SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Ingegneria dell'Informazione
Indirizzo Informatica

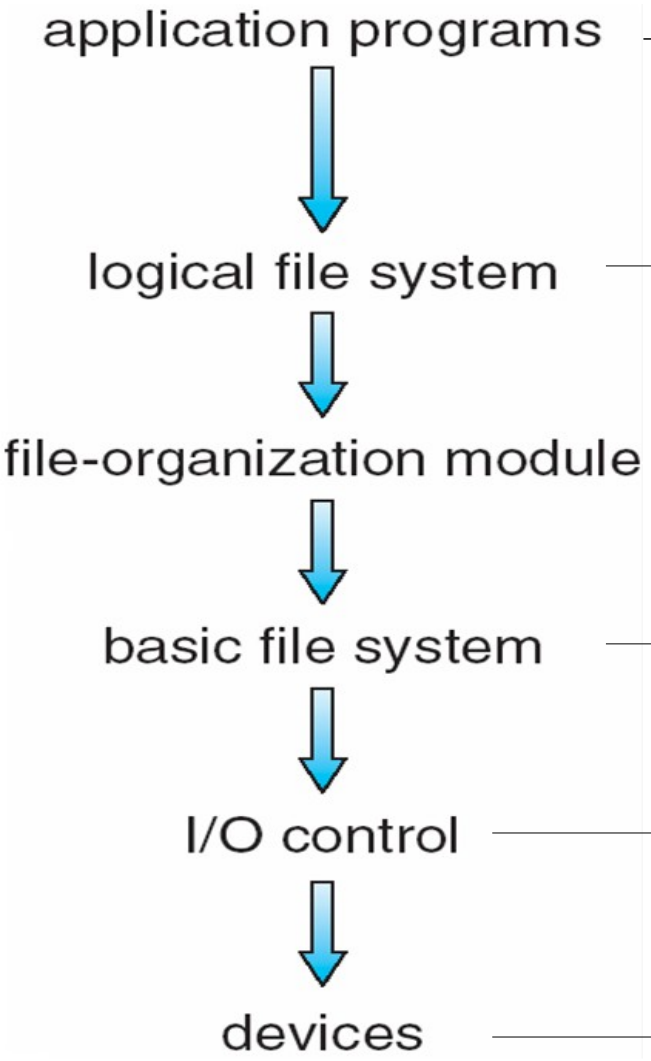
Reti e sistemi operativi

Implementazione del file system

Struttura di un file system

- Operativamente parlando, il file system è un insieme delle **strutture dati** e **funzioni** che ci permettono la creazione, l'accesso e la modifica di dati e programmi presenti in un sistema di calcolo.
- Il file system risiede permanentemente nella memoria secondaria (“disco”):
  - Permette un semplice interfacciamento con il disco, nascondendo all'utente le difficoltà implementative e mappando dall'organizzazione logica (astrazione) a quella fisica.
  - Mette a disposizione efficienti funzioni per archiviare e reperire dati a disco

Organizzazione a livelli (1/3)



Accedono al file system attraverso syscal (open(), read(), write(), ...)

Gestisce i **metadati** associati al file (File Control Block) → proprietario, gruppo, permessi, locazione del file.

Mappa indirizzi logici e fisici, gestisce l'allocazione dello spazio libero

Gestisce i trasferimenti I/O invocando comandi di basso livello, gestendo il buffering e l'unbuffering e utilizzando una cache per mantenere metadati usati di frequente

Device driver: traduce i comandi di basso livello in sequenze di segnali di controllo

Effettua l'accesso fisico ai dati

Organizzazione a livelli (2/3)

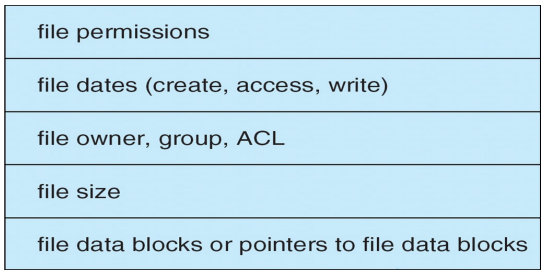
- **Controllo I/O (device driver)** → Traduce un comando (input) del tipo “leggi dal drive1, cilindro 72, traccia 2, settore 10 nella locazione di memoria 1060” in una sequenza di segnali di controllo di basso livello impartiti direttamente all'hardware
- **File system di base** → traduce un comando del tipo “carica il blocco 123 nella locazione 1062” nel corrispondente comando input del device driver
  - Utilizza buffer temporanei e cache per gestire i dati in transito.
- **Modulo di organizzazione dei file** → gestisce la memorizzazione in memoria dei dati conoscendo quali blocchi logici afferiscono ad uno specifico file e qual è la mappa tra indirizzi logici e fisici.

Organizzazione a livelli (3/3)

- **File system logico** → gestisce i metadati associati ai file
  - Traduce nomi in identificatori univoci (es. file descriptor in Unix), mantenendo un **file control block** associato ad ogni file (**FCB**, inode in Unix) con memorizzati dati quali permessi, proprietario, ecc..
  - Gestisce le directory (struttura ad albero)
  - Gestisce la protezione degli accessi
- Organizzazione a livelli spesso non implementata direttamente per ragioni di **efficienza**.
- Ogni sistema operativo può includere più di una tipologia di file system
  - Linux ne possiede molte decine (file system general purpose, di compatibilità e virtuali).

Implementazione

- Per realizzare un file system sono necessarie alcune strutture memorizzate direttamente all'interno del dispositivo (partizione)
  - **Boot control block** → necessario se il file system incluso nella partizione contiene un sistema operativo eseguibile.
  - **Partition control block** (superblock in Unix) → contiene informazioni quali il numero totale di blocchi di cui è composta la partizione, array di puntatori ai blocchi liberi, ecc...
  - **Strutture delle directory** → nomi e numero FCB (inode)
  - **FCB (inode)** → per ogni file →



- Queste strutture variano a seconda della particolare implementazione.

Gestione del file system (1/3)

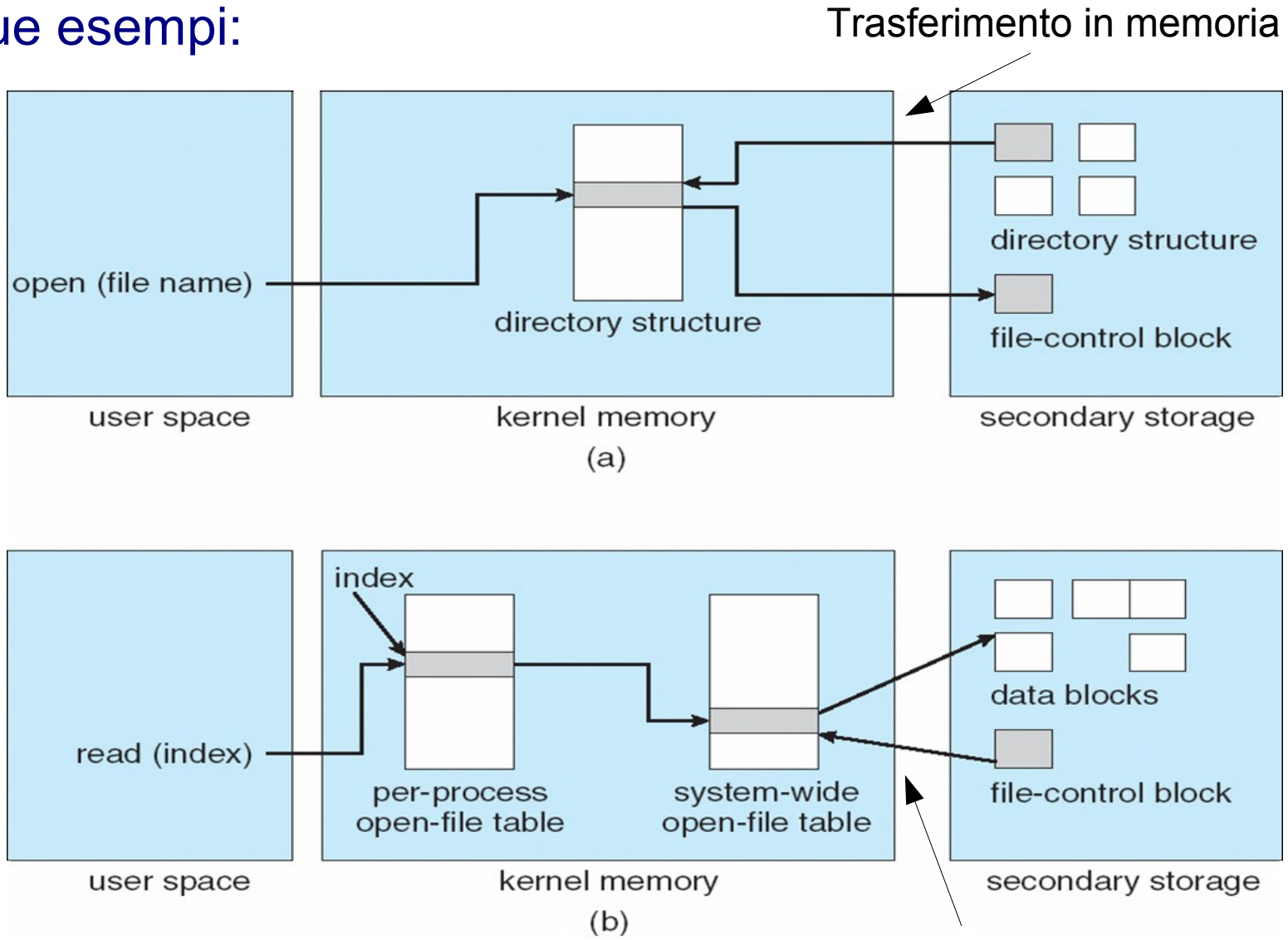
- Per gestire direttamente il file system e per eseguire le operazioni in maniera efficiente, il sistema operativo mantiene in memoria:
  - **Open-file table globale dei file aperti**
  - **Open-file table di processo** → punta a entry della tabella globale
  - **Buffer in memoria** per rendere più efficiente l'I/O
- **Creare** un nuovo file:
  - Il programma comunica con il logical file system (con syscall)
  - Viene allocato un un nuovo FCB, che viene aggiunto alle file table
  - Si carica **in memoria** la struttura della directory in cui risederà il nuovo file
  - Si modifica la struttura aggiungendo il nuovo file
  - Si salva la directory e il nuovo file (FCB + dati)

Gestione del file system (2/3)

- **Aprire** un file:
  - Si consulta l'open-file table globale per verificare se il file è già aperto da un altro processo.
  - In caso affermativo, si aggiunge una entry nella open-file table di processo e la si fa puntare alla corrispondente entry nella tabella globale. Un contatore associato a quest'ultima viene incrementato.
  - Viceversa, prima viene caricato il FCB e aggiornata la tabella globale.
  - Il file viene gestito attraverso un ID (file descriptor in Unix)
- **Chiudere** un file:
  - I dati eventualmente modificati e bufferizzati vengono salvati a disco, la entry nella page table locale viene rimossa
  - Se il contatore di riferimenti è a 0, vengono salvati anche i metadati e la entry nella page table globale viene rimossa

Gestione del file system (3/3)

- Due esempi:



Avvio del sistema e montaggio

- Il boot (control) block è una piccola partizione che contiene un piccolo programma in grado di caricare da disco il sistema operativo (kernel)
  - **L'hardware infatti NON è in grado di comprendere il file system ove il S.O. è memorizzato**
  - E' necessario sapere **dove** è memorizzato (puntatore al kernel) → **partizione di root**, montata all'avvio
- In alternativa, il boot block può puntare ad un programma specializzato (**boot loader**) in grado di gestire più file system e far partire più sistemi operativi a scelta.
- In fase di montaggio delle partizioni, l'integrità del file system viene testata, in caso di errori si prova a sistemare le cose.
- Altre partizioni possono essere in seguito montate automaticamente o manualmente

Directory: implementazione (1/2)

- La scelta delle strutture dati e degli algoritmi di gestione delle directory influenza l'efficienza e l'affidabilità di un sistema.
- **Lista lineare**: array non ordinato di coppie {nome, puntatore ai dati} (o {nome, puntatore} a FCB}).
  - **Problema**: apertura e creazione di un file richiedono O(n)
  - **Possibile soluzione**: si mantiene una lista ordinata attraverso un'apposita struttura dati, ad esempio un binary search tree (può diventare sbilanciato) o strutture più sofisticate quali i B-tree
  - Se la lista è ordinata, non è richiesto ordinare i file in un momento successivo

Directory: implementazione (2/2)

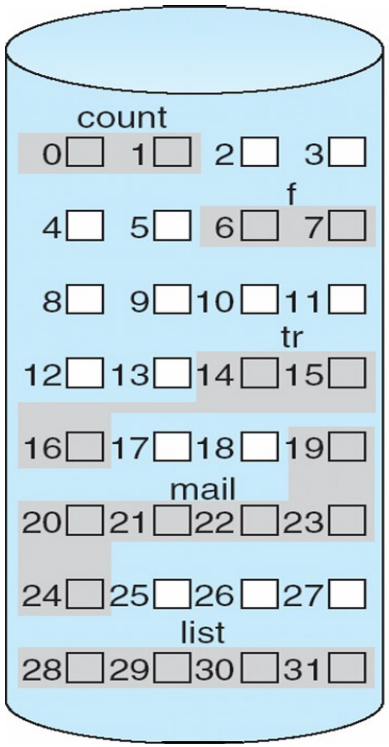
- **Hash table**: si accede al file (o al FCB) mediante una struttura dati hash, ovvero hash_func(nome file) → entry nella lista.
  - Ricerca e creazione dei file molto più efficiente, O(1)
  - **Problemi**: spreco di spazio per directory con pochi file e gestione delle collisioni
  - **Possibili soluzioni**: ingrandire e aggiornare l'hash table se necessario oppure utilizzare una **chained hash table**.
- Il file system di default di Linux (ext4) utilizza un'efficiente soluzione ibrida basata su H-tree (hashed B-tree)

Allocazione dello spazio a disco

- Gli obiettivi di un algoritmo di allocazione di memoria secondaria sono:
  - Rapidità d'accesso
  - Minor spreco possibile di spazio (ovvero minimizzare la frammentazione e lo spazio richiesto per contenere le strutture dati usate dal file sysyem)
- Un metodo di allocazione definisce come i blocchi del disco sono assegnati ad ogni file
  - E' necessario infatti tenere conto della struttura “a blocchi” in cui solitamente sono organizzate le memorie di massa

Allocazione contigua (1/2)

- Ciascun file occupa un insieme di blocchi contigui sul disco.
- Per reperire il file è necessario conoscere solamente la locazione iniziale (**indice** di blocco) e la **lunghezza** (in blocchi).



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

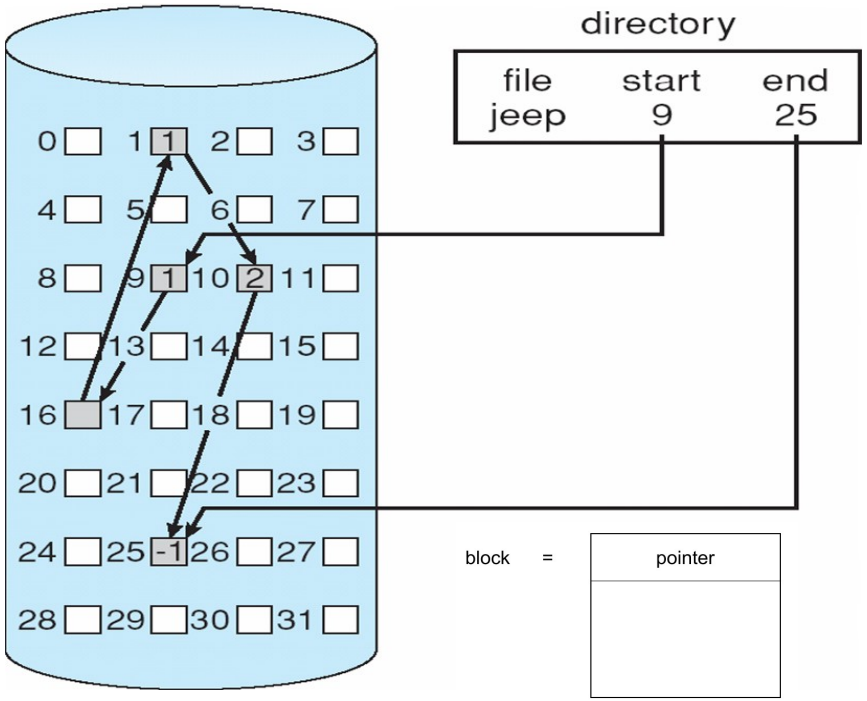
Per mappare un indirizzo da logico (LA) a fisico (si supponga dimensione blocchi 512):
LA/512 → (quoziente **q**, resto **r**)
Blocco fisico = start + q
Offset nel blocco = r

Allocazione contigua (2/2)

- **Vantaggi**:
  - Accesso casuale
  - Rapidità di accesso: vengono minimizzati i seek a disco.
- **Probleimi**:
  - **E' necessario conoscere a priori la lunghezza del file**, ovvero i file non possono crescere dinamicamente, può essere necessaria una rilocazione dei file (deframmentazione) → **alto overhead**
  - **Reperimento blocchi liberi dispendioso**
  - **Frammentazione** sia interna che esterna
- **Possibili soluzioni**: utilizzo di **extent**, ovvero di porzioni **contigue** di memoria secondaria composte da più blocchi. Se un extent non è sufficiente per contenere il file, un nuovo extent viene allocato linkato con il precedente.

Allocazione concatenata (1/2)

- Ciascun file è composto da una linked-list di blocchi a disco: i blocchi possono essere **sparsi ovunque nel disco**.
- La directory (o l'FCB puntato dalla entry in directory) contiene un puntatore alla testa della lista di blocchi (e possibilmente alla coda, oppure l'ultimo elemento della lista punta a NULL).
- Ogni blocco contiene un puntatore al successivo (trasparente all'utente).



Per mappare un indirizzo da logico (LA) a fisico (si supponga dimensione blocchi 512 e puntatori di dimensione 1):
LA/511 → (q, r)
Blocco fisico = il q-esimo nella lista
Offset nel blocco = r + 1

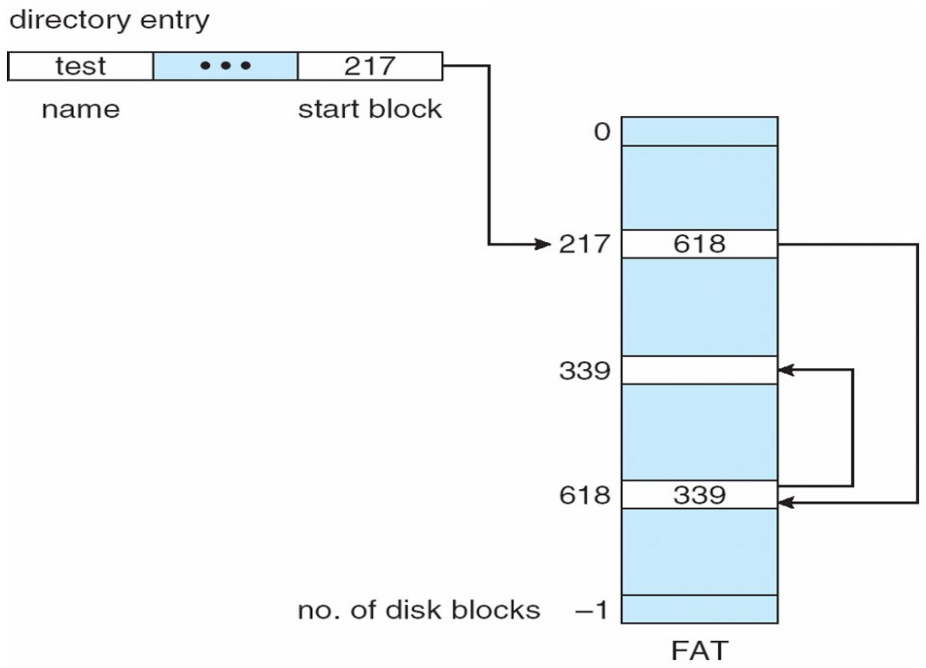
Allocazione concatenata (2/2)

- In scrittura, nuovi blocchi liberi vengono allocati e linkati *on-demand* se il file cresce oltre l'ultimo suo blocco.
- In lettura, si parte dal primo blocco e si segue **sequenzialmente** l'ordine della linked list
- **Vantaggi**: Non c'è frammentazione esterna
- **Problemi**:
  - Non è possibile l'accesso casuale
  - L'accesso ad un file richiede frequenti seek
  - Spreco di spazio per memorizzare i puntatori
  - Affidabilità
- **Possibile soluzione (utilizzata freuquentemente)**:Cambiare la granularità: utilizzare cluster di blocchi contigui anziché blocchi singoli → **maggiore frammentazione interna**

FAT: File Allocation Table (1/2)

- Variazione dell'allocazione concatenata utilizzata ad esempio nell'MS-DOS
- La FAT è una tabella memorizzata ad esempio all'inizio della partizione con un **elemento per ogni blocco del disco**, l'indice è il numero di blocco.
- L'entry di un file all'interno della directory (oppure il corrispondente FCB) contiene l'**indice del primo blocco in cui è memorizzato il file**.
- Ogni entry della FAT contiene l'**indirizzo del successivo blocco** appartenente al file, oppure uno specifico valore EOF (es. -1) se è l'ultimo blocco.
- Blocchi non allocati hanno valore 0 nella FAT

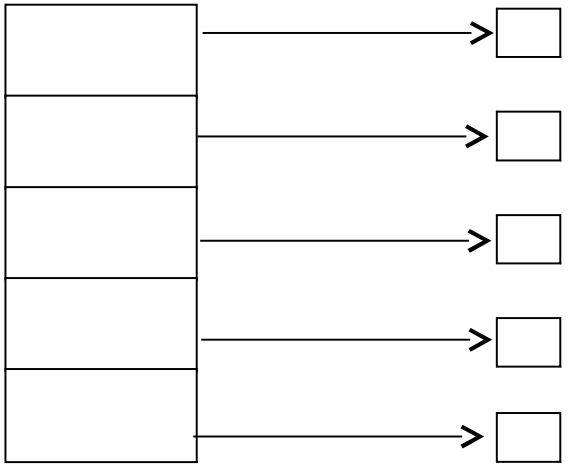
FAT: File Allocation Table (2/2)



- **Vantaggi**:
  - Accesso casuale migliorato rispetto a linked list di blocchi
- **Problemi**:
  - Frequenti seek dalla FAT ai blocchi dei file
- **Soluzione**:
  - Caching della FAT

Allocazione indicizzata (1/3)

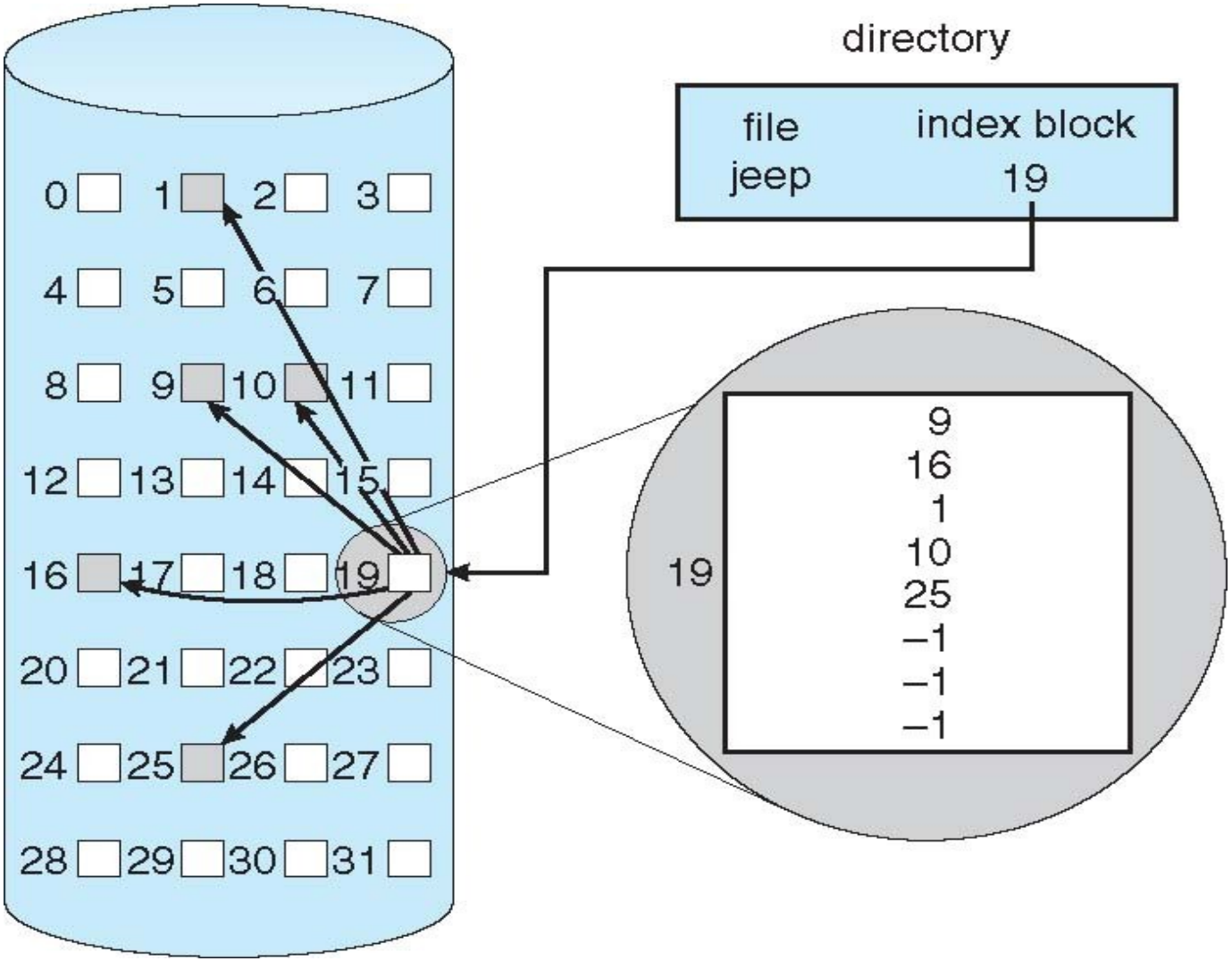
- **Ogni file** utilizza un **blocco indice**, ovvero un blocco che contiene i puntatori a tutti i suoi blocchi.
- Quando il file è creato, tutte le entry del blocco indice vengono settate a NULL



index table

Per mappare un indirizzo da logico (LA) a fisico (si supponga dimensione blocchi 512):
LA/512 → (q, r)
Blocco fisico = q-esima entry nella tabella indice
Offset nel blocco = r

Allocazione indicizzata (2/3)



Allocazione indicizzata (3/3)

- **Vantaggi**: accesso casuale
- **Problemi**:
  - Grande spreco di spazio → un blocco indice per ogni file, anche quelli più piccoli
  - Frequenti seek per accedere al blocco indice
  - Se il file è molto grande, un solo blocco indice può essere inadeguato
- **Possibili soluzioni**:
  - Caching del blocco indice
  - Linked list di blocchi indice, blocchi indice multi-livello, schemi combinati

Linked list di blocchi indice

- Non c'è limite alla dimensione
- Entry a NULL se ultimo blocco indice

Per mappare un indirizzo da logico (LA) a fisico (si supponga dimensione blocchi 512 e puntatori di dimensione 1):

LA/(512 * 511) → (q1, r1)

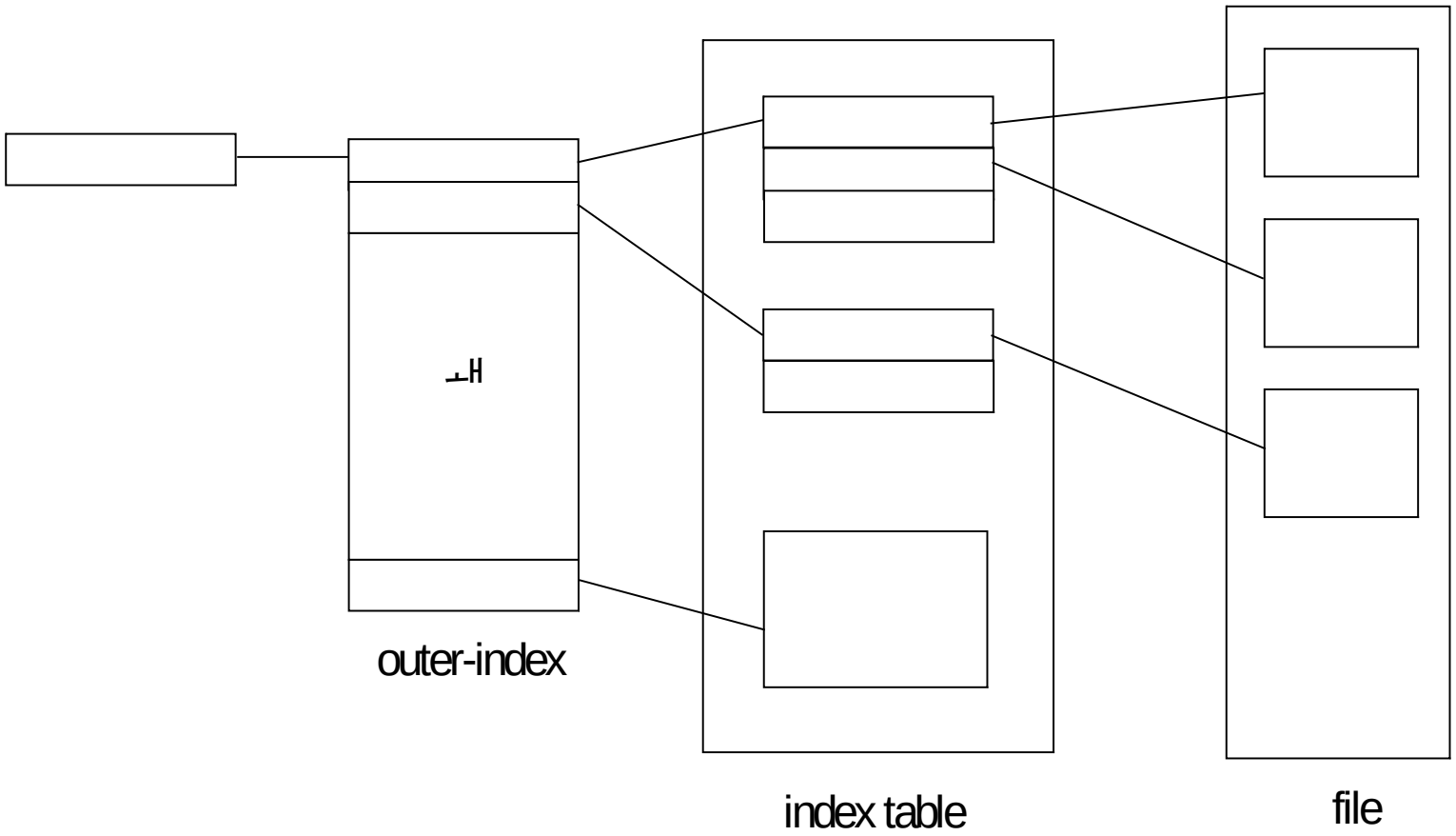
Blocco indice = il q1-esimo nella lista dei blocchi indice

r1/512 → (q2,r2)

Spostamento nel blocco indice = q2 + 1
Offset nel blocco del file = r2

Blocchi indice multi-livello (1/2)

- Variazione delle linked list di blocchi indice



Blocchi indice multi-livello (2/2)

Per mappare un indirizzo da logico (LA) a fisico in caso di **2 livelli** (si supponga dimensione blocchi 512):

LA/(512 * 512) → (q1, r1)

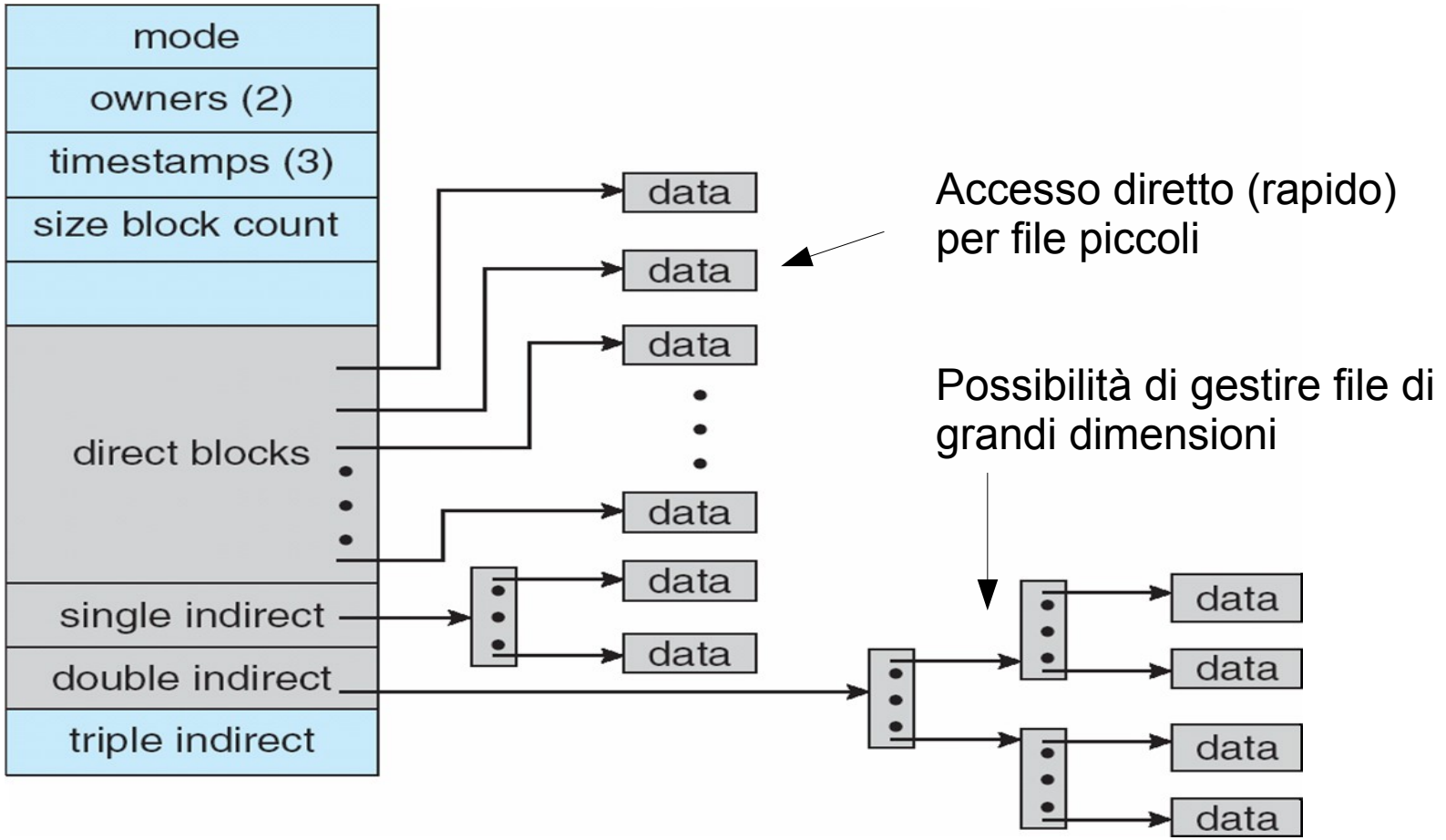
Offset nell'indice esterno (blocco indice) = q1

r1/512 → (q2,r2)

Spostamento nel blocco indice selezionato = q2
Offset nel blocco del file = r2

Schema combinato

- Esempio in Unix

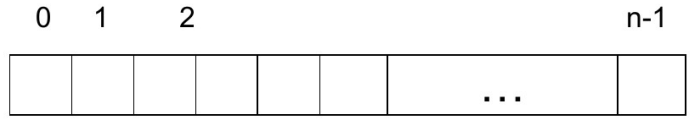


Prestazioni del file system

- Il migliore metodo da scegliere dipende dal tipo di accesso:
  - L'allocazione contigua è efficiente sia per accessi sequenziali che casuali
  - Allocazione concatenata efficiente solo per accessi sequenziali
- Alcuni sistemi operativi permettono di dichiarare il tipo di accesso, e il metodo di allocazione viene scelto di conseguenza
- Spesso utilizzano sistemi ibridi (vedi Unix)
- Spesso i metodi preferiscono utilizzare la velocità della CPU, eseguendo molte istruzioni prima dell'accesso, per eseguire accessi efficienti che minimizzano i movimenti inutili della testina

Gestione dello spazio libero (1/3)

- **Vettore di bit** (n blocchi) → ogni blocco presente a disco rappresentato da un bit:



bit[*i*] = { 1 ⇒ block[*i*] free ; 0 ⇒ block[*i*] occupied

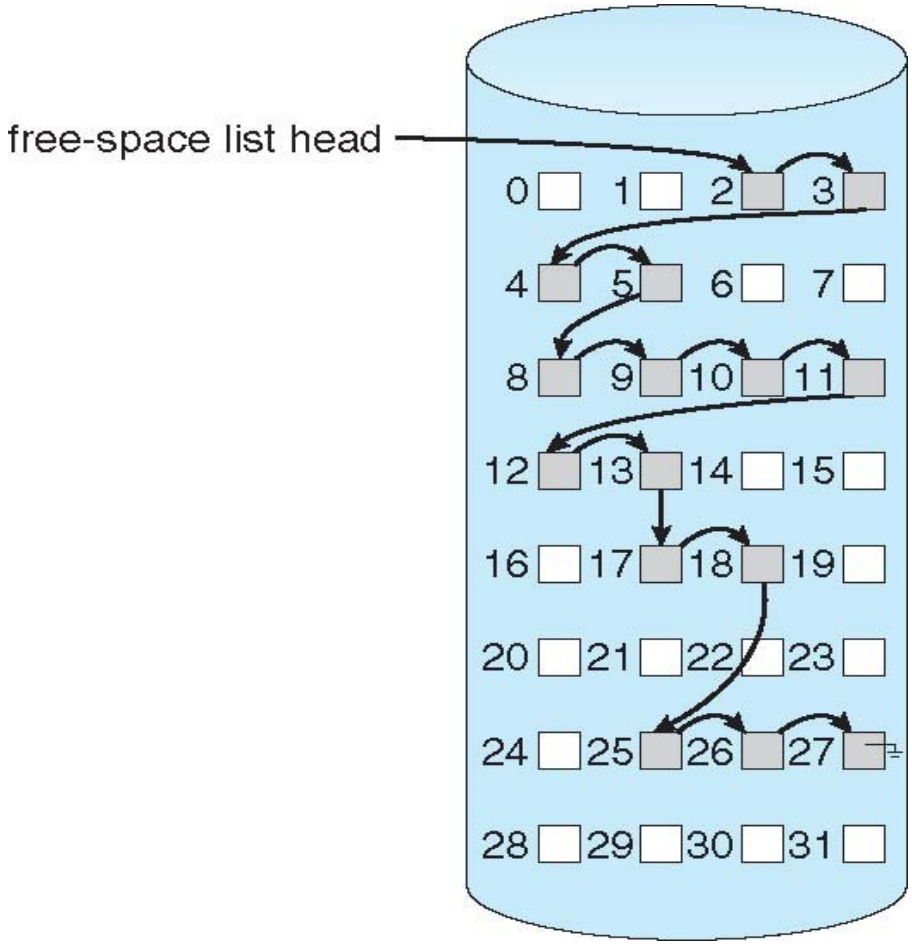
- Calcolo del numero del primo blocco libero: si scorre il vettore, cercando il primo word (es. byte) diverso da 0.

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

- Efficiente se il vettore è mantenuto in memoria centrale → solo per dischi di ridotte dimensioni

Gestione dello spazio libero (2/3)

- Con linked list → Difficilmente fornisce aree contigue
- Non si spreca spazio.
- Efficiente nel fornire spazi liberi



Gestione dello spazio libero (3/3)

- **Raggruppamento**: sul primo blocco libero sono memorizzati gli indirizzi di n blocchi liberi Di questi, l'ultimo sono sono memorizzati gli indirizzi di n blocchi liberi, e così via
- **Conteggio**: tiene traccia dei primi blocchi liberi di ogni "hole", e per ognuno la dimensione dell'hole