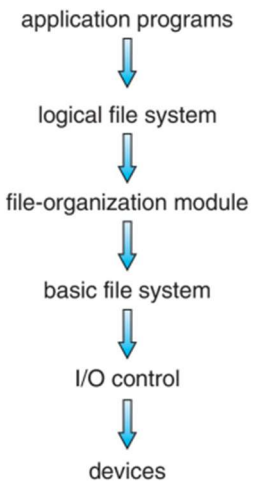
CH14 – FILE SYSTEM IMPLEMENTATION

Dobbiamo realizzare file e direttori

- **File**:
  - Unita logica di memoria
  - Collezione di informazioni
- **File system:**
  - Fa riferimento ai dischi che hanno blocchi di settori
    - Secondary storage
  - Interfaccia utente fornita per **l'archiviazione**, mappando la logica con quella fisica
  - Fornisce un accesso efficiente e conveniente al disco consentendo l'archiviazione dei dati, la localizzazione e la facilità di recupero
- **FCB**: struttura contenente le informazioni su file
- **Device dirver**: organizza il dispositivo

*Il file system è organizzato in **livelli** :*



I livelli più importanti sono:

- **Basic file system:** dato un comando, lo traduce in device driver
  - permette di gestire eventuali buffer e cache
    - buffer: per dati in transito
    - cache: per dati usati frequentemente
  - Es traduzione: Leggi blocco 123 → leggi quel determinato cilindro
- **File-organization module:** capisce file, indirizzi logici e blocchi fisici
  - Traduce # blocco logico in # blocco fisico
  - Gestisce spazio libero e allocazione su disco
- **Logical file system**
  - Gestisce i metadati
    - Traduce nomi in numeri
    - Gestisce direttori
    - Protezione

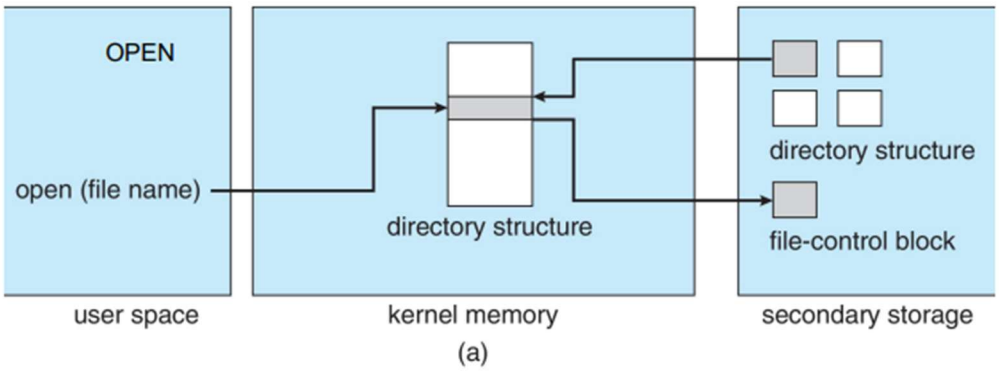
Operazioni file system:

- Mette a disposizione delle **system call** :
- **Boot** control block: ha a disposizione delle informazioni per bootstrappare
- **Volume** control block: serve a gestire i dischi
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- **FCB**: file ControlBlock → uno per ogni file, contiene informazioni sui file
  - inode number, permissions, size, dates

IN MEMORIA:

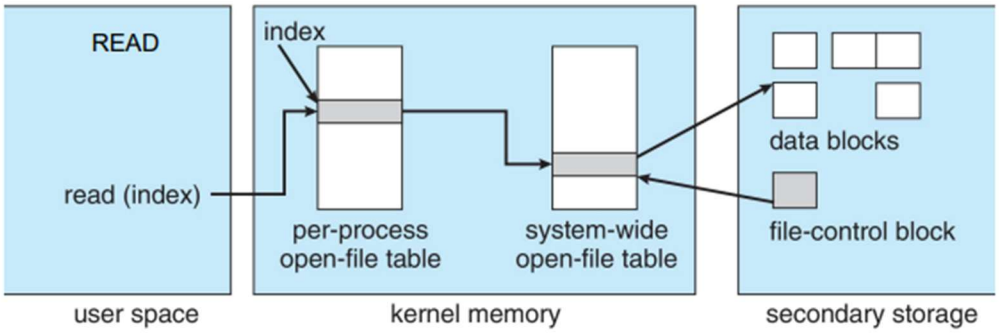
In memoria sono presenti delle strutture dati, si caricano in memoria dei duplicati dei file. Si organizzano delle strutture dati che servono per lavorare sul file system:

- **Mount able** → gestisce volumi mounted
  - Quali sono i volumi
- **System-wide open-file table**→ contiene una copia di FCB di ogni file e altre informazioni
  - *Una sola copia del file (indipendente dal nr di user che lo stanno usando)*
- **Per-process open-file table** → contiene puntatori alle entry appropriate del system-wide open-file table e altre info.
  - Quando più user stanno scrivendo/leggendo un file:
    - Ognuno sta lavorando su degli specifici byte



(a)

OPEN chiede nome file, nella kernel memory c'è una struttura dati di direttori. Dato il nome del file, chiedo alla directory structure di avere il file control block. File control block viene inserito nella system- wide table. Inoltre crea una entry sulla pre-process per lavorare su file. Dunque, la open ha aggiornato le tabelle.



(b)

Read: ha un file descriptor; alla read dici già dove sta il file nella pre-process file tabel → deve avere accesso diretto alla fcb. Alla read dice l'indice che ti è tornato dalla open. Quando hai finito la open hai quindi accesso diretto al file e quando fai read, usi quesyo puntatore.

Una volta che ho la read, devo implementarla

Ogni file system ha la propria struttura e propria strategia:

- La read vuole localizzare completamente il file, la copia del FCB
- Poi localizzare i dati e portarli in memoria da qualche parte

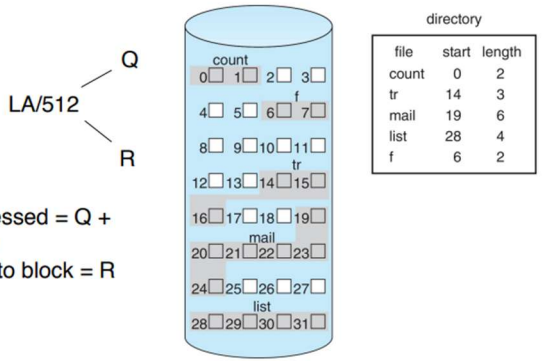
Come si implementano i direttori:

- **Lista lineare** di nomi di file con puntatori ai blocchi di dato
  - Problema di ricerca
- **Tabella di hash**
  - Problema di collisioni

Come si allocano i file:

Allocazione contigua

- Basta ricordarsi chi è il primo
- Funziona bene con sequenziale
- Problemi di frammentazione
- Di ogni file, basta il nome e sapere dove inizia

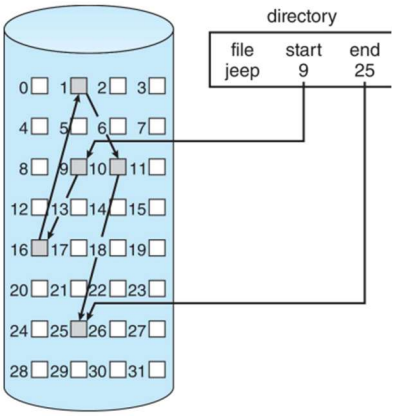


LA: LogicalAddress → byte nel file

- Esempio con blocco da 512 byte
- LA/512 → numero di blocco
- LA%512 → offset nel blocco
  - Facendo divisione ottengo quindi l'indirizzo fisico
  - Per passare a quello che è il vero inidirizzo fisico nel blocco, va aggiunto lo starting address

Lista concatenata:

- Ogni file è una lista di blocchi
- Ogni blocco, contiene dati e puntatore al blocco successivo



Vantaggio:

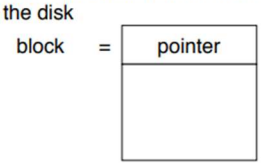
- Non c'è frammentazione esterna
  - Non c'è problema di compattazione
- Facile gestire spazio libero

Problema:

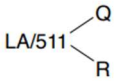
- Non c'è accesso diretto, localizzare un blocco è un problema
  - Se voglio andare al blocco 1 del file, devo fare 9-16-1
- Se perdo un blocco, perdo anche quello a cui è collegato e così via

Nota: nella lista concatenata una parte di dato è per indirizzo del prossimo; quindi, non divido per 512 ma per 511 in quanto stiamo supponendo che ne serva 1 per indirizzo next

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



- Mapping



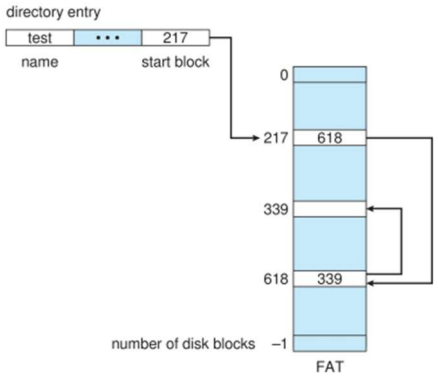
Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1

FAT → *File Allocation Table*

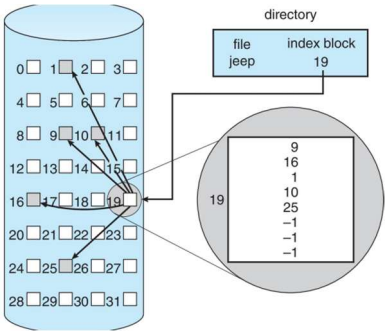
I blocchi sono concettualmente un vettore → una lista di blocchi in un disco può essere visto come dei blocchi in ognuno dei quali c'è il puntatore al successivo o come dati da una parte e puntatori da un'altra → FAT:

- All'entry 217 dela FAT, corrisponde il blocco 217 su disco
- **Blocchi contengono solo dati, i puntatori saranno in in FAT**
- Se la FAT viene memorizzata in modo sicuro, si evita il problema della rottura del blocco con conseguente perdita di blocchi successivi
- Entry della FAT ha dimensione in base a quanti bit servono per rappresentare indirizzo blocco
  - Molto piccola rispetto ai blocchi dato (rapporto 1/1000)
- Nota: se porti la FAT in memoria già al boot → hai già tutti i puntatori



Metodo di allocazione indicizzato

Ogni file ha il proprio index block di puntatori ai propri blocchi dato.
Equivalente degli alberi ma con profondità limitata → alberi larghi ma poco profondi



- **Tabelle di indici**: ogni file ha il proprio index block di puntatori ai data blocks
  - Es: jeep ha 5 blocchi (9-16-1-20-25)
  - Limitato in dimensioni, un blocco che contiene puntatori a blocchi, potrebbe non essere abbastanza
- I**ndexTable**: più blocchi contingui contengono la tabella degli indici
  - LA/512=Q → nr di blocco che fa traduzione logico-fisica
- **Schema linkato**:
  - Primo blocco ha un puntatore ad un blocco contenente altri puntatori a blocchi
- **Gerarchico:** page table a più livelli

Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

LA/512 → Q, R

Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)

LA / (512 x 511) → Q_1, R_1

Q_1 = block of index table
R_1 is used as follows:

R_1 / 512 → Q_2, R_2

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

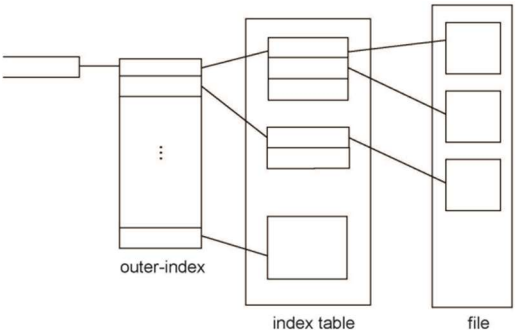
- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

LA / (512 x 512) → Q_1, R_1

Q_1 = displacement into outer-index
R_1 is used as follows:

R_1 / 512 → Q_2, R_2

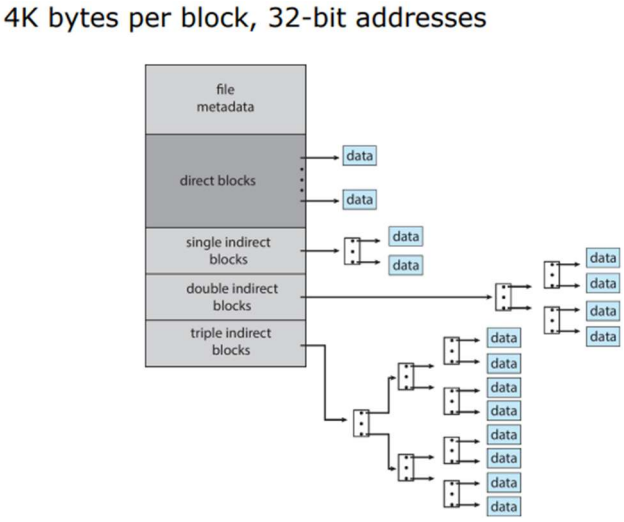
Q_2 = displacement into block of index table
R_2 displacement into block of file:



i-node

Schema combinato in cui si **usano più approcci insieme all'aumentare delle dimensioni**:

- Primi 10/12 blocchi di dato, puntatore diretto all'i-node (file max 10kB) → accesso diretto rapido.
- File più grosso → metto inidici nel single indirect block (tabella ad un solo livello)
- Se acnora più grosso→ due livelli
- Altrimenti 3 livelli



Performance

La locazione indicizzata è più complicata ma ha un costo accettabile anche per accesso diretto nel caso in cui il numero di livelli sia maggiore di 0.

- Allocazione contigua permette di andare direttamente se sai indirizzo
  - Ottimo per sequenziale e casuale
- Lista linkata e Fat non permettono accesso diretto
  - Bene per sequenziale, no casuale
- Accesso indicizzato, attraverso la tabella degli indici permette di avere un accesso costante

Negli HardDisk c'è il problema del costo del riposizionamento: se i blocchi sono vicini paghi meno. Nelle NVM non ci sono questi problemi in quanto non c'è la testina del disco.

Free-space list: per tenere traccia dei blocchi/cluster disposinibili

- Mi serve un **solo blocco libero**
  - Liste
    - Ordinate
    - Non ordinate
  - I-node
- Mi serve **un blocco contiguo**
  - Devo cercare un intervallo
    - Meglio con Bitmap
- Free-space List:
- Bit-vector o bit-map

DIFFERENZA TRA LISTE ORDINATE E NON:

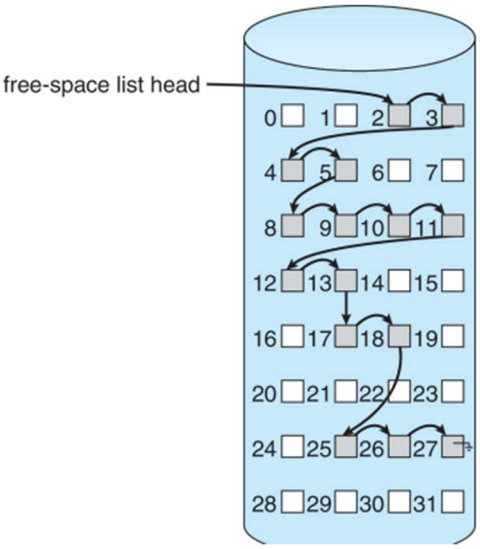
In liste non ordinate, difficile trovare un blocco libero → algoritmo quadratico

- In liste non ordinate:
  - Free costa meno
  - Allocazione blocco singolo → ok
  - Allocazione blocchi contigui → quasi impossibile

In liste ordinate, è possinile gestire intervalli contigui → blocchi consecutivi, si troveranno di fila in una lista.

- In liste ordinate:
  - Estrai da testa
  - Metti in coda

Esempio di lista di blocchi liberi- Linked Free Space List



Note:

- Difficile ottenere spazio contiguo
- Non si spreca spazio
- Non bisogna attraversare l'intera lista se il nr di blocchi liberi è salvato

***Nota**: la tipologia usata per i blocchi liberi può essere diversa da quella usata per i file allocati.*

***Nota**: con una FAT si può fare una free-list, si può anche mettere una free-list dentro una fat.*

Le **liste linkate possono essere modificate** leggermente in modo tale che queste rappresentino blocchi contigui:

- **Grouping**: Modificare l'elenco dei collegamenti per memorizzare l'indirizzo dei successivi n-1 blocchi liberi nel primo blocco libero, più un puntatore al blocco successivo che contiene puntatori liberi (come questo)
- **Counting**: indirizzo primo blocco e metti un contatore dei blocchi allocati disponibili

PARLANDO DI HARD DISK Si può dire che costo di scrittura e lettura di equivalgono.

Mentre Nei NVM non c'è simmetria

- Quando si scrive, si cancella e poi si scrive
- Ci sono strategie che permettono di gestire in modo ottimizzato lettura e scrittura con NVM
  - Es: **TRIMing**: TRIM è un meccanismo più recente per il file system per informare il dispositivo di archiviazione NVM che una pagina è libera. Può essere sottoposto a garbage collection o, se il blocco è libero, ora il blocco può essere cancellato

efficienza e performance

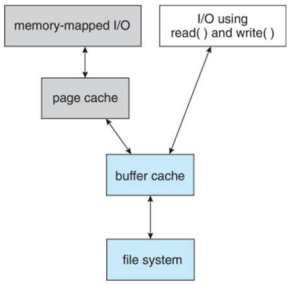
L'efficienza del filesystem dipende anche da come si alloca il disco e i direttori e dipende da come sono gestiti questi dati:

- Allocazione di meta-dati
  - Blocchi per generare direttori
  - Blocchi di indice
- Algoritmi di allocazione e di gestione di direttori
- Bisogna pre-allocare
- Vogliamo organizzare blocchi di dimensione fissa o di dimensione variabile

Performance:

- Mantenere dati e metadati chiusi insieme
  - Vantaggio: uso blocchi vicini
  - Svantaggi: potrebbero esserci complicazioni
- **Buffer cache**: sezione di memroia principale per blocchi più usati
  - Serve per accesso a disco con tentativo di avere una copia ridondante in RAM
- **Sincrono**: scrive qualcosa richiesto dalle app o necessarie al OS
- Free-behind and read-ahead:
  - Mentre leggi uno, leggi anche il successivo perché molto probabilmente ti servirà
- **Page cache:**
  - Strategia con il seguente contesto:
    - Esistono operazioni sul disco dette memory-mapping
      - Invece di accedere ad un disco con operazioni esplicite su file (Read, writE), realizza una specie di malloc che ritorna un puntatore in RAM che è associato ad un pezzo di disco
        - Vedo un file come fosse un vettore perché lo ho associata ad una mia aria di memoria → memory-mapping del file
          - Posso fare facilmente ordinamenti, swap, etc…
          - Diventa efficiente perché c'è una page cache (senza page cache, non lo sarebbe)

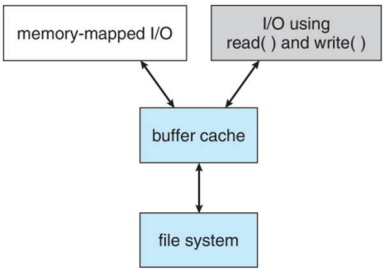
IO senza buffer cache unificata:



I/O con una buffer cache unificata

Realizzato un pezzo comune → bufferCache in modo tale da evitare problematiche quando memory-mapped I/O e I/O using read()write() agiscono contemporaneamente.

- Una cache buffer unificata utilizza la stessa cache delle pagine per memorizzare nella cache sia le pagine mappate in memoria che l'I/O ordinario del file system per evitare la doppia memorizzazione nella cache



Garantire che non si perdano dei :

- **Check della consistenza**
  - Compara dati nella struttura del direttorio con quelli del disco e cercare di fixare le inconsistenze
- **Back-up :**
  - Utilizzare i programmi di sistema per eseguire il backup dei dati dal disco a un altro dispositivo di archiviazione (nastro magnetico, altro disco magnetico, ottico)
- **Restoring**
  - Recover lost file or disk by restoring data from backup

Precedentemente ad npfs:

- Verifica se esistono file non agganciati
- Verifica se ci sono dei file parzialmente inconsisstenti

NPFS ha migliorato questa strategia usando **journaling**:

- Prima di fare un operazione segna e memorizza sul disco quel che si vuole fare
- Log (cose che si vogliono fare) viene mantenuto in un file di log
    - Una transizione è considerata committed solo quando scritta al log
- Queste cose saranno poi fatte, anche in modo asincrono e il record del lock può essere cancellato
- Le transazioni nel log vengono scritte in modo asincrono nelle strutture del file system
    - Quando le strutture del file system vengono modificate, la transazione viene rimossa dal log
- Se il file system si arresta in modo anomalo, tutte le transazioni rimanenti nel registro devono comunque essere eseguite
- Recupero più rapido da incidente, rimuove la possibilità di incoerenza di Metadati