# 5. Gestione dei file - Livello Logico

La parte del S.O. che si occupa della gestione dei file è la parte più visibile di un sistema operativo: fornisce meccanismi per la memorizzazione (su memoria secondaria), l'accesso e la protezione di dati e programmi del S.O. e degli utenti. **Il gestore dei file** è quella parte del S.O. che si occupa dell'organizzazione generale dei dati che risiedono in memoria secondaria (**permanente**). Anche il file system "virtualizza" dispositivi di memorizzazione permanente fornendone una visione logica uniforme. Quindi il sistema di gestione dei file **deve nascondere** all'utente tutti i dettagli relativi ai dispositivi di memorizzazione che sono usati come memoria secondaria, infatti ogni dispositivo ha caratteristiche fisiche diverse, il sistema operativo deve fornire una **astrazione** di lavoro con una visione logica e metodi di accesso uniformi tramite una interfaccia comune ed efficiente allo stesso tempo. La strttura logica con cui il sistema operativo gestisce i file prende il nome di **file system (logico)**

## 5.1 File System Logico

Vedremo più in dettaglio, quando parleremo del punto di vista **interno**, che il gestore dei file è stratificato. In particolare, lo strato più esterno è quello che viene chiamato **file system logico** che è responsabile per:

- Struttura dei dati nei file
- Organizzazione in directory
- Fornire all'utente un insieme di funzioni di alto livello per operare su di essi, mascherando le operazioni che vengono realmente effettuate per allocare la memoria di massa e per accedervi in lettura/scrittura
- Garantire protezione e condivisione di file

Il **file system logico** garantisce una gestione dei file indipendente dalle caratteristiche fisiche dei dispositivi che costituiscono la memoria secondaria: astrazione utile sia per l'utente sia per i programmi.

## 5.2 File

Un file rappresenta un insieme di informazioni correlate e memorizzate nella memoria secondaria → **persistenza**. Dal **punto di vista dell'utente**, un file è la più piccola porzione di memoria secondaria indirizzabile logicamente: i dati possono essere scritti nella memoria secondaria soltanto all'interno di un file. Dal **punto di vista del S.O.** i file vengono allocati su dispositivi fisici di memorizzazione non volatili (memoria secondaria o di massa). Un file viene identificato sempre tramite un **nome**:

- Una sequenza limitata di caratteri: la massima lunghezza dipende dal S.O. come l'insieme di caratteri leciti
- Il S.O. può fare o meno distinzione fra maiuscole e minuscole: case-sensitive o case-insesitive.
- Alcuni S.O. dividono il nome in due parti separate da un punto '.' la parte dopo il punto si definisce **estensione**. In alcuni S.O. l'estensione rappresenta il tipo del file.

## 5.2.1 Attributi dei File

Oltre al **nome**, che identifica il file per l'utente, ogni file ha **altri attributi** detti **metadati**:

- Tipo
- Locazione
- Dimensione
- Identificatori dell'utente che ha creato/possiede il file
- Protezione: permessi d'accesso in lettura, scrittura ed esecuzione
- Ora, data di creazione, modifica, ultimo accesso: dati per il controllo d'uso

## 5.2.2 Tipi di File

Il tipo e' un attributo di un file che ne indica la struttura logica interna e permette di interpretarne correttamente il contenuto. Alcuni S.O. gestiscono diversi tipi di file: conoscendo il tipo del file, il S.O. potrebbe evitare alcuni errori comuni, un S.O. che riconosce il tipo di un file può manipolarlo in maniera corretta. Esistono tre tecniche principali per identificare i tipi di file:

- **Estensioni**: il tipo viene indicato da un suffisso (estensione) del nome (MS-DOS)
- **Attributo tipo** associato al file (MacOS)
- **Magic number**: valore posto all'inizio del file (unix)

## 5.2.3 Struttura Logica dei File

A seconda del S.O. i file possono avere una delle seguenti strutture logiche:

- Nessuna: sequenza di parole o byte
- Sequenza a record semplici: linee, record di lunghezza fissa variabile
- Strutture più complesse: documenti formattati, archivi (ad albero, con chiavi, etc...), eseguibili rilocabili, etc...

La **struttura** viene decisa o dal S.O. o dall'applicativo che genera il file. Il tipo di un file e la corrispondente struttura logica possono essere riconosciuti e gestiti in modi diversi nei diversi S.O.

- Se il S.O. gestisce molti formati:
    - Codice di sistema ingombrante
    - Incompatibilità di programmi con file di formato differente
    - Gestione efficiente per i formati supportati
- Altrimenti il S.O. **non** gestisce specifici formati:
    - Codice di sistema piu snello
    - Formati gestiti dal programmatore
    - I file sono considerati semplici stream di byte
    - Solamente i file eseguibili hanno un formato predefinito dal S.O.

## 5.2.4 Directory

L'accesso ai file avviene attraverso l'uso di nomi simbolici gestiti dal File System. I file sono **oggetti persistenti**, bisogna preservare il collegamento fra nome del file e le informazioni contenute in esso. Un **descrittore** associa il nome di un file con i suoi attributi (metadati) in particolare la sua locazione sul dispositivo fisico di memorizzazione. I descrittori dei file sono mantenuti in strutture dette **directory**: una directory è un **insieme di descrittori** di file. Queste sono le operazioni che si possono effettuare su una directory:

- Ricerca di file al suo interno

- Elenco totale o parziale del contenuto
- Creazione di un file
- Cancellazione di un file
- Creazione/cancellazione di una directory

# 5.3 Punto di Vista Utente

Per gli utenti di un S.O. la parte relativa ai file è il **servizio** più visibile. La maggior parte dei comandi forniti dallo strato più esterno del S.O. riguarda i file.

## 5.3.1 Organizzazione delle Directory

L'organizzazione delle directory caratterizza un S.O. soprattutto dal punto di vista dell'utente: deve garantire **efficienza** nel reperire i file, regole di **nomenclatura** dei file ed infine condivisione e protezione dei file.

### Directory ad un solo livello (flat)

Questa rappresenta la più semplice struttura che può avere una directory. Tutti i descrittori di file sono contenuti in un'unica directory, questo approccio era usato specialmente nei primi S.O. monoutente con il **vantaggio** di condivisione immediata ma i seguenti **svantaggi**:

- **Non adatti** a sistemi multiutente
- **Lunghi elenchi** di nomi di file, poco efficiente
- Problema dell'unicità dei nomi che porta ad un naming complesso

### Directory a Due Livelli

Per superare le difficoltà legate all'uso di un'**unica** directory in ambito multiutente, **ogni utente** ha una propria directory (ad un singolo livello):

- **1° Livello**: Master File Directory (**MFD**)
- **2° Livello**: User File Directory (**UFD**)

Ogni utente quando accede al sistema, viene posto nel suo UFD, questa directory prende il nome di **home**. Questo approccio comporta i seguenti **vantaggi**:

- Adatto a sistemi multiutente
- Elimina il problema della unicità dei nomi fra utenti diversi
- Semplifica la protezione

E **svantaggi**:

- Lungo elenco di nomi di file, poco efficiente
- Unicità dei nomi nelle relative home directory
- Condivisione non immediata

#### Condivisione

Gli utenti, come abbiamo appena visto, sono completamente separati e confinati nelle loro home directory. Come fanno quindi a condividere un file nel caso in cui vogliano farlo? L'utente deve avere un modo di referenziare un file che non è nel suo UFD. Ad esempio tramite `username + filename`. Qui nasce il concetto di **path** ovvero percorso. Gli utenti **devono** poter usare **strumenti comuni**, ad esempio, compilatori linker, ecc... Viene definito un UFD **speciale** che contiene gli strumenti comuni, per non dover sempre far riferimento al path completo di questi strumenti lo spazio di ricerca per un eseguibile viene **ampliata**: prima si cerca all'interno del UFD dell'utente e poi in quello speciale, da qui nasce il concetto di **search path**.

## Directory ad Albero

Se consideriamo la struttura a due livelli un caso particolare di un albero possiamo facilmente pensare di **estendere** questa struttura ad un numero qualsiasi di livelli, si parla di organizzazione gerarchica a N livelli in cui ogni directory può contenere file e altre directory. Questo consente ad ogni utente di poter **creare sottodirectory**.Vediamo anche per questo approccio i vantaggi:

- Elenco corto di nomi di file, maggiore efficienze
- Eliminato il problema dell'unicità dei nomi pero ogni singolo utente, naming semplificato

Gli svantaggi:

- Condivisione ancora non immediata, ma risolvibile tramite **path** e **search path**

E le sue proprietà:

- L'albero ha un'**unica** radice
- Ogni file ha un **unico** path name
- Ogni directory può contenere sia file che **directory**

Ogni utente può definire una directory corrente con la possibilità di cambiarla. I **path name** possono essere relativi (alla directory corrente) o assoluti (a partire dalla root).

## Directory a Grafo Aciclico

Per facilitare la condivisione, se due utenti vogliono lavorare sugli stessi file, si concede la possibilità di "vederli" direttamente. Questo è diverso da avere due copie dello stesso file. Lo stesso file viene riferito con path name diversi da utenti diversi. Possiamo generalizzare lo schema ad un **grafo diretto aciclico** (DAG) esistono quindi path name molteplici per ogni file. Con una organizzazione delle directory a **DAG** si ha quello che viene anche detto **aliasing**: due o più path possono venire utilizzati per identificare lo stesso file o directory.

**Implementazione Base della Condivisione dei File**

1. Link simbolici: l'utente crea una entry particolare nella sua directory, questa identifica il path name completo del file condiviso. `ln -s`
2. Duplicazione descrittori: questo è un link hw, crea un clone dell'originale la quale riferisce il file condiviso `ln`

> Nota Bene
>
> I link simbolici sono indispensabili per creare link fra file system fisici diversi e sono necessari per creare link a directory.

Come bisogna gestire la cancellazione di un file condiviso?

1. Nel caso di link **simbolici** se viene eliminato il link non succede nulla all'originale. Se viene eliminato il file il link rimane **dangling → problema**
2. Si cancella effettivamente il file solo se non ci sono più link ad esso. In tal caso dobbiamo tenere un contatore dei link per ogni file.

Mantenere il grafo **A**ciclico, facilita l'implementazione di algoritmi necessari per attraversarlo, per esempio durante un backup. Per assicurare l'assenza di cicli bisogna ammettere **solo** link a file, e non a sottodirectory e ogni volta che viene aggiunto un nuovo link, verificare la presenza di cicli. Se si ammette la presenza di cicli, in fase di scansione del file system, occorre marcare, man mano che si percorre il ciclo, ciò che ho già visitato.

### Volumi

Un volume è una partizione contenente un file system a livello fisico: **file system fisico**. Nei S.O. semplici, come ad esempio MS-DOS, l'utente doveva conoscere in quale volume era memorizzato il file su cui voleva operare. S.O. più evoluti come unix, rendono **trasparente** all'utente qualunque nozione sui volumi: più volumi vengono montati in un unico **file system logico**.

#### File System Logico

Un file system logico è in generale composto da più file system fisici: uno di partenza e altri che devono essere **montati** in modo che la loro root sia un nodo e quindi una directory del grafo totale.

## 5.3.2 Protezione dei File

Soprattutto in sistemi multi-utente e' molto importante **controllare l'accesso** ai file. In genrale, per la rappresentazione di politiche di protezione e' necessario introdurre due concetti:

1. **Risorsa**: Le risorse sono le entità da proteggere, ad esempio, file directory, dispositivi... Ad ogni risorsa si associa un utente proprietario, che può concedere o negare ad altri utenti il permesso di accedere ad essa
2. **Dominio di protezione**: Il dominio di protezione viene definito come un insieme di coppie `<risorse, diritti>`. In genere si associa univocamente ad un utente e rappresenta l'insieme delle risorse alle quali e' abilitato ad accedere.

Le possibili operazioni su un file sono lettura (r), scrittura (w), esecuzione (x). Mentre su una directory lettura nomi di file (r) e creazione/cancellazione di un file (w). Ogni processo viene associato ad un particolare dominio, in genere, quello dell'utente che ha richiesto la sua creazione e quindi eredita i suoi diritti (quelli specificati nel suo dominio di accesso). In alcuni S.O. durante l'esecuzione i processi possono cambiare dominio, quindi **variare dinamicamente**. Chiaramente e' compito del S.O. tenere traccia in opportune strutture dati le relazioni tra risorse e domini di protezione. Questo avviene nella cosiddetta **matrice di protezione** dove ogni riga rappresenta un dominio di protezione associato ad un certo utente e ogni colonna rappresenta una risorsa, (file, directory, dispositivo...). Questa matrice a causa della sua vasta dimensione ed elevati tempi di accesso non e' l'approccio consigliato. Gli approcci più comuni sono: liste di controllo e liste di capability.

### Liste di Controllo

Una Access Control List rappresenta la politica di protezione associata ad una particolare risorsa (la colonna della matrice di protezione). Quindi per ogni risorsa `R`, vengono elencati i permessi concessi ad ogni utente `U`. Il S.O. autorizza o meno una operazione su un risorsa `R` ricercando nella corrispondente ACL se il processo dell'utente `U` che la sta eseguendo ha i diritti corretti. Sia unix che Windows utilizzano tecniche basate su ACL, in unix ad esempio per ogni file sono tracciati i permessi r w x per: user, group e other, per un totale di 9 bit.

### Liste di Capability

Per ogni utente `U`, il S.O. compila una lista di permessi associati alle diverse risorse cui l'utente e' abilitato ad operare (la riga della matrice di protezione). Questa tecnica offre maggiore efficienza rispetto ad ACL: per accedere ad un risorsa al processo viene assegnata, di norma, una **capability**, una specie di riferimento che mantiene anche le informazioni di quali operazioni sono consentite su quella risorsa. Questo approccio ha lo **svantaggio** che una operazione di revoca dei diritti di accesso associati ad una risorsa risulta una operazione onerosa, dato che richiede la ricerca e l'aggiornamento di tutte le c-list o di tutte le capability.

# 5.4 Punto di Vista del Programmatore di Sistema

Questa categoria di utenti, accede ai file tramite **chiamate di sistema** e vede un file come un **tipo di dato astratto**.

## 5.4.1 Operazioni

Iniziamo con il vedere quali sono le **operazioni** possibili su un file:

1. Creazione `create(filename)`
    - Bisogna trovare spazio per la memorizzazione
    - Creazione di un nuovo descrittore nella directory e quindi associazione nome/locazione
2. Scrittura `write(filename, data)`
    - Bisogna cercare la locazione del file: nome→descrittore→locazione
    - Si scrive il dato
3. Lettura `read(filename)`
    - Bisogna cercare la locazione del file: nome→descrittore→locazione
    - Si legge e ritorna il dato
4. Cancellazione `delete(filename)`
    - Bisogna cercare la locazione del file: nome→descrittore→locazione
    - Bisogna recuperare lo spazio
    - Cancellazione o invalidazione del descrittore nella directory, distruzione associazione nome/locazione

> Osservazione
>
> In tutte le operazioni (a parte la `create`), il S.O. deve ricercare il descrittore del file: se si devono eggettuare una serie di operazioni di lettura o scrittura questo risulta molto **inefficiente**. Introduciamo quindi due operazioni che agiscono su una struttura dati mantenuta in memoria centrale: la **tabella dei file aperti** per raggiungere una maggior efficienza.

5. Apertura `open(filename)`
    - Bisogna cercare la locazione del file: nome→descrittore→locazione
    - Si inserisce questa informazione (file control block) nella tabella dei file aperti
6. Chiusura `close(filename)`
    - Si elimina l'informazione relativa al file dalla tabella dei file aperti

In alternativa la open ritorna un indice nella tabella dei file aperti (detto **file descriptor**) e le funzioni che ne hanno bisogno usano quel file descriptor al posto del filename. Una operazione di open può servire anche per stabilire la modalità di accesso che si vuole usare: lettura, scrittura, entrambe. La modalità richiesta deve essere concessa dal meccanismo di protezione dei file a seconda dei diritti che si hanno su tale file. In base alla modalità dichiarata, nel caso di accesso contemporaneo di più utenti e/o processi allo stesso file il S.O. può adottare schemi per garantire la **consistenza** delle informazioni.

### Metodi di Accesso

In base al metodo di accesso definito dal S.O. o scelto dal programmatore, le scritture e le letture avvengono in maniera diversa:

1. Sequenziale
    - Metodo di accesso ai file più semplice

- Simile come principio alla lettura di un nastro
- Risulta impossibile la lettura oltre l'ultima informazione scritta, la scrittura aggiunge informazioni in fondo al file

Con il metodo di accesso sequenziale le operazioni di lettura e scrittura agiscono sul dato riferito dalla posizione corrente all'interno del file (**file pointer** o **I/O pointer**) all'atto della apertura in lettura, il file pointer viene posizionato all'inizio del file, con la creazione invece alla `end_of_file`. In genere tramite l'operazione **rewind** e' possibile tornare all'inizio.

2. Accesso diretto (casuale)
    - Metodo di accesso a file che si "ispira" al disco, il file viene visto come un array di record (di dimensione fissa) si può accedere direttamente al **record numero N** → accesso diretto in uno dei seguenti modi:
        1. Indicando esplicitamente a quale informazione si vuole accedere
        2. Introducendo un'altra operazione **seek** che sposta la posizione corrente del file pointer

    Il **vantaggio** di questo approccio e' che necessita di solo 3 primitive: due per leggere/scrivere con accesso sequenziale e una per spostare il file pointer. In questo modo le operazioni di read e write rimangono **invriate**: operanosempre in modo sequenziale.

Esistono **altri** metodi di accesso costruiti a partire dal metodo di accesso diretto: ad esempio, **accesso ad indice** usato per la gestione di basi di dati.

# 5.5 Punto di Vista del Sistema Operativo

Il gestore dei file e' stratificato, cioè organizzato in livelli: ogni livello si serve delle funzioni dei livelli inferiori per crearne di nuove, impiegate dai livelli superiori.

## 5.5.1 Strati del File System

Partendo dall'alto abbiamo:

1. **Logical File System**: quello che abbiamo già trattato
    - Gestisce i metadati, cioè tutte le  strutture del file system eccetto i dati veri e propri memorizzati nei file
    - Mantiene le strutture di file tramite i descrittori dei file, che contengono le informazioni sui file quali proprietario, permessi, posizione del contenuto...
    - Gestisce le directory
    - Gestisce protezione e sicurezza
2. **File Organization Module**
    - Traduce gli indirizzi logici di blocco in indirizzi fisici
    - Contiene il modulo per la gestione dello spazio libero
3. **Basic File System**
    - Invia comandi generici l driver di dispositivo per leggere/scrivere blocchi fisici su disco
    - Gestisce il buffer del dispositivo e la cache che conserva i metadati
4. **I/O Control**
    - Traducono comandi di altolivello in sequenze di bit che guidano l'hw di I/O a compiere una specifica operazione in una data locazione

La struttura a strati e' utile per ridurre la complessità e la ridondanza, ma aggiunge overhead nel e può diminuire le performance.

## 5.5.2 Struttura Interna dei File

Dispositivi di memorizzazione usati come memoria secondaria: **disco rigido**, dischi ottici, unità a stato solido, memorie flash... Trattiamo in particolare dei **dischi**. In un disco si indirizzano **blocchi** di datti, non singole word o byte come in memoria centrale, la dimensione dei blocchi, detti anche **record fisici** e' fissata a 4KB. La dimensione del **record logico** e' diversa da quella del bocco e possibilmente anche variabile. Bisogna rsolvere il problema della corrispondenzafra record logici e record fisici. Questo problema viene risolto attraverso il **packing** ovvero l'impacchettamento di più record logici in un unico blocco. Allocare lo spazio disco in termini di blocchi vuol dire che l'ultimo blocco di un file non sarà completamente pieno causando **frammentazione interna**, maggiore e' la dimensione del blocco e maggiore sarà la frammentazione interna.

## 5.5.3 Directory

Come abbiamo già visto le directory devono essere mantenute in **memoria secondaria** si copiano in memoria centrale solo quelle in utilizzo. Anche le directory quindi vengono trattate come file, possono essere **organizzate** dal S.O. in diversi modi:

1. Ogni entry di una directory contiene il nome e gli alti attributi del file
2. Ogni entry di una directory contiene il nome e un riferimento ad una struttura separata che contiene gli altri attributi del file (vedremo questa nello specifico)

### Organizzazione di Una Directory

L'organizzazione più flessibile e' quella che prevede che le informazioni che a livello astratto devono essere memorizzate all'interno di un directory siano collocate in due strutture dati separate.

1. Directory dei Nomi dei File (**DNF**) mantiene la corrispondenza fra **nome logico** del file e un identificatore interno assegnato dal S.O. In unix l'id interno si chiama i-number e quindi in una directory per ogni file si ha: `<filename, i-number>` Esiste una DNF per ogni directory definita dall'utente.
2. Directory Base dei File (**DBF**) Mantiene tutte le altre informazioni di un descrittore astratto di file, in unix e' l'insieme degli i-node. L'indicatore interno che si trova nella DNF, viene usato come indice all'interno della DBF.

**Vantaggio**: questa organizzazione consente di realizzare facilmente il meccanismo di **link hw**. Due entry differenti in due DNF che fanno riferimento allo stesso identificatore, una sola entry nella DBF e contatore d'uso nella entry della DBF

**Osservazione**: la DNF della directory radice e la DBF hanno un indirizzo fisico fisso all'interno del disco

Nel momento in cui si deve **accedere** ad un file, bisogna tramite il suo nome accedere al DNF interessato e ritrovare l'**identificatore interno**, trovare nella DBF l'indirizzo fisico di partenza, oltre a tutte le altre informazioni. **Ottimizzazione**: con l'operazione open, si porta, si porta in memoria il valore dell'id e la riga corrispondente del DBF in una struttura dati mantenuta in memoria centrale → File Control Block (FCB). Questi FCB vengono raccolti in una struttura dati detta **tabella dei file aperti**. In unix si hanno:

1. Una tabella dei file aperti per ogni processo
2. Due tabelle di kernel (globali):
    1. Tabella dei file aperti di sistema
    2. Tabella dei file/i-node attivi di sistema

**Realizzazione delle Directory (DNF)**

1. Lista lineare di nomi di file
    - **Vantaggio**: semplice da implementare
    - **Svantaggio**: esecuzione onerosa dal punto di vista del tempo di ricerca, complessità lineare nel numero di elementi contenuti nella directory
2. Lista ordinata
    - **Vantaggio**: migliora il tempo di ricerca. Utile per produrre l'elenco ordinato dei file contenuti nelle directory
    - **Svantaggio**: l'ordinamento deve essere mantenuto a fronte di ogni inserimento/cancellazione
3. Tabella hash: lista lineare con struttura hash
    - **Vantaggio**: migliora il tempo di ricerca nella directory, inserimento e cancellazione costano O(1), se non si verificano collisioni
    - **Svantaggio**: dimensione fissa e necessità di rehash

## 5.5.4 Montaggio di un File System Fisico

Un file system fisico deve essere montato prima di poter essere acceduto dagli utenti e quindi dai processi di un S.O. La posizione dove viene montato vien detta mount point o punto di montaggio. Alcuni S.O. richiedono che i file system fisici che possono essere montati siano solo di un tipo prefissato, altri ne supportano diversi e sondano le strutture del dispositivo per determinare il tipo di file system fisico presente.

### Procedura di montaggio

Innanzitutto si deve fornire al S.O. il nome del dispositivo da montare e il punto di montaggio, normalmente il punto di montaggio e' una directory vuota cui sarà agganciato il file system fisico che deve essere montato. La directory su cui viene montato un file system fisico punto anche non essere vuota, ma nel momento in cui si effettua il montaggio, i dati ivi contenuti non sono più visibili fino all'operazione di **unmount**. Una volta montato, il file system fisico risulta accessibile agli utenti (e ai loro processi). In unix Linux e MacOS diventa parte integrante del file system logico. Il montaggio può avvenire in maniera automatica (MacOS, Windows) oppure manuale (Linux).

# 6. Gestione dei File - Livello Fisico

## 6.1 Realizzazione del File System

Le funzioni del file system vengono realizzate tramite chiamate di sistema (API), che utilizzano **dati** gestiti dal S.O. residenti sia su disco che in memoria.

1. Strutture dati del file system residenti su disco:
    1. Blocco di controllo di avviamento (Boot Control Block): contiene le informazioni per l'avviamento di un S.O. da quel volume, ad esempio boot block per Unix e partition boot sector per Windows. Questo blocco e' necessario se il volume contiene un S.O. e normalmente occupa il primo posto nel volume.
    2. Blocchi di controllo dei volumi (Volume Control Block): contengono dettagli riguardanti la partizione, quali numero totale dei blocchi e loro dimensione, contatore dei blocchi liberi e relativi puntatori; ad esempio superblocco UFS in Unix e master file table in NTFS

    3. Strutture delle directory: usate per organizzare i file, ad esempio in UFS comprendono i nomi dei file e i numeri di i-node
    4. Descrittori dei file: contengono dettagli sul file come permessi, dimensione, date di creazione/ultimo accesso/modifica, puntatori ai blocchi di dati. NTFS, memorizza questi dati nella master file table utilizzando una struttura stile DB relazionale
2. Strutture dati del file system residenti in memoria:
    1. Tabella di montaggio: contiene le informazioni relative a ciascun volume montato
    2. Directory cache: contiene informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente
    3. Tabella dei file attivi (di sistema): contiene una copia del descrittore di file per ciascun file aperto nel sistema insieme con altre informazioni **FCB**.
    4. Tabella dei file aperti per ciascun processo: contiene un puntatore all'elemento corrispondente nella tabella di sistema, più informazioni di accesso specifiche del processo
    5. Buffer per la lettura scrittura

# 6.2 Partizioni e Montaggio

Una partizione può essere un volume contenente un S.O. o essere dotata della sola formattazione di basso livello cioè una sequenza non strutturata di blocchi. Nella fase di avviamento, il S.O. non ha ancora caricato i driver dei dispositivi e quindi non può usare i servizi messi a disposizione dal file system (non può interpretarne il formato). La partizione di avviamento e' una serie sequenziale di blocchi, che si carica in memoria come un'immagine. Questa immagine può contenere più informazioni di quelle necessarie al caricamento di un unico S.O. (boot loader). Nella fase di caricamento del S.O., si esegue il montaggio della partizione radice (root partition), che contiene il kernel del S.O. ed eventualmente altri file di sistema. A seconda del S.O., il montaggio degli altri volumi avviene automaticamente in questa fase o si può compiere successivamente in modo esplicito.

# 6.3 Gestione dello Spazio su Disco

Abbiamo visto come viene gestita la memoria centrale di sistema, chiaramente anche quella secondaria ha bisogno di logiche e politiche per la sua gestione. Bisogna tenere traccia dei **blocchi liberi** e di quelli **allocati** ai file. I fattori di cui deve tenere conto per una buona strategia di allocazione sono:

1. Velocità di esecuzione dell'accesso sequenziale, dell'accesso casuale e dell'allocazione/deallocazione dei blocchi: data la maggiore frequenza degli accessi nei confronti delle operazioni di allocazione/deallocazione, la velocità degli accessi risulta più importante
2. Possibilità di utilizzare trasferimenti multipli di settori: i trasferimenti multipli di settori consentono di limitare il tempo di accesso
3. Grado di utilizzo del disco: si intende la percentuale di spazio totale del disco allocabile agli utenti, la frammentazione (esterna) abbassa il grado di utilizzo
4. Quantitativo di memoria centrale necessario per ogni algoritmo: alcuni algoritmi necessitano di strutture dati mantenute in memoria centrale

La gestione dello spazio su disco assomiglia al problema di gestione della memoria centrale ma con **due nuovi aspetti**:

1. Gli **accessi** ai dischi sono di alcuni ordini di grandezza **più lenti**
2. Il **numero di blocchi** e' almeno di un ordine di grandezza **più grande**

Inoltre c'è una elevata variabilità nelle dimensioni dei file, risulta quindi impossibile fare delle previsioni sulle richieste delle risorse. La gestione dei blocchi su disco comporta una **frammentazione interna** indipendente dalla politica di allocazione che viene adottata: dipende dalla dimensione del blocco e da quella del file. Lo spreco medio risulta di mezzo blocco per file, osserviamo la similitudine con il problema di frammentazione interna nel caso di paginazione. Alcuni sistemi adottano uno schema di allocazione che si basa su **cluster** di allocazione (numero intero di blocchi) riducendo il tempo per l'allocazione ma aumentando la frammentazione interna. Come abbiamo visto per la memoria centrale i metodi di allocazione per quella secondaria si suddividono in:

1. Allocazione contigua
2. Allocazione non contigua

## 6.3.1 Allocazione Contigua

Ogni file viene allocato su blocchi di disco contigui, per reperire il file occorrono solo la locazione iniziale e la lunghezza.

### Vantaggi

1. Questo approccio comporta un **accesso sequenziale** e diretto **facilitato**
2. Possibilità di sfruttare trasferimenti multi-settore

### Svantaggi

1. Tende a generare **frammentazione esterna**, la formazione di cluster troppo piccoli, ma che globalmente sarebbero sufficienti per le richieste di uno o più file.

   Per far fronte a questo problema bisogna effettuare una **compattazione periodica** (squeeze o deframmentazione), deve essere effettuata fuori linea per evitare accessi ai file e risulta essere piuttosto costosa in termini di risorse.

2. La allocazione contigua richiede che le **dimensioni** del file siano dichiarate **in anticipo** al momento della creazione, questo risulta facile nel caso di creazione di file per effettuare una copia ma difficile in caso di creazione di file in generale, ad esempio, per una fase di **editing**.

   Per risolvere questo problema si effettua una **stima iniziale** della dimensione del file e se poi questa viene superata:

   - Si genera un errore: questo comporta perdita di tempo per l'utente e di spazio per il sistema in caso di sovrastime.
   - Si ricopia il file in un cluster di maggiore dimensione: perdita di tempo per il sistema
   - Si usa uno (o più) cluster non contiguo: bisogna fare attenzione ai casi di **file overflow**, periodicamente (ad esempio durante il compattamento) si può ricostruire la contiguità fisica.
3. Settori difettosi: l'allocazione contigua rende difficile "aggirare" i settori di disco difettosi, aumentando la frammentazione esterna.

## 6.3.2 Allocazione non Contigua

Per quanto riguarda le politiche di allocazione non contigua vedremo principalmente le varianti di **concatenamento** e **indicizzazione**.

## Concatenamento

Ogni file e' una **lista** concatenata **di blocchi**, sparsi ovunque nel disco, vengono riservati alcuni byte in ogni blocco (o in ogni cluster di allocazione) per **puntare** al blocco successivo (viene inserito il numero del prossimo blocco). Quando un file deve essere **creato** si crea un descrittore del file (in una directory), che punta a NIL.

**Vantaggi**

1. Eliminato il problema della **frammentazione esterna**, non e' più necessaria la compattazione
2. Risulta facile "aggirare" i settori di disco difettosi, basta eliminarli dalle liste dei file oppure dalla lista dei blocchi liberi
3. Accesso sequenziale facilitato
4. Possibilità di modificare la dimensione del file su necessità

**Svantaggi**

1. Impossibilità di usare trasferimenti multisettore
2. Accesso diretto **impossibile**: richiederebbe di accedere a tutti i blocchi che precedono nella lista
3. Possibile deterioramento dei puntatori

I punti 2 e 3 possono essere risolti mantenendo i puntatori in una tabella dedicata su disco, quindi per quanto riguarda il 2, basta copiare questa tabella in memoria centrale per velocizzare gli accessi casuali, mentre per il 3, basta avere copie ridondanti della tabella su disco, come garanzia per eventuali deterioramenti. Questo metodo nello specifico viene utilizzato in Windows e si chiama **F**ile **A**llocation **T**able (**FAT**) e per contenere la FAT si riserva una sezione del disco all'inizio di ciascun volume.

#### Allocazione File con FAT

FAT indica la File Allocation Table, **cluster** invece un aggregazione di settori adiacenti (uno o più blocchi adiacenti). La FAT non e' altro che una tabella che contiene i riferimenti ai cluster liberi ed occupati, quindi al suo interno ha un elemento per ogni cluster. In ogni disco ci sono **due copie** della FAT. Sia la FAT che la directory radice () sono memorizzati in posizioni fisse del disco.

## Indicizzazione

L'allocazione indicizzata e' simile al metodo precedente dato che mantiene i puntatori ai blocchi allocati. La differenza consiste nel **miglioramento** nella velocità di accesso diretto: i puntatori vengono raggruppati in **blocchi indice**. Il descrittore del file (sempre considerato dal punto di vista astratto) **non** contiene l'indirizzo del primo blocco su disco, ma l'indirizzo del blocco indice.Il blocco indice copre una funzione simile a quella della tabella delle pagine. Quando un **file** deve essere **creato** si crea un descrittore del file (in una directory), che punta al blocco indice che contiene tutti i puntatori a **NIL**.

**Vantaggi**

Come nel concatenamento viene eliminata la frammentazione esterna e si possono aggirare i settori difettosi. In più l'**accesso sequenziale e diretto** sono possibili solo se il blocco indice viene portato in memoria.

**Svantaggi**

1. File corti: nel caso in cui ci siano file di piccole dimensioni si spreca una grande quantità di memoria con il blocco indice, per questo il blocco indice deve essere il **più piccolo** possibile tenendo conto però del punto 2.

   Un'alternativa per risolvere questo problema consiste nel mantenere in ogni descrittore di file un certo numero di puntatori ai primi blocchi del file (sufficienti per file corti) e un puntatore ad una lista di blocchi indice o vari puntatori a blocchi indice di primo e secondo livello (vedi punto 2)

2. Dimensione massima dei file: La dimensione massima dei file dipende dalla dimensione del blocco e da quella dei puntatori. In genere la dimensione del blocco indice e' quella di un blocco.

   > Esempio
   >
   > Con un blocco indice di 512 Byte e una dimensione dei puntatori di 4 Byte (128 puntatori) si hanno $128 * 512 = 64KB$ di dimensione massima di un file. Limite chiaramente **inaccettabile**.

   Per rimediare a questo tipo di problema si ricorre all'indicizzazione a più livelli con una delle seguenti implementazioni:

   1. Il blocco indice contiene come ultimo puntatore NIL se il file e' piccolo (singolo livello di indicizzazione), altrimenti contiene il puntatore ad un altro blocco indice (secondo livello di indice), etc...
   2. Il primo blocco indice contiene i puntatori a blocchi indice (secondo livello), etc... In genere due livelli di indici sono sufficienti

## 6.3.3 Conclusioni

In conclusione il miglior metodo per l'allocazione dei file dipende dal tipo di accesso:

- Il metodo di allocazione contigua ha ottime prestazioni sia per l'accesso sequenziale che casuale
- Il metodo di allocazione concatenata si presta in modo molto naturale per l'accesso sequenziale

Alcuni S.O. combinano questi due metodi di allocazione in base alla modalità di accesso. Questo implica che al momento della creazione si specifichi anche se la modalità di accesso e' casuale (allocazione contigua) oppure sequenziale (allocazione concatenata)

- Il metodo di allocazione indicizzata risulta essere il più complesso: l'accesso ai dati può richiedere più accessi al disco (tre, nel caso di un indice a due livelli), i blocchi indice devono essere caricati in memoria centrale e necessita ottimizzazioni nel caso di file corti

  La soluzione più semplice e' quella di mantenere in un **array di bit** lo **stato** del blocco corrispondente su disco: **libero** (1) o **allocato** (0) → **bit map**. In questo modo, per trovare gli spazi liberi basta iterare il vettore cercando bit con valore 1, copiando il vettore in memoria centrale si queste operazioni possono essere effettuate velocemente. Per i dischi attuali ci possono essere difficoltà nel mantenere la bitmap in RAM. Questo metodo si addice molto bene alla gestione dei file contigui, in alternativa all'array si potrebbe implementare come:

  1. Lista concatenata: si usa lo **stesso** schema di allocazione dei file con allocazione non contigua a concatenamento. Quindi si collegano tutti i blocchi liberi mediante puntatori (l'ultimo blocco indica in modo opportuno il termine della lista) e si mantiene un puntatore alla testa della lista in una zona riservata del disco, solo questo puntatore alla testa viene copiato in memoria centrale minimizzando gli sprechi di spazio.

**Osservazioni**: quando si cerca un solo blocco libero la lista così realizzata risulta efficiente (si stacca il primo blocco libero e si riconcatena il puntatore alla testa col secondo). Quando invece si cercano n blocchi liberi consecutivi si rischia di dover scorrere tutta la lista, con tempi di attesa piuttosto lunghi (bassa efficienza). Nella FAT, l'informazione sui blocchi liberi viene inclusa nella struttura dati per l'allocazione e non richiede quindi un metodo di gestione separato. Vediamo come risolvere questo problema di efficienza nei prossimi punti

2. Il primo blocco libero contiene gli indirizzi di altri $n-1$ blocchi liberi più un puntatore al successivo (anch'esso con la stessa struttura), la lista si trasforma quindi in un albero a 2 livelli
3. Counting: se ci sono n blocchi liberi consecutivi viene memorizzato un puntatore al primo e poi il numero di blocchi, si mantiene una lista contenente un indirizzo del disco, che indica un blocco libero, ed un contatore (che indica da quanti altri blocchi liberi contigui e' seguito. La lista si accorcia drasticamente

# 6.4 Osservazioni Generali

## 6.4.1 Efficienza

L'**efficienza** dipende da:

- Tecniche di allocazione del disco e algoritmi di realizzazione/gestione delle directory
- Tipi di dati conservati nel descrittore del file
- Preallocazione delle strutture necessarie a mantenere i metadati
- Strutture dati a lunghezza fissa o variabile

## 6.4.2 Prestazioni

Le **prestazioni** dipendono da:

- Mantenere dati e metadati "vicini" nel disco
- Disporre di **buffer cache**, cioè sezioni dedicate della memoria centrale in cui si conservano i blocchi usati di frequente. Se devono essere effettuate scritture sincrone risulta impossibile usare il buffering/caching dato che l'operazione di scrittura su disco deve essere completata prima di proseguire l'esecuzione. Le scritture asincrone, che sono le più comuni, sono invece bufferizzabili.

Sempre in merito alle prestazioni bisogna sottolineare che l'I/O mappato in memoria impiega una cache delle pagine, mentre l'I/O da file system utilizza la buffer cache del disco (in memoria centrale). Molti S.O. unificano la cache del disco con la cache delle pagine, una buffer cache unificata prevede l'utilizzo di un'**unica cache** per memorizzare sia le pagine dei file mappati in memoria che i blocchi trasferiti per operazioni di I/O ordinario da file system. L'algoritmo più ragionevole per questo tipo di cache e' **LRU** a meno che non si tratti di letture **sequenziali** in qual caso si ricorre ai metodi **free-behind** e **read-ahead** che consistono nel rimuovere una pagina dalla cache non appena si verifica la richiesta della pagina successiva e nel caricare in anticipo alcune delle pagine successive a quella appena richiesta. Un ulteriore metodo per aumentare le prestazioni di lettura e scrittura su disco e' creare **dischi RAM** che consiste nel adibire parte della RAM ad un disco virtuale, il cui contenuto viene periodicamente riscritto su un supporto permanente.

### 6.4.2 Ripristino (Recupero)

I file e le directory sono mantenuti sia in memoria RAM (parzialmente) che su disco, bisogna assicurarsi che malfunzionamenti del sistema non comportino la perdita di dati o la loro incoerenza. Se **blocchi critici** vengono modificati ma non salvati mai (perché acceduti frequentemente) si rischia l'inconsistenza in seguito ai crash, si devono quindi dividere i blocchi in due categorie:

1. Blocchi **critici**: blocchi necessari a garantire la consistenza del file system, ogni modifica deve essere immediatamente trasferita al disco
2. Blocchi **dati**: se riutilizzato a breve, rimane nella cache

Le modifiche ai blocchi dati devono essere riportate su disco anche prima della loro sostituzione, in modo:

- Asincrono: ogni 20-30 secondi (Unix, Windows)
- Sincrono: ogni scrittura viene immediatamente trasferita anche al disco (write through cache)

Ci possono essere altri problemi se il crash si verifica in situazioni di modifica di metadati. Se si ha un crash si possono avere incoerenze fra le strutture, il contatore dei descrittori liberi potrebbe indicare un descrittore allocato, ma la directory non contenere ancora un puntatore all'elemento relativo! Esistono due approcci al problema della consistenza del file system:

1. Curare le inconsistenze dopo che si sono verificate, con programmi di controllo della consistenza. Si deve utilizzare un **verificatore di coerenza** come **fsck** in Linux e **chkdsk** in Windows. Questo tipo di programma confronta i dati nella struttura di directory con i blocchi di dati sul disco e tenta di fissare le eventuali incoerenze, sono lenti e non sempre funzionano.
2. Prevenire le inconsistenze tramite i **Journaled File System** (JFS).

I dispositivi di memoria di massa hanno un Mean Time Between Failures (MTBF) relativamente breve, quindi i sistemisti hanno dovuto implementare soluzioni per aumentarne l'affidabilità:

1. Aumentare l'affidabilità dei dispositivi, ad esempio tramite configurazioni **RAID**
2. **Backup** (automatico o manuale) dei dati dal disco ad altro supporto attraverso:
    1. Dump fisico: direttamente i blocchi del file system, veloce ma difficilmente incrementale e non selettivo
    2. Dump logico: porzioni del file system, può essere completo, differenziale o incrementale; può essere più selettivo, ma a volte troppo astratto (link, file con buchi, etc...). In ogni modo, il recupero dei file perduti (o interi file system) dal backup può essere fatto o dall'amministratore o direttamente dall'utente.

## 6.5 Generalizzazione del Concetto di File

Il concetto di file può essere usato anche come astrazione del sistema di ingresso/uscita. Questo vuol dire che anche le periferiche come ad esempio mouse e tastiera sono trattate come file. L'utente può usare un unico ed uniforme insieme di servizi per trattare i file e l'I/O che e' indipendente dalle periferiche. I programmi usano un'**astrazione** dei dispositivi → standard input e standard output. Il collegamento fra l'**astrazione** e il dispositivo avviene a tempo di esecuzione usando la **ridirezione** mediante appositi metacaratteri (<,>,|) e chiamate di sistema (close, open, create). Chiaramente esistono associazioni di **default** come standard input → tastiera e standard output → video. In Unix, per fornire questa astrazione unica le periferiche stesse sono viste come **file**, in particolare sono situate nella sottodirectory `/dev` in questa directory troviamo file

organizzati a blocchi (dischi) e file organizzati a caratteri (terminali e stampanti). Su tutti questi dispositivi/file si utilizzano le operazioni di open, close, read e write. In alcuni S.O. viene fornito un meccanismo detto **pipe** che permette di fare comunicare due processi, stabilisce un canale di comunicazione unidirezionale con bufferizzazione. Da un lato della pipe, un processo scrive dei dati e , dall'altro lato un altro processo li può leggere. Si usano le stesse operazioni che si usano per file e dispositivi.

## 6.5.1 Network File System

Il Network File System (NFS) e' un buon esempio di **file system di rete**. Definito e implementato dalla SUN sulla base dei protocolli TCP/IP risulta disponibile nella maggior parte delle distribuzioni Unix e Linux e in alcuni S.O. per PC. Nel contesto del NFS, si considera un insieme di stazioni di lavoro interconnesse in rete come un insieme di stazioni indipendenti con file system indipendenti. Lo scopo dell'NFS e' quello di consentire un certo grado di condivisione fra questi file system, sulla base di richieste esplicite, ma **in modo trasparente all'utente**. La condivisione e' basata su una **relazione client-server** una stazione può essere sia un client che un server. Affinché una directory remota sia accessibile in modo trasparente a una certa stazione cliente, questa deve prima effettuare una operazione di **montaggio** nel proprio file system locale, la directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito. Dopo tale operazione la directory remota risulta accessibile in **modo trasparente** tramite il file system locale. Potenzialmente ogni file system o ogni directory in un file system, nel rispetto dei diritti di accesso, si può montare in modo remoto in una qualsiasi directory locale. Se si monta un file system condiviso in corrispondenza delle directory iniziali degli utenti in tutte le stazioni in una rete locale, un utente può aprire una sessione di lavoro e usare i propri file da una qualunque delle stazioni in rete → **mobilità degli utenti**.