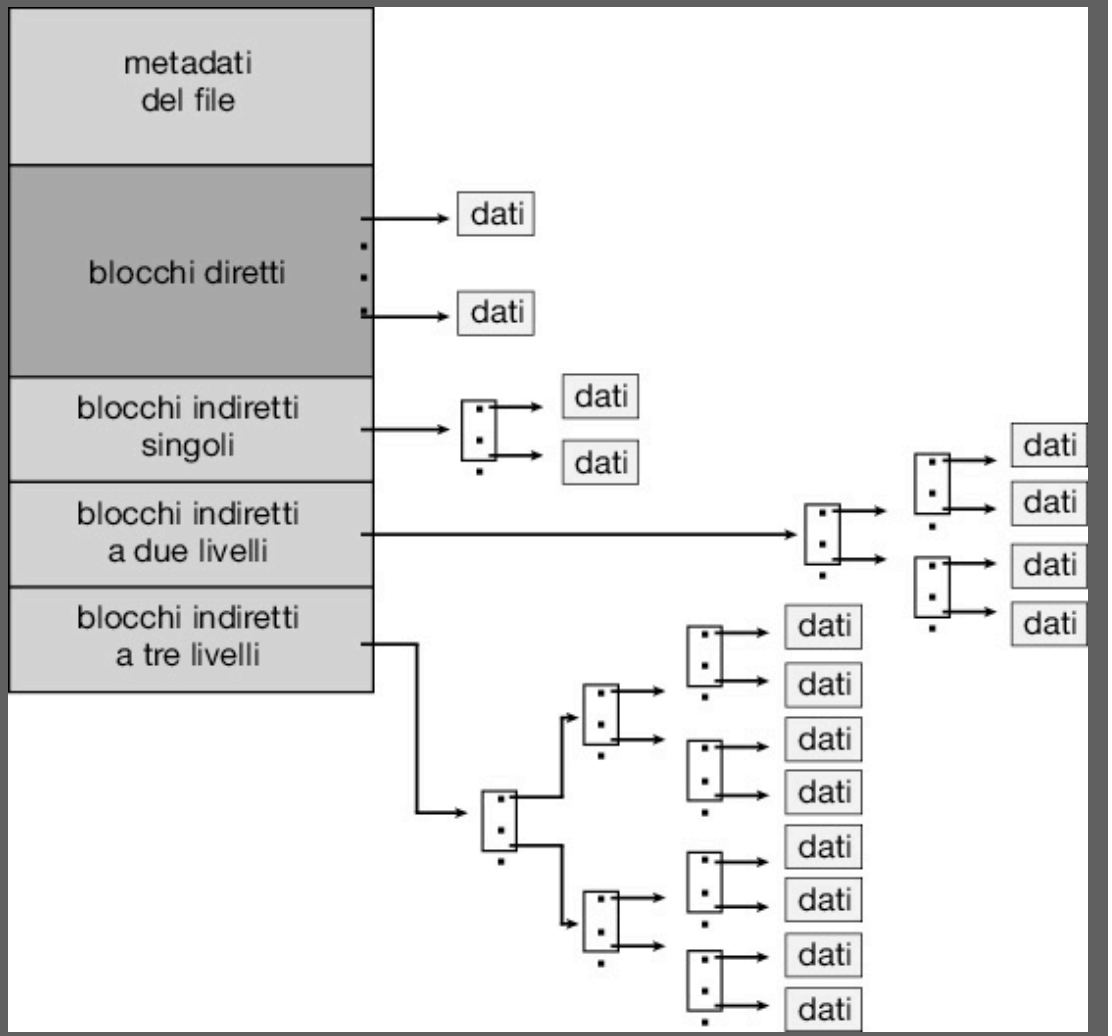


14.4.3 Gli i-node Unix

- Per tenere traccia di tutti i blocchi di dati del file, ogni i-node contiene lo spazio per memorizzare (cfr slide successiva):
  - 10 puntatori **diretti** a blocchi di dati del file.
  - un puntatore **single indirect** che punterà ad un blocco indice che conterrà puntatori a blocchi di dati file
  - un puntatore **double indirect** che punterà ad un blocco indice che conterrà puntatori a blocchi indice ognuno dei quali conterrà puntatori a blocchi di dati del file
  - Un puntatore **triple indirect** che punterà ad un blocco indice che conterrà puntatori a blocchi indice ognuno dei quali conterrà puntatori a blocchi indice, ognuno dei quali conterrà puntatori a blocchi di dati del file

14.4.3 Allocazione indicizzata: gli i-node Unix (fig. 14.8)

Se i blocchi sono da 1024 byte e un numero di blocco è scritto su 32 bit, quanto può essere grosso al massimo un file in un sistema Unix che adotta questo schema?



MaxSize = $10 \times 1024 + 256 \times 1024 + 256^2 \times 1024 + 256^3 \times 1024$ byte

14.4.3 NTFS

- Un approccio non molto diverso è usato nel file system adottato da Windows (da XP in poi):
  NTFS, da **New Technology File System.**
- Ogni file è descritto da un **elemento**. Gli elementi di un file system NTFS hanno dimensione fissa (configurabile da 1 a 4 Kbyte all'atto della creazione del file system) e sono numerati consecutivamente.
- Tutti gli elementi di un file system sono contenuti in una **master file table** (MFT): un file memorizzato nei primi blocchi del disco e gestito esclusivamente dal SO.
- Il numero dell'elemento è associato al nome del file nella directory che "contiene" quel file, ed è detto **file reference**

14.4.3 NTFS

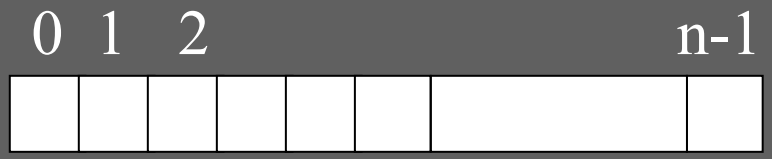
- Un elemento contiene gli attributi del file corrispondente: il proprietario del file, permessi di accesso, dimensioni correnti, date di creazione, accesso, modifica, ecc.
- Se il file è molto piccolo, anche i dati del file possono essere direttamente contenuti nel corrispondente elemento, in questo modo l'accesso al file risulta molto efficiente (con la sola lettura dell'elemento possiamo accedere sia agli attributi che ai dati del file)
- Altrimenti, l'elemento contiene più puntatori a cluster del disco che contengono dati del file, ed eventualmente almeno un puntatore ad un altro blocco di puntatori (si usa di fatto una allocazione indicizzata a schema concatenato)

14.5 Gestione dello spazio libero

- Il SO deve tenere anche traccia di tutti i blocchi liberi del disco, utilizzando una opportuna struttura dati.
- In Unix queste informazioni sono memorizzate nel **superblocco**, contenuto in alcuni blocchi iniziali dell'HD, a partire dal blocco numero 1. Nei sistemi Windows queste informazioni stanno nella MFT.
- Per creare o estendere un file, si cercano blocchi liberi nell'elenco e si allocano al file
- Quando un file viene cancellato i suoi blocchi sono reinseriti nell'elenco dei blocchi liberi

14.5.1 Vettore di bit

- usiamo un bit per ogni blocco. Il valore del bit indica se il blocco è occupato o libero



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

- E' il sistema usato da MAC/OS

14.5.1 Vettore di bit

- Naturalmente il vettore di bit richiede spazio aggiuntivo. Ad esempio:

  block size = 2^9 bytes

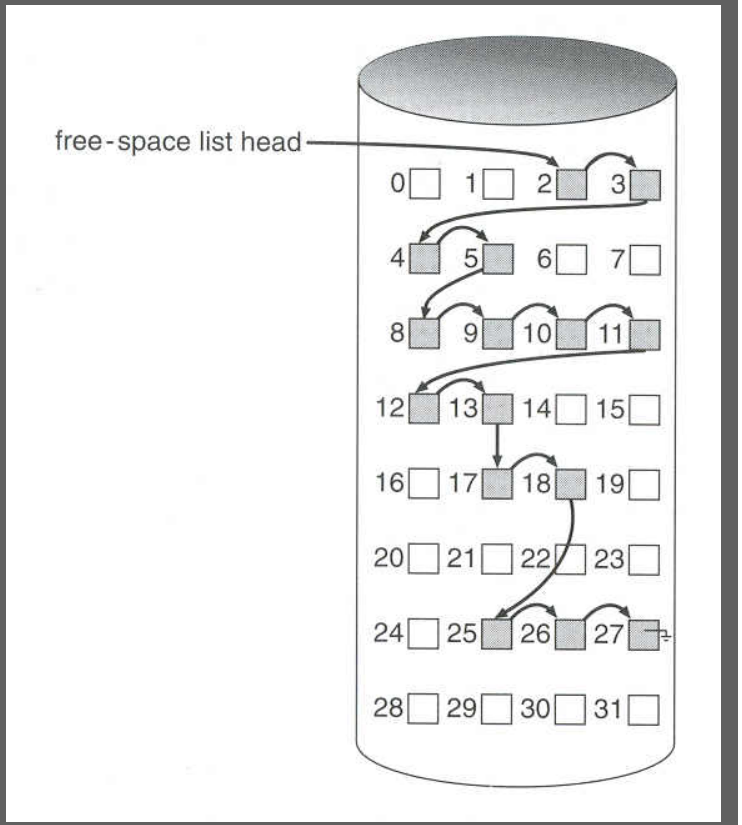
  disk size = 2^{35} bytes (32 gigabyte)

  $n = 2^{35}/2^9 = 2^{26}$ bit (ossia 8 megabyte)

- Perchè la gestione sia efficiente, l'intero vettore deve essere sempre tenuto in MP (e periodicamente salvato in MS)
- L'uso di cluster diminuisce lo spazio occupato: è sufficiente usare un bit per ogni cluser.

14.5.2 Lista concatenata

- Formare con i blocchi liberi una lista: ciascun blocco punta al successivo blocco libero
- Nessuno spreco di spazio
- la FAT può essere usata anche per gestire la lista dei blocchi liberi, altrimenti la ricerca di molti blocchi liberi sarebbe troppo inefficiente (fig. 14.9)



Altri metodi

- 14.5.3 Raggruppamento
  - Raggruppare in un blocco più puntatori a blocchi liberi. L'ultimo puntatore punta ad un altro blocco di blocchi liberi, e così via
  - Con questo metodo è facile trovare in fretta molti blocchi liberi. Funziona sostanzialmente come l'allocazione indicizzata
- 14.5.4 Conteggio
  - mantenere il numero di un blocco e quanti blocchi consecutivi liberi lo seguono
  - è un metodo simile alla lista concatenata ma con meno entry

14.6 Efficienza e prestazioni

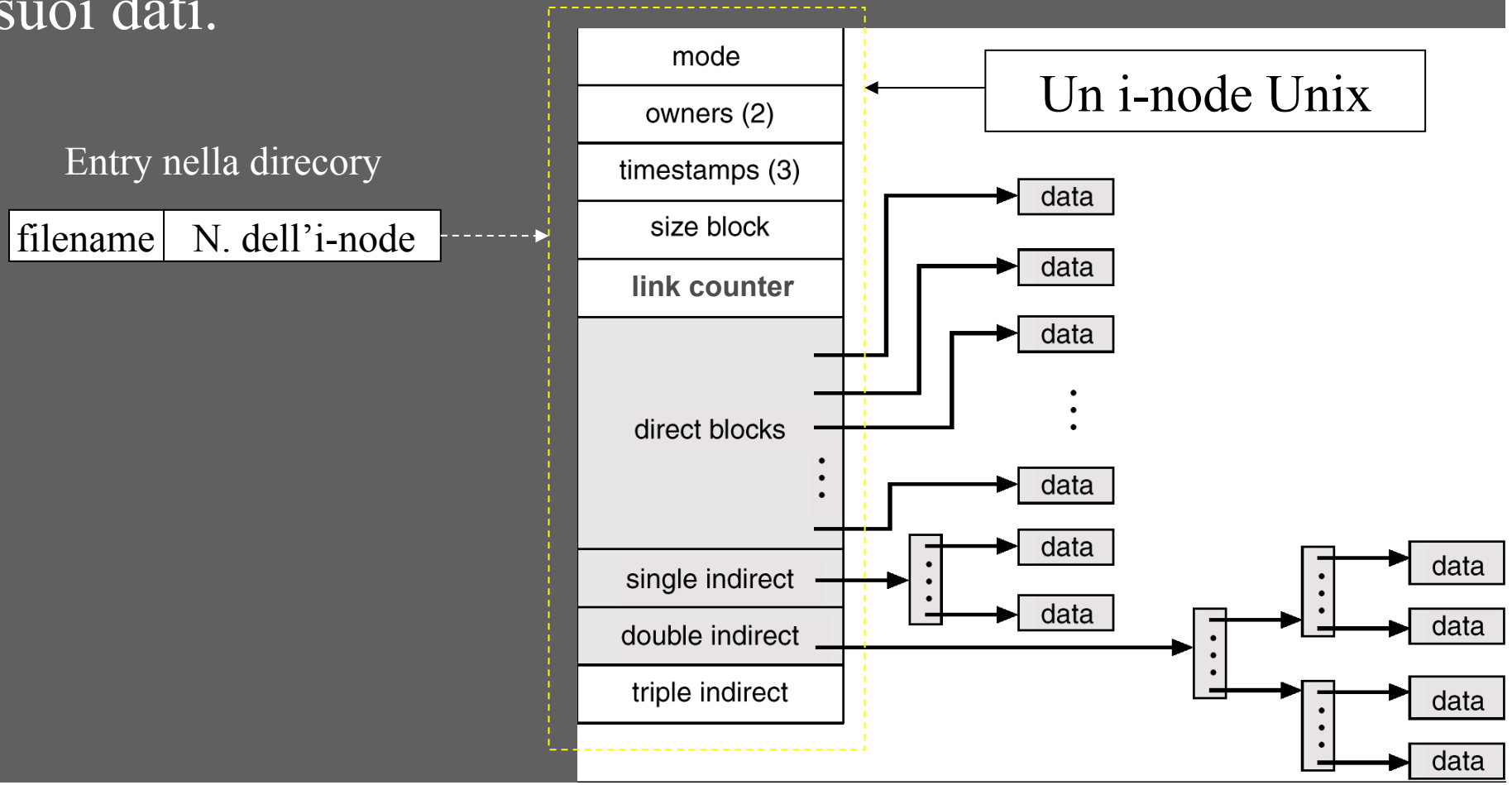
- A parte le osservazioni già fatte sulle prestazioni offerte dai vai metodi di allocazione dello spazio, l'efficienza nell'uso del FS dipende dal fatto che il SO implementi un sistema di caching in MP dei file (e dei relativi attributi) usati più di frequente e di recente, ed in particolare di tutti i file aperti.

- In questo modo, tutte le operazioni sui file e l'aggiornamento dei loro attributi viene fatto accedendo solo alla copia in RAM dei dati del file e dei suoi attributi.

- Sarà il SO ad occuparsi di mantenere la consistenza con le corrispondenti informazioni in Memoria Secondaria, salvando periodicamente le informazioni in RAM sul disco

I link Unix

- In Unix, un file è identificato univocamente dall'index-node che contiene tutte le informazioni relative al file: i suoi attributi e in quali blocchi sono memorizzati i suoi dati.



I link Unix

- Il numero dell'index-node rappresenta univocamente un file esattamente come un Process ID (PID) rappresenta univocamente un processo e un User ID (UID) rappresenta univocamente un utente del sistema.

- Ma poichè per gli utenti è scomodo identificare i propri file attraverso dei numeri, nella directory che "contiene" un file, al numero dell'i-node è associato il nome del file

- Quando l'utente specifica il nome di un file in una cartella, il SO legge il numero del corrispondente i-node, recupera questo i-node sul disco e può così accede a tutte le informazioni relative al file stesso

mydir:

21	prog.c
128	old_name
11	letter

gli hard link (link fisici)

- La stringa di caratteri scritta a fianco del numero di un i-node in una directory (ossia il nome del file) è ciò che, nella terminologia Unix, viene chiamato un **link fisico** o **hard link** al file.
- In Unix un file regolare (ossia, non una directory) può essere identificato da uno o più nomi, **ossia può avere uno o più link fisici**. In altre parole, possono esistere più entry diverse (nella stessa cartella o in cartelle diverse) in cui il numero dell'i-node specificato è lo stesso.
- L'unico vincolo è che una cartella non può contenere due entry con lo stesso nome e lo stesso numero di i-node (ma cartelle diverse si)

gli hard link (link fisici)

- In altre parole, una stessa cartella (mydir nell'esempio qui sotto) può contenere più entry con lo stesso i-node, ma non con lo stesso nome. (Notate, il "prog.c" di anotherdir è un file diverso dal "prog.c" di mydir, infatti i due hard link sono associati a i-node diversi)

mydir:

21	prog.c
128	**old_name**
11	letter

mydir:

21	prog.c
128	**old_name**
11	letter
128	**new_name**

anotherdir:

47	myfile.txt
128	**old_name**
19	quake
188	prog.c

gli hard link (link fisici)

- Una delle entry di un index-node è un campo intero chiamato **link counter**. Quando un file regolare (non una directory) viene creato e gli viene dato un nome (ossia viene associato ad un hard link) il suo link counter viene inizializzato ad 1.
- E' possibile creare un nuovo hard link ad un file in due modi: col comando **ln** e con la system call **link**:
- ln **existing_file_name new_file_name**
- link(**existing_file_name**, **new_file_name**)
- I due argomenti possono ovviamente anche contenere un pathname assoluto o relativo, ma *existing_file_name* **non può essere una directory**

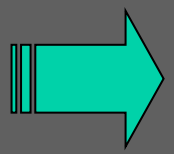
gli hard link (link fisici)

- In entrambe i casi, il sistema operativo:

1. recupera l'i-node di *existing_file_name* e controlla se l'utente chiamante ha i permessi per eseguire il comando/syscall.
2. Incrementa di uno il link-counter dell'i-node associato a *existing_file_name*. Infatti ora nel sistema c'è un hard link in più associato a quell'i-node
3. alloca una nuova entry nella directory specificata dal pathname di *new_file_name* associandogli l'i-node di *existing_file_name*

21	prog.c
128	**old_name**
11	letter

il link-counter dell'i-node 128 è incrementato di uno



21	prog.c
128	**old_name**
11	letter
128	**new_name**

gli hard link (link fisici)

- Il comando: **rm *nome_file*** e la system call **unlink(*nome_file*)** si comportano in maniera opposta. Il sistema operativo:

1. recupera l'i-node di *nome_file*, e controlla se l'utente chiamante ha i permessi per eseguire il comando/syscall (e ovviamente controlla anche se *nome_file* esiste).

2. rimuove la entry *nome_file* nella directory specificata

3. decrementa di uno il link-counter dell'i-node associato a *nome_file*. Infatti ora nel sistema c'è un hard link in meno che fa riferimento a quell'i-node

4. Se link-counter = 0 viene anche rimosso l'i-node e vengono recuperati i blocchi di dati del file. Infatti ora nel sistema non c'è più nessun nome/hard link che fa riferimento a quell'i-node.

gli hard link (link fisici)

- Notate che gli hard link ad uno stesso file sono indistinguibili fra loro: non importa quale è stato creato per primo.

```
$> echo “ciao a tutti!” > new_file     // crea un nuovo file, e quindi un
                                       // un nuovo i-node. link-counter=1
$> ln new_file another_file            // crea un secondo hard link allo
                                       // stesso file. link-counter=2
$> rm new_file                         // rimuove il primo hard link al file.
                                       // Ora link-counter=1
$> cat new_file
no such file or directory
$> cat another_file
ciao a tutti!
```

gli hard link (link fisici)

- Che i vari hard link ad un file siano indistinguibili fra di loro si vede facilmente usando il comando "ls –li":

```
$> echo "ciao a tutti!" > new_file
$> ls -li new_file
128 -rw-------   1 gunetti       13 Dec 27 17:23 new_file

$> ln new_file another_file
$> ls -li another_file
128 -rw-------   2 gunetti       13 Dec 27 17:23 another_file
```

notate il valore del link-counter

- A parte il valore del link-counter, che dopo l'uso di **ln** è incrementato di 1, i due link fisici sono completamente identici: infatti le informazioni stampate da ls vengono estratte dallo stesso i-node

gli hard link (link fisici)

- Notate anche che i comandi “ln” e “cp” sono diversi. Infatti:

$> **cp new_file another_file**

- Se *another_file* non esiste, questo comando alloca un nuovo i-node e lo associa al link fisico *another_file*. L’i-node associato a *another_file* **è diverso** dall’i-node associato al link fisico *new_file*
- Il link-counter dell’i-node associato ad *another_file* è inizializzato a 1
- Domanda: che succede se *another_file* è un file esistente? (provate su un qualsiasi sistema Unix…)

gli hard link (link fisici)

- Quando viene creata una directory (chiamiamola **newdir**) all'interno di un'altra directory (chiamiamola **parentdir**), la nuova directory non nasce vuota, ma contiene già due entry: "." e "..", come conseguenza:

1. Il link-counter di **newdir** viene inizializzato a 2, perché ci sono due link fisici associati all'i-node (l'i-node 47 nel nostro esempio) di **newdir**: la entry "." dentro **newdir** e la entry "**newdir**" nella directory padre **parentdir**.
2. Il link-counter di **parentdir** viene incrementato di 1, perché ora c'è un nuovo link fisico che fa riferimento all'i-node (52) di **parentdir**, ossia il link fisico ".." dentro **newdir**.

- Domanda: leggendo l'i-node di una cartella come facciamo a sapere quante sottocartelle contiene?

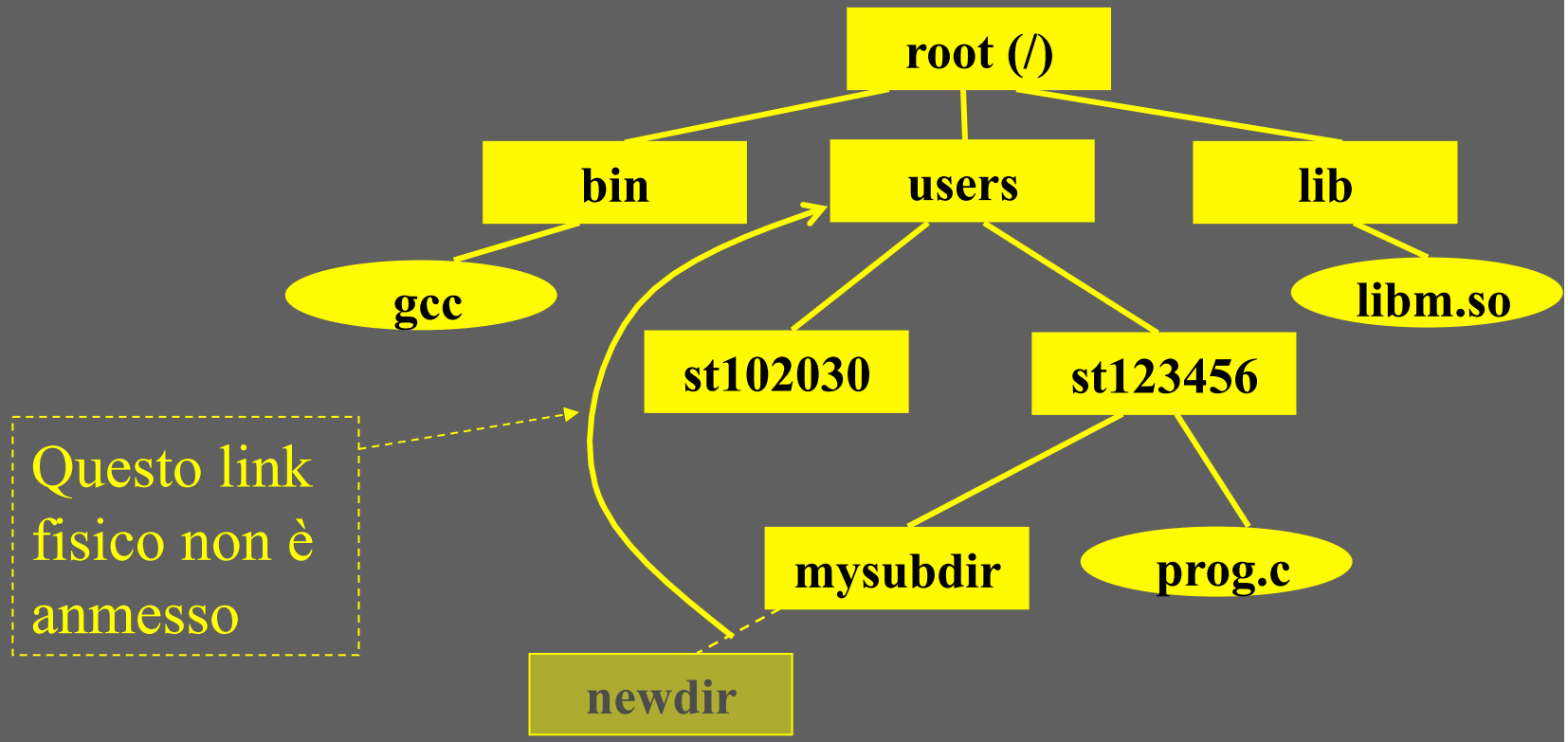
newdir:

47	.
52	..

gli hard link (link fisici)

- Unix proibisce la costruzione di link fisici tra directory perché questo potrebbe portare alla costruzione di file system a grafo generale, ossia che possono contenere cicli.

$> cd /users/st123456/mysubdir
$> ln /users newdir // **errore!**



gli hard link (link fisici)

- Infatti, poiché i link fisici ad uno stesso i-node sono indistinguibili fra loro, se il comando del lucido precedente fosse ammesso, dopo la sua esecuzione si creerebbe la seguente situazione:

root:

0	.
8	bin
11	users
15	lib

mysubdir:

58	.
47	..
11	newdir

- Se un programma visita ricorsivamente il file system, quando si trova in mysubdir vede che “contiene” newdir e ci entra dentro, aprendo il file cartella con i-node 11. Ma “dentro” questo file cartella c’è anche la sottodirectory st123456, e il programma ci entra dentro. E così via entrando in un loop infinito.

i symbolic link (link simbolici)

- (N.B.: per alcune varianti di sistemi Unix, ad esempio per Solaris, i link simbolici sono implementati in modo leggermente diverso da quanto descritto qui di seguito)
- Per permettere i link fra cartelle (e anche per un'altra ragione) Unix usa anche il concetto di **link simbolico** (noto anche come **soft link**):

$> **ln -s existing_directory new_directory** // notate l'opzione -s

- Questo tipo di link (che in realtà si può usare anche tra file che non siano directory) funziona esattamente come il "collegamento" dell'ambiente Windows. E' quindi legale un comando come:

$> ln -s /users newdir

i symbolic link (link simbolici)

- **I link simbolici vengono implementati allocando un nuovo i-node che viene associato al link simbolico**. Dentro questo i-node viene annotato il fatto che quello è l'i-node di un link simbolico, e in un campo apposito "Pathname" viene scritto il primo argomento del comando "ln -s", ossia il pathmane specificato.

$> ln -s /users newdir

root:

2	.
2	..
8	bin
11	users
15	lib

mysubdir:

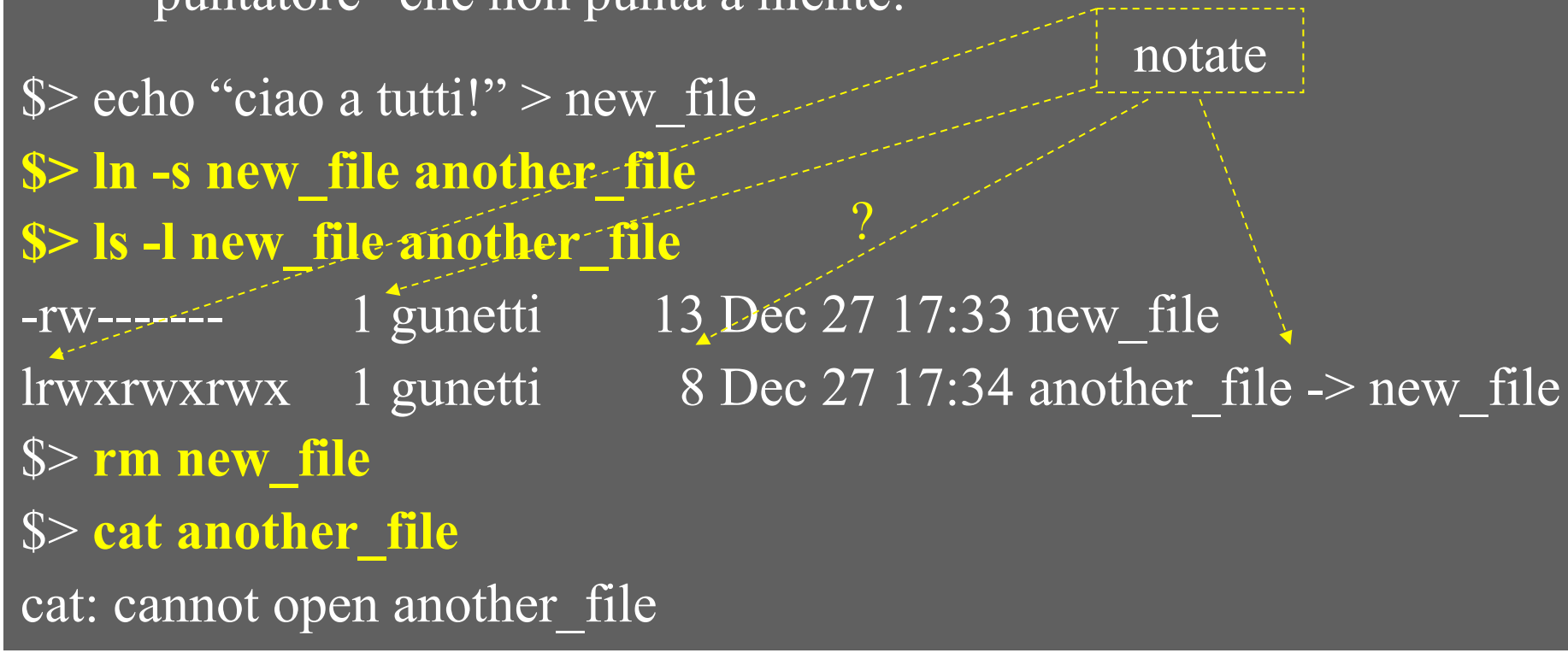
58	.
47	..
91	newdir

Index-node **91**:

Type: **symbolic link**
Pathname: **/users**
other attributes…
other attributes…

I symbolic link (link simbolici)

- Poiché i link simbolici sono distinguibili dai link fisici, i comandi che visitano ricorsivamente il file system evitano di visitare sottocartelle che sono in realtà link simbolici ad altre cartelle
- Un link simbolico non incrementa il link-counter del file sorgente, e se si rimuove il file sorgente di un link simbolico, rimane un "puntatore" che non punta a niente:




```
$> echo "ciao a tutti!" > new_file
$> ln -s new_file another_file
$> ls -l new_file another_file
-rw-------    1 gunetti    13 Dec 27 17:33 new_file
lrwxrwxrwx   1 gunetti     8 Dec 27 17:34 another_file -> new_file
$> rm new_file
$> cat another_file
cat: cannot open another_file
```

I symbolic link (link simbolici)

- All'interno di un file system, l'uso dei link fisici è più efficiente dell'uso dei link simbolici, dato che usando un link simbolico per accedere ad un file il SO deve leggere un i-node in più.
- Tuttavia (oltre che per permettere i link fra directory), i link simbolici sono necessari in Unix perché non è possibile realizzare hard link tra file che stanno su volumi/partizioni diverse.
- Per trovare la ragione dovete essere un po' nerd/geek, ma siete invitati a provarci.
- N.B.: NTFS supporta gli hard link mediante opportune system call, sebbene la cosa rimanga un po' "nascosta"…

Per chi vuole approfondire

- Capitolo 15: Dettagli interni del file system