(segue **GESTIONE DEI FILE**)

REALIZZAZIONE DEL FILE SYSTEM

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

A) 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* nell'UFS (UNIX), *partition boot sector* nei sistemi Windows
   - Necessario se il volume contiene un S.O.
   - Normalmente è il primo blocco del 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* in UFS (UNIX), *master file table* (MFT) nell'NTFS (Windows)

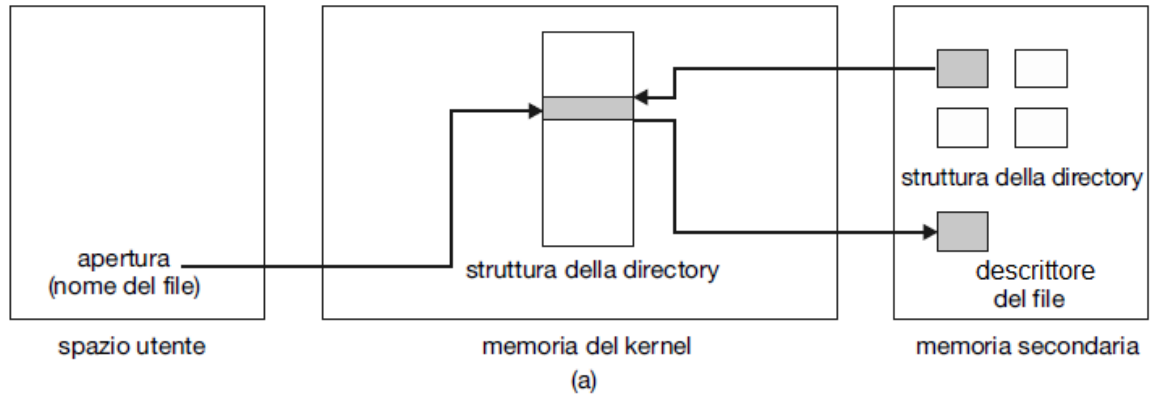
3) **Strutture delle directory**: usate per organizzare i file (in UFS comprendono i nomi dei file e i numeri di inode associati) ➔ DNF

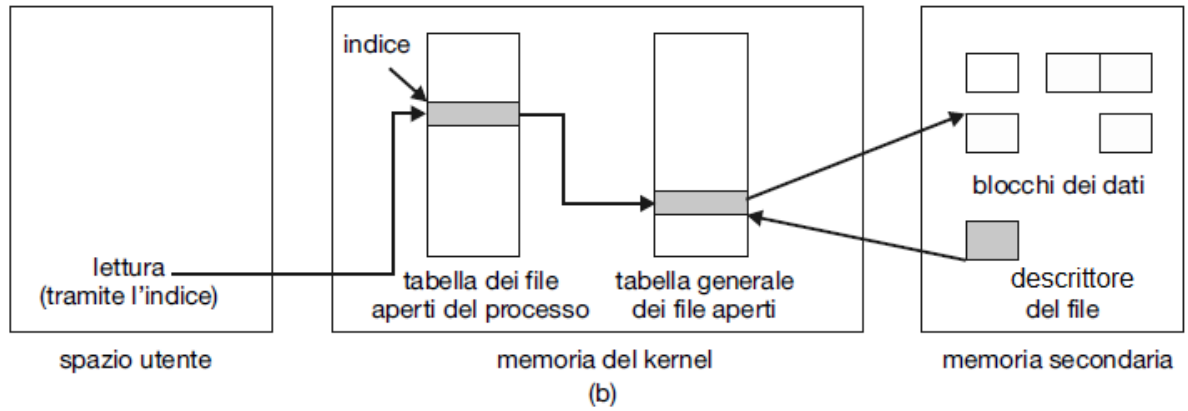
4) **Descrittori dei file**: contengono dettagli sul file come permessi, dimensione, date di creazione/ultimo accesso/ultima modifica, puntatori ai blocchi di dati ➔ DBF (I-node nell'UFS)
   Invece, NTFS (Windows) memorizza questi dati nella *master file table* utilizzando una struttura stile DB relazionale

(segue **REALIZZAZIONE DEL FILE SYSTEM**)

B) 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[1]
5) **Buffer** per la lettura/scrittura





(a) Apertura di un file
(b) Lettura da file (tralasciando la visualizzazione dei buffer di lettura)

[1] In UNIX esiste anche un'altra tabella di sistema: tabella dei file aperti di sistema!

(segue **GESTIONE DEI FILE**)

PARTIZIONI E MONTAGGIO

Una partizione può essere un volume contenente un S.O. (cooked), o essere dotata della sola formattazione di basso livello (*raw*) cioè una sequenza "non strutturata" di blocchi

Esempio: UNIX usa una partizione raw per l'area di avvicendamento dei processi (cioè per lo *swapping*)

Le informazioni relative all'avviamento del S.O. si registrano in una partizione apposita, con formato proprio

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 è 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)

➔ si può ottenere quindi un PC dual–boot (come ad esempio quelli dei laboratori didattici Linux/Windows)

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

➔ in ogni modo, il S.O. annota nella tabella di montaggio, residente in memoria, che un file system (di un dato tipo) è stato montato

(segue **GESTIONE DEI FILE**)

GESTIONE DELLO SPAZIO SU DISCO

➔ gestione dello spazio sulla **MEMORIA SECONDARIA**

Bisogna tenere traccia dei **BLOCCHI LIBERI** e di quelli **ALLOCATI** ai file

FATTORI di cui deve tenere conto una buona strategia di allocazione:

1) velocità di esecuzione dell'accesso sequenziale, dell'accesso casuale e dell'allocazione/deallocazione dei blocchi
2) possibilità di utilizzare trasferimenti multipli di settori
3) grado di utilizzo del disco
4) quantitativo di memoria CENTRALE necessario per ogni algoritmo

Aspetti da tenere in conto per:

1) ➔ Data la maggiore frequenza degli accessi nei confronti delle operazioni di allocazione/deallocazione, la velocità degli accessi risulta più importante

2) ➔ i trasferimenti multipli di settori consentono di limitare il tempo di accesso

3) ➔ si intende la percentuale di spazio totale del disco allocabile agli utenti
La frammentazione (esterna) ABBASSA il grado di utilizzo

4) ➔ alcuni algoritmi necessitano di strutture dati mantenute in memoria centrale *(ulteriori rispetto alle strutture dati precedentemente descritte)*

(segue **GESTIONE DELLO SPAZIO SU DISCO**)

La gestione dello spazio su disco assomiglia al problema di gestione della memoria centrale ma con …

DUE NUOVI ASPETTI:

a) gli **accessi ai dischi** sono di alcuni ordini di grandezza ***più lenti***

b) il **numero di blocchi** è almeno di un ordine di grandezza ***più grande***

Inoltre, c'è una elevata variabilità nelle dimensioni dei file ➔ difficile fare delle previsioni sulle richieste di blocchi

OSSERVAZIONE IMPORTANTE:

La gestione dei blocchi su disco comporta una **FRAMMENTAZIONE INTERNA** (si riveda slide 29 di Gestione dei file) che è *indipendente* dalla politica di allocazione che viene adottata ➔ dipende dalla dimensione del BLOCCO e da quella del file

➔ **"spreco" medio di mezzo BLOCCO per file**

➔ **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** (un numero intero di BLOCCHI)

VANTAGGIO: si riduce il tempo per l'allocazione
SVANTAGGIO: si aumenta la frammentazione interna

METODI DI ALLOCAZIONE

I) ALLOCAZIONE CONTIGUA

II) ALLOCAZIONE NON CONTIGUA

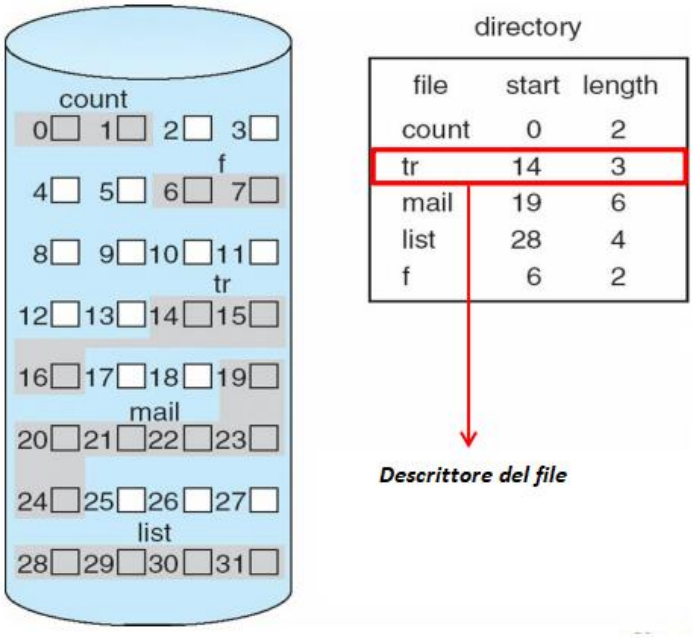
* CONCATENAMENTO
* INDICIZZAZIONE

(segue **METODI DI ALLOCAZIONE**)

ALLOCAZIONE CONTIGUA

Ogni file è allocato su BLOCCHI di disco contigui
➔ per reperire il file occorrono solo la locazione iniziale (numero di blocco iniziale) e la lunghezza (numero di blocchi)

Esempio: (*NB*: il numero di blocchi del disegno ha uno scopo puramente didattico; del descrittore del file, *mostrato in termini astratti*, sono riportate sono le informazioni essenziali)



OSSERVAZIONE:
Quando un **file** deve essere **CREATO** si crea un descrittore del file (in una directory), che punta al blocco iniziale e devono essere allocati tutti i blocchi che servono al file

VANTAGGI:

1) Accesso sequenziale e diretto sono facilitati
2) POSSIBILITÀ di sfruttare trasferimenti multi-settore

SVANTAGGI:

1) **FRAMMENTAZIONE ESTERNA**

➔ formazione di CLUSTER troppo piccoli, ma che globalmente sarebbero sufficienti per le richieste di uno o più file

SOLUZIONE: **compattazione periodica** (SQUEEZE o *deframmentazione*) ➔ deve essere effettuata fuori linea per evitare accessi ai file ➔ costosa!

(segue **ALLOCAZIONE CONTIGUA-SVANTAGGI**)

2) DIMENSIONE FILE

La allocazione contigua richiede che le **dimensioni** del file siano ***dichiarate in anticipo*** al momento della creazione

➔ facile nel caso di creazione di file per effettuare una copia

➔ difficile in caso di creazione di file in generale, ad esempio, per una fase di ***EDITING***

SOLUZIONI:

tutte basate su una stima iniziale e se poi il file non ci sta

* si genera un ERRORE
  - ➔ **perdita di tempo per l'utente**
  - ➔ **perdita di spazio per il sistema** a causa di SOVRASTIME
* si ricopia il file in un CLUSTER più grande
  - ➔ **perdita di tempo per il sistema**
* si usa uno (o più) CLUSTER non contiguo
  - ➔ **FILE OVERFLOW**

  Periodicamente (ad esempio, durante la compattazione) si può ricostruire la contiguità FISICA

3) SETTORI DIFETTOSI

La allocazione contigua rende difficile "aggirare" i settori di disco difettosi ➔ aumenta la frammentazione esterna

ESEMPIO: Nel file system **VxFS**[2] (originariamente sviluppato dalla Veritas Software agli inizi degli anni '90), il disco viene allocato con granularità maggiore della dimensione del blocco fisico (**extent**: una "porzione di spazio contiguo") ➔ Ciascun file consiste di uno o più extent (di dim. variabile ed eventualmente definita dall'utente) ➔ Inizialmente, per ciascun file viene allocato un extent ➔ Se questo non è sufficientemente grande, si aggiunge un'ulteriore extent ➔ Il descrittore del file contiene l'indirizzo iniziale del primo extent, la sua dimensione e un puntatore al primo blocco dell'extent successivo

[2] È un journaling File System

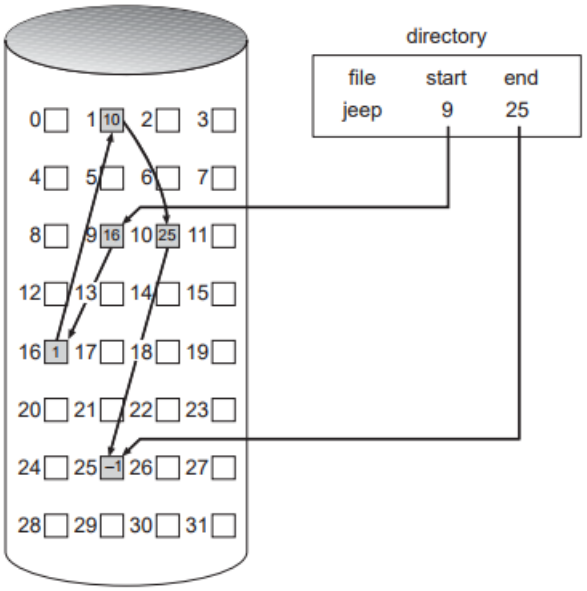
(segue **METODI DI ALLOCAZIONE**)

ALLOCAZIONE NON CONTIGUA: CONCATENAMENTO

Ogni file è allocato utilizzando 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!

Esempio:



Il descrittore del file (*sempre considerato dal punto di vista astratto*) contiene l'indirizzo del blocco iniziale (ed eventualmente di quello finale)

➔ il S.O. inserisce nell'ultimo blocco del file un puntatore NULLO (ad esempio **-1**, nella figura)

OSSERVAZIONE: 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 è più necessario la compattazione
2) Risulta facile "aggirare" i settori di disco difettosi
   ➔ basta eliminarli dalle liste dei file oppure dall'insieme dei blocchi liberi
3) Accesso sequenziale facilitato

(segue **CONCATENAMENTO**)

4) Possibilità di modificare la dimensione del file su necessità

OSSERVAZIONE: Lo spazio necessario in ogni blocco (o cluster) per memorizzare i PUNTATORI è solitamente minore del 1%, quindi la sua influenza sul grado di utilizzo del disco è trascurabile

SVANTAGGI:

1) Impossibilità di usare trasferimenti multisettore
2) **ACCESSO DIRETTO impossibile:** richiederebbe di accedere a tutti i blocchi che precedono nella lista
3) Possibilità di deterioramento dei PUNTATORI

SOLUZIONE per 2 e 3:
mantenere i puntatori in una tabella DEDICATA su disco (non si spreca spazio nei blocchi di file), quindi per risolvere

- il problema 2), basta copiare questa tabella in memoria centrale per velocizzare gli accessi casuali
- il problema 3), basta avere copie ridondanti della tabella su disco, come garanzia per eventuali deterioramenti

ESEMPIO: metodo usato in **MS-DOS**, **OS/2** e **Windows** ➔ **F**ile **A**llocation **T**able (**FAT**) in duplice copia

Per contenere la FAT, si riserva una sezione del disco all'inizio di ciascun volume

La FAT ha un elemento per ogni cluster del disco

Il descrittore del file contiene il numero del primo cluster del file ➔ con questo numero si 'entra' nella FAT e lì si trova il numero del cluster successivo

Si vedano i dettagli nella slide seguente …

(segue **CONCATENAMENTO: il caso della FAT**)

ALLOCAZIONE FILE CON FAT

CLUSTER ➔ aggregazione di settori adiacenti (può essere un blocco o più blocchi adiacenti)

ALLOCAZIONE NON CONTIGUA CON VARIANTE DEL METODO A CONCATENAMENTO

FAT (File Allocation Table)

➔ TABELLA che contiene i riferimenti ai cluster liberi ed occupati

➔ un elemento per ogni CLUSTER

N.B.: DUE COPIE della FAT per ogni disco (per ridondanza)

Sia la FAT che la directory radice (\) sono chiaramente memorizzati in **posizioni fisse** del disco

ESEMPIO DI FAT:

(consideriamo il caso di FAT12, usata per i floppy disk, ma l'organizzazione delle FAT16 e FAT32 sono analoghe)

VALORI SPECIALI:

0 CLUSTER LIBERO
FFFh ultimo CLUSTER di un file
FF7h CLUSTER difettoso

DIRECTORY: **F ===> 2**

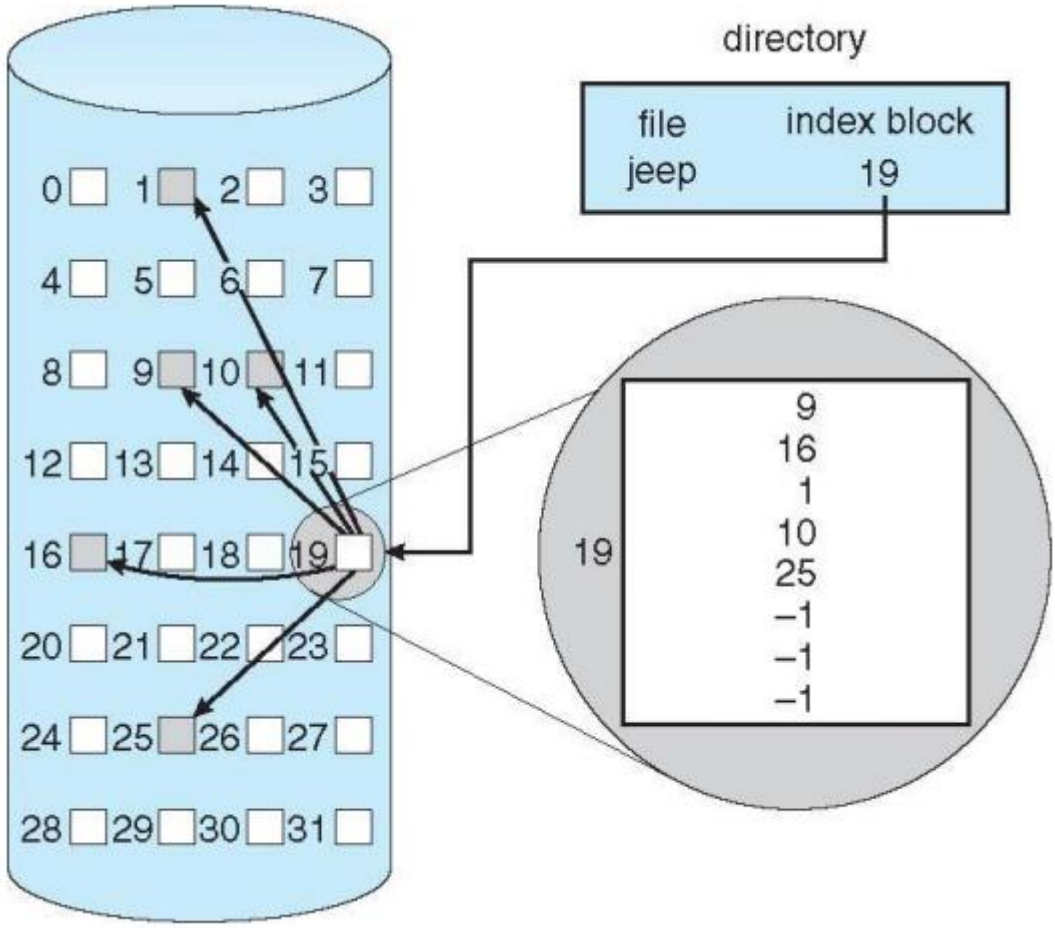
Elem.	Valore (in Hex)	Significato
0	FD	disco doppia faccia, 9 settori per traccia
1	FFE	non usato
2	3	prossimo cluster del file F è il 3
3	5	prossimo cluster del file F è il 5
4	FF7	inutilizzabile
5	**FFF**	ultimo cluster del file F (FINE CATENA)

(segue **METODI DI ALLOCAZIONE**)

ALLOCAZIONE NON CONTIGUA: INDICIZZAZIONE

L'allocazione indicizzata è 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**

Esempio:



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**

Nota:
BLOCCO INDICE ➔ simile ad una TABELLA delle PAGINE

(segue **INDICIZZAZIONE**)

OSSERVAZIONE: 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 caso del concatenamento

1) Eliminato il problema della **FRAMMENTAZIONE ESTERNA** ➔ non è più necessario la compattazione
2) Risulta facile "aggirare" i settori di disco difettosi
   ➔ basta eliminarli dalle liste dei file oppure dalla lista dei blocchi liberi

OSSERVAZIONE: L'accesso sequenziale e diretto sono possibili solo se il blocco indice viene portato in memoria

SVANTAGGI:

1. FILE CORTI

Nel caso di avere ***file corti*** si **SPRECA** una grande quantità di memoria con il BLOCCO INDICE

➔ bisognerebbe fare il BLOCCO INDICE ***più piccolo*** possibile

MA se il blocco indice è troppo piccolo abbiamo dei problemi con la

2. DIMENSIONE MASSIMA DEI FILE

La dimensione massima dei file dipende dalla dimensione del BLOCCO e da quella dei PUNTATORI

NOTA: in genere, la dimensione del BLOCCO INDICE è quella di un BLOCCO

(segue **ALLOCAZIONE CON INDICIZZAZIONE - SVANTAGGI**)

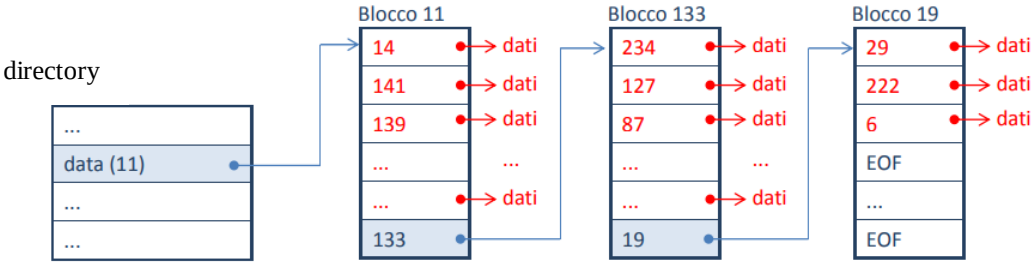
ESEMPIO:
con un BLOCCO INDICE di 512 byte e una dimensione dei puntatori di 4 byte ➔ 128 PUNTATORI
128*512 ➔ 64 Kb dimensione MASSIMA di un file

In genere, questo ***limite non è accettabile***

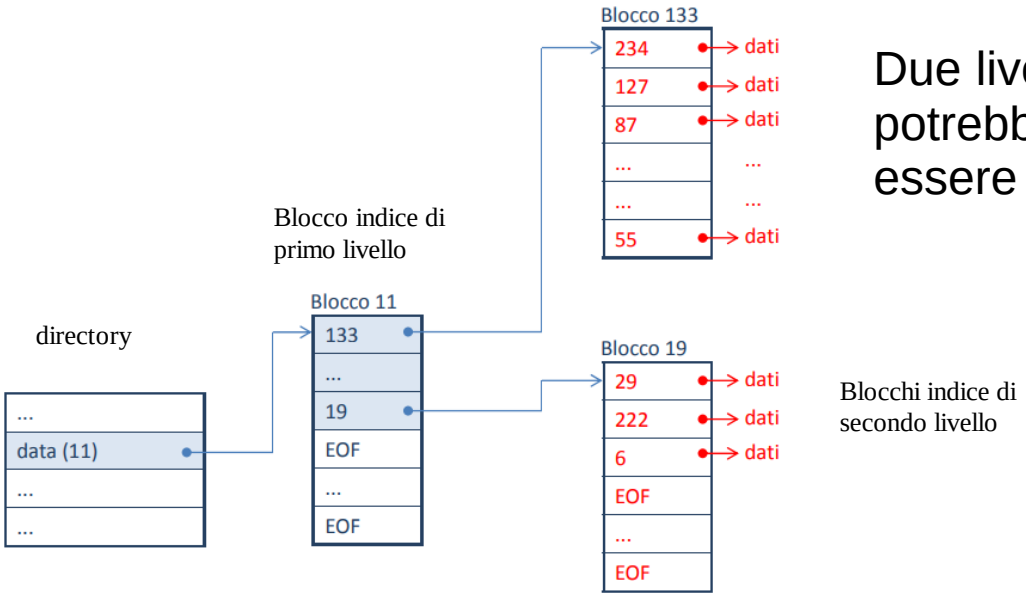
SOLUZIONE: SCHEMI di INDICI a **più livelli**

Diverse implementazioni:

a) il BLOCCO INDICE contiene come ultimo puntatore NIL se il file è piccolo (solo un livello di indice), altrimenti contiene il puntatore ad un altro BLOCCO INDICE (secondo livello di indice), etc.
➔ quindi, per allocare gli indici si usa uno schema a CONCATENAMENTO



b) il primo BLOCCO INDICE contiene i puntatori a BLOCCHI INDICE (secondo livello), etc.



Due livelli di indici potrebbero essere sufficienti

(segue **ALLOCAZIONE CON INDICIZZAZIONE - SVANTAGGI**)

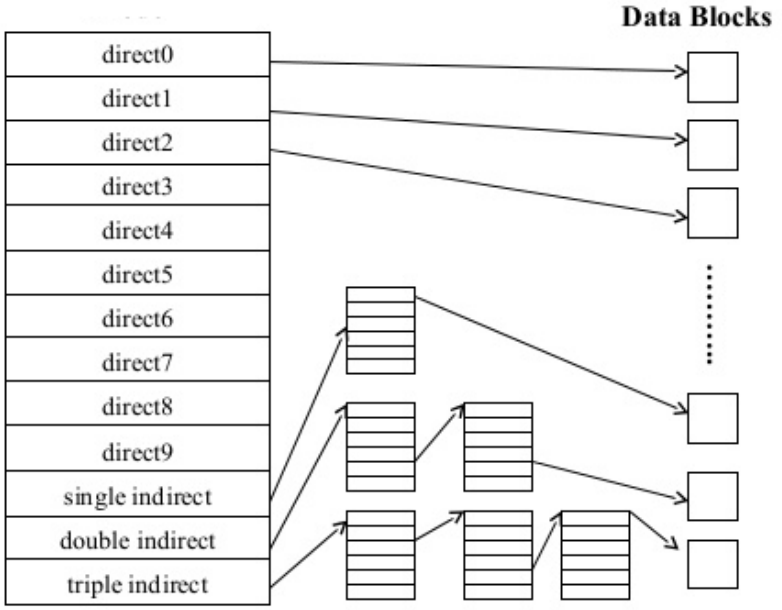
Nel caso si scelga di utilizzare una delle due soluzioni al problema della dimensione massima di un file, tale soluzione *chiaramente* peggiora ulteriormente il discorso relativo allo spreco in caso di file corti (svantaggio 1)

➔ Spazio richiesto per memorizzare i puntatori è eccessivo in caso di FILE PICCOLI

SOLUZIONE:
una alternativa è di 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 (necessari per file lunghi)

ESEMPIO (si riveda slide 32 di Gestione dei file):
Metodo utilizzato in UNIX nello **UFS** (Unix File System)
Il numero di puntatori mantenuti direttamente nell'I-NODE sono in questo caso 13:

- i primi 10 puntano ai primi 10 BLOCCHI del file
- l'11° punta ad un BLOCCO INDICE (di **primo** livello)
- il 12° punta al BLOCCO INDICE della struttura di indicizzazione a **due** livelli
- il 13° punta al BLOCCO INDICE della struttura di indicizzazione a **tre** livelli



(segue **METODI DI ALLOCAZIONE**)

CONCLUSIONI SUI METODI DI ALLOCAZIONE

Il miglior metodo per l'allocazione dei file dipende dal tipo di accesso

* **METODO DI ALLOCAZIONE CONTIGUA** ha ottime prestazioni sia per l'accesso sequenziale che casuale

* **METODO DI ALLOCAZIONE CONCATENATA** si presta in modo molto naturale per l'accesso sequenziale

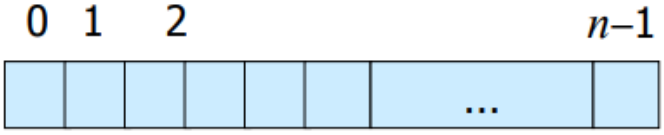
➔ Alcuni S.O. combinano questi due METODI di allocazione in base alla modalità di accesso:
Questo implica che al momento della creazione si specifichi anche se la modalità di accesso è
CASUALE ➔ allocazione contigua
SEQUENZIALE ➔ allocazione concatenata (per eliminare il problema della frammentazione esterna)

* **METODO DI ALLOCAZIONE INDICIZZATA** è 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
  ➔ ci devono essere delle OTTIMIZZAZIONI nel caso di file corti

(segue **METODI DI ALLOCAZIONE**)

GESTIONE BLOCCHI LIBERI

La soluzione più semplice è quella di mantenere, in un **array di bit,** lo **stato** del blocco corrispondente su disco (nel VOLUME CONTROL BLOCK, si veda slide 1): **LIBERO** (bit a 1) o **ALLOCATO** (bit a 0) ➔ **BIT MAP**



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

ESEMPIO: facendo riferimento alla figura della Slide 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1

19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	1	1	1	0	1	1	1	1	1	1

Ricerca dello spazio libero:

- ➔ Scansione del vettore cercando bit con valore 1
- ➔ L'efficienza dipende dalle istruzioni della CPU per l'analisi di vettori di bit
- ➔ Buone prestazioni se il vettore è copiato in memoria centrale

OSSERVAZIONE: Per i dischi attuali ci possono essere difficoltà nel mantenere la bitmap in RAM

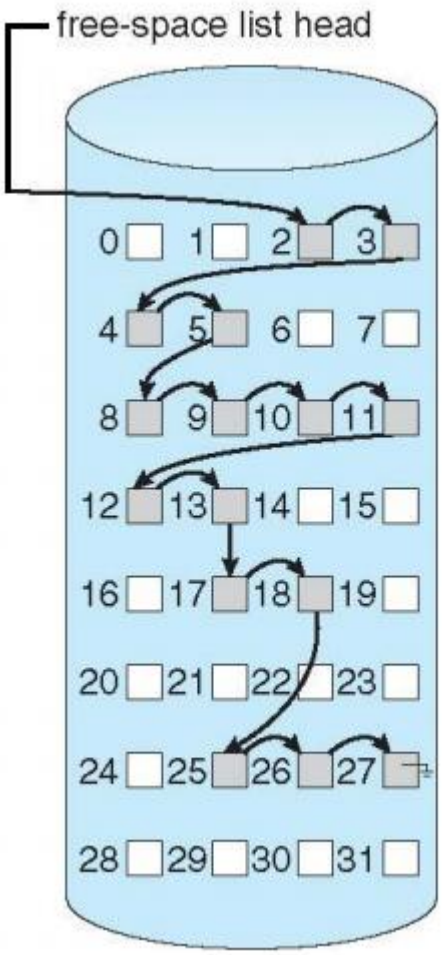
Esempio: dim. blocco = 2^{12} byte (4 KByte)
dim. disco = 2^{40} byte (1 TeraByte)
n = $2^{40}/2^{12} = 2^{28}$ blocchi ➔ 2^{28} bit cioè 32 Mbyte
Si può ridurre la dimensione usando cluster da 4 blocchi ➔ 8 MB di memoria

VANTAGGIO: La Bit Map è adatta per gestire file contigui

(segue **GESTIONE BLOCCHI LIBERI**)

Vediamo che metodi possono essere progettati come alternative alla BIT MAP

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 indicherà in modo opportuno il termine della lista) e si mantiene un puntatore alla testa della lista in una zona riservata del disco (sempre nel VOLUME CONTROL BLOCK, slide 1)➔ solo questo puntatore alla testa viene copiato in memoria centrale ➔ non si spreca spazio

OSSERVAZIONI:

– Quando si cerca un solo blocco libero, la lista così realizzata è efficiente (si stacca il primo blocco libero e si riconcatena il puntatore alla testa col secondo)

– Quando si cercano **n** blocchi liberi consecutivi, si rischia di dover scorrere tutta la lista, con tempi di attesa piuttosto lunghi ➔ Problemi di efficienza

NOTA BENE: Nella **FAT**, l'informazione sui blocchi liberi è inclusa nella struttura dati per l'allocazione (blocchi contrassegnati con 0) e **non** richiede quindi un metodo di gestione separato!

(segue **GESTIONE BLOCCHI LIBERI**)

Per migliorare l'efficienza del metodo precedente, si può ricorrere al:

2) **Grouping**:
   il primo blocco libero contiene gli indirizzi di altri **n-1** blocchi liberi più un puntatore al successivo (che ha anch'esso la stessa struttura)
   ➔ la lista si trasforma 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 è seguito)
   ➔ la lista si accorcia drasticamente
   ➔ utile quando si usa l'allocazione contigua o gli extent

(segue **GESTIONE DEI FILE**)

OSSERVAZIONI GENERALI

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

Ad esempio:

- In UNIX, gli I-node sono preallocati e distribuiti nel disco, per mantenere dati e metadati vicini e diminuire il tempo di seek[3]
- Se, nel descrittore di un file, si mantiene la data di ultimo accesso al file per consentire all'utente di risalire all'ultima volta che un file è stato letto, ogni volta che si apre un file per la lettura, si deve non solo leggere il descrittore del file, ma anche scriverlo!

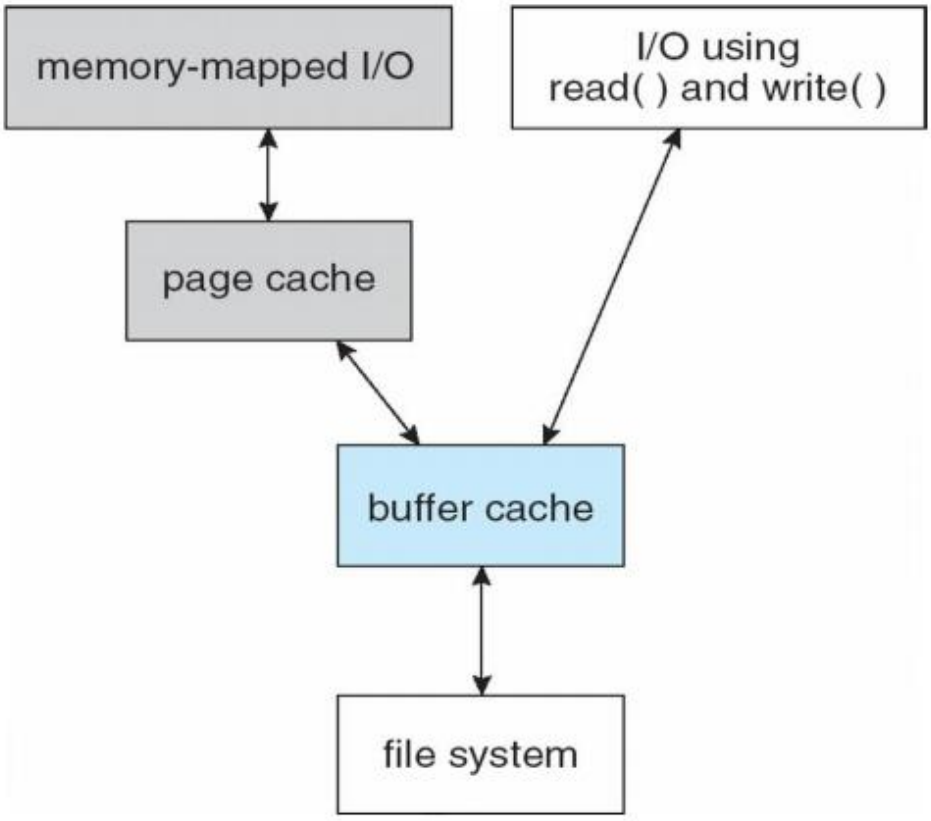
Le **PRESTAZIONI** dipendono da:

- Mantenere dati e metadati "vicini" nel disco
- Disporre di **buffer cache** (o **disk cache**), cioè sezioni dedicate della memoria *centrale* in cui si conservano i blocchi usati di frequente
  - ➔ se devono essere effettuate scritture *sincrone* (talvolta richieste dalle applicazioni o necessarie al sistema operativo) è impossibile usare il buffering/caching dato che l'operazione di scrittura su disco deve essere completata prima di proseguire l'esecuzione
  - ➔ per fortuna però le scritture *asincrone*, che sono le più comuni, sono invece bufferizzabili

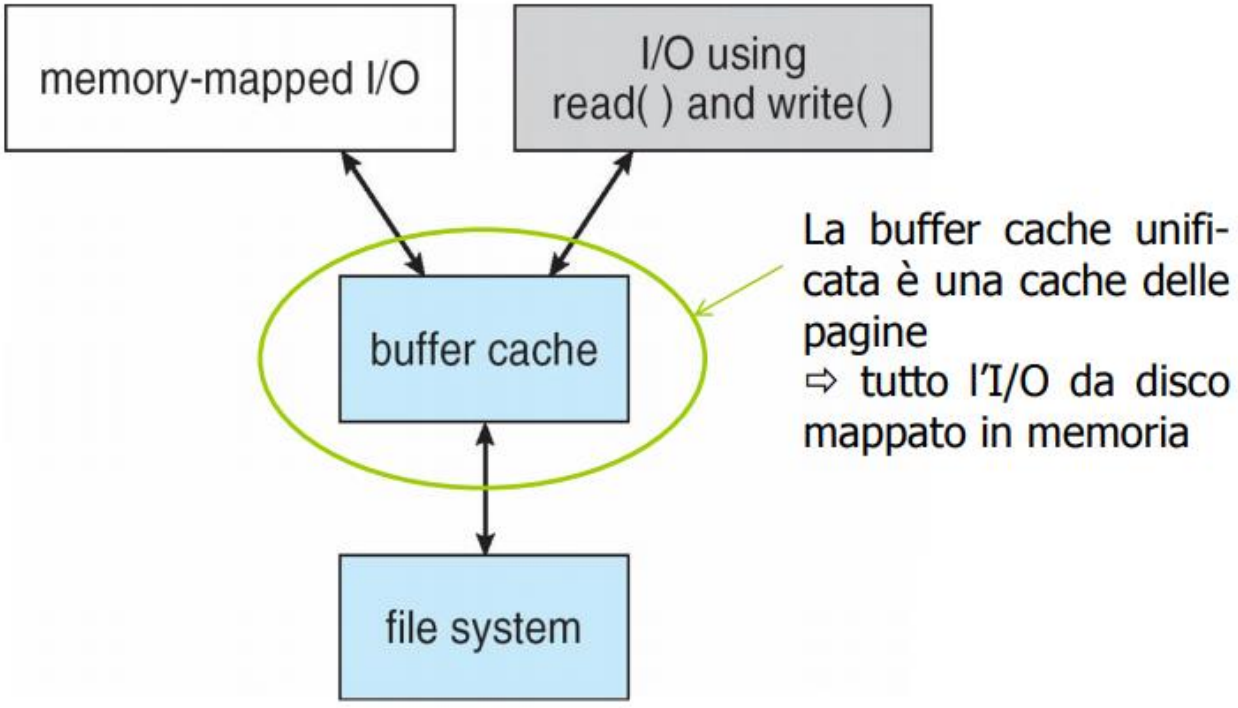
[3] ***Seek time***: è il tempo necessario a spostare la testina sulla traccia che contiene il blocco che deve essere acceduto

(segue **PRESTAZIONI**)

NOTA:
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, evitando il double caching
➔ usato ad esempio da Solaris e da alcune versione di UNIX



(segue **PRESTAZIONI**)

L'algoritmo **LRU** è, in generale, ragionevole per la sostituzione delle pagine e dei blocchi nella cache (unificata o meno)

Tuttavia ci sono casi in cui **non** risulta adatto

➔ Le pagine relative ad un file da leggere o scrivere in modo **sequenziale** *non* si dovrebbero sostituire nell'ordine *LRU*, dato che la pagina usata più di recente non verrà probabilmente più utilizzata

Nel caso di file acceduti in modo sequenziale è più conveniente adottare altre tecniche:

- Con il **rilascio indietro o free–behind** che rimuove una pagina dalla cache non appena si verifica una richiesta della pagina successiva
- Con la **lettura anticipata o read–ahead** si leggono e si copiano nella cache la pagina richiesta e diverse pagine successive, che verranno probabilmente accedute in sequenza

Si possono migliorare le prestazioni relativamente all'accesso ai file dedicando sezioni della memoria centrale come dischi virtuali ➔ **DISCHI RAM**

➔ la virtualizzare di un disco in memoria è effettuata in modo trasparente all'utente e:

– Usa il disco RAM in fase di lettura/scrittura del file

– Usa *obbligatoriamente* una implicita fase periodica di riscrittura del file su supporti permanenti

(segue **GESTIONE DEI FILE**)

RIPRISTINO (RECUPERO)

Poiché i file e le directory sono mantenuti sia in memoria RAM (*parzialmente*) sia nei dischi, è necessario 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 molto acceduti), si rischia l'inconsistenza in seguito ai crash ➔ si devono quindi dividere i blocchi in due categorie:

- se il blocco corrisponde a dati e verrà riusato a breve, viene mantenuto nella cache
- se il blocco è **critico** per la consistenza del file system (cioè tutti i blocchi tranne quelli di dati), allora ogni modifica deve essere immediatamente trasferita al disco

Da notare che le modifiche ai blocchi dati devono comunque 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, MS-DOS)

Ci possono essere altri problemi se il crash si verifica in situazioni di modifica di metadati

Esempio: all'atto della creazione di un file:

- Creazione del descrittore di file e inserimento nella directory
- Allocazione blocchi di dati
- Aggiornamento delle informazioni (puntatori) blocchi liberi e descrittori liberi

Se si ha un crash si possono avere incoerenze fra le strutture ➔ Il contatore dei descrittori liberi potrebbe indicare che il descrittore sia stato già stato considerato occupato, ma la directory non contenere *ancora* un puntatore all'elemento relativo!

(segue **RIPRISTINO**)

Esistono due approcci al problema della *consistenza del file system*:

1) risolvere le inconsistenze *dopo* che si sono verificate, con programmi di controllo della consistenza ➔ si deve utilizzare un **VERIFICATORE DI COERENZA**
   ➔ **fsck** in UNIX, **chkdsk** in DOS/Windows
   ➔ confronta i dati nella struttura di directory con i blocchi di dati sul disco e tenta di "fissare" le eventuali incoerenze ➔ lenti, e non sempre funzionano
2) *prevenire* le inconsistenze: i **journalled file system**!

Poiché i dispositivi di memoria di massa hanno un MTBF[4] relativamente breve, i sistemisti devono valutare soluzioni per aumentare l'affidabilità dei dati (che possono anche essere combinate assieme):

1) Aumentare l'affidabilità dei dispositivi (es. **RAID**)
2) **Backup** (automatico o manuale) dei dati dal disco ad altro supporto (altri dischi magnetici, supporti ottici, etc.):
   - dump fisico: direttamente i blocchi del file system
   ➔ veloce, ma difficilmente incrementale e non selettivo
   - 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

Terminologia: **Backup completo**: si salvano tutti i file e directory richiesti
Backup differenziale: Per crearlo, si deve prima creare un backup completo; il backup differenziale memorizzerà *poi* solo le modifiche che sono state apportate al file system dall'ultimo backup completo ➔ per il recupero, è necessario solo il backup completo e l'ultimo backup differenziale
Backup incrementale: Per crearlo, si deve prima creare un backup completo; poi sono memorizzate solo le ultime modifiche rispetto all'ultimo backup ➔ per il recupero, è necessario il backup completo e tutti i backup incrementali

[4] **MTBF** = Mean Time Between Failures ➔ tempo medio fra i guasti

(segue **GESTIONE DEI FILE**)

SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE

OPERAZIONE CREATE

routine CREATE (nome-file, attributi) **(*)**
begin

RICERCA nella directory del nome-file;
if TROVATO then
- avviso di duplicazione
  ➔ errore; return;
- creazione di una nuova versione
- sovrascrittura ⇐ utilizzato in UNIX/Linux

RICERCA di un descrittore libero
if NOT TROVATO then
- ➔ errore; return; ⇐ utilizzato in UNIX
- allocazione di un nuovo descrittore
  if NOT SPAZIO then
  ➔ errore; return;

REGISTRAZIONE nella directory del descrittore

ALLOCAZIONE spazio per il file
- tutto ⇐ nel caso di allocazione contigua
- parte ⇐ il blocco indice nel caso di allocazione ad indicizzazione
- niente ⇐ nel caso di allocazione concatenata

REGISTRAZIONE nel descrittore dei blocchi allocati
REGISTRAZIONE nel descrittore degli attributi

[invocazione della routine OPEN in scrittura]
end;

(*) *Nota Bene: uso di una notazione Pascal-like*

(segue **SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE**)

OPERAZIONE OPEN

routine OPEN (nome-file, modo-accesso): connection-ID
begin

RICERCA nella directory del nome-file per trovare il descrittore del file;
if NOT TROVATO then
    indicazione di errore; return;

VERIFICA del modo di accesso richiesto rispetto ai diritti sul file del processo/utente
if NOT AUTORIZZATO then
    indicazione di errore; return;

INSERIMENTO della copia del descrittore nella tabella dei file attivi di sistema (se non già presente; se presente aggiornamento del contatore di uso) e inserimento di informazioni nella tabella dei file aperti (TFA) del processo
        if NOT SPAZIO then
            ➔ errore; return;

DEFINIZIONE del connection-ID[5] per il file ⇐ in UNIX/Linux indice TFA

INIZIALIZZAZIONE del puntatore all'interno del file (di default a 0)

return connection-ID

end;

ACCESSI CONCORRENTI allo stesso file

➔ aperture separate da parte di processi diversi implicano la condivisione dell'elemento nella tabella dei file attivi di sistema!

Ogni processo ha un suo connection-ID e quindi il proprio file pointer ➔ Nota bene: in UNIX, la presenza della ulteriore Tabella dei file aperti di sistema consente a processi in parentela di condividere il file pointer!

[5] File Descriptor (fd) in UNIX e File Handle (in MS-DOS)

(segue **SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE**)

OPERAZIONE READ

routine READ (connection-ID, num-byte, in-buffer): stato
begin

VERIFICA se il connection-ID è valido
if NOT VALIDO then
indicazione di errore; return stato-errato;

VERIFICA se il file è aperto in lettura
if NOT VALIDO then
indicazione di errore; return stato-errato;

CALCOLO dell'indirizzo fisico
INVIO comandi di lettura al controllore del dispositivo

VERIFICA se il numero di byte rientra nelle dimensioni del file
if NOT VALIDO then
- indicazione di errore; return stato-errato;
- return stato che indica il numero di byte letti ⇐ utilizzato in UNIX/Linux

stato=risultato lettura

COPIA di num-byte dati dai buffer interni a in-buffer

AGGIORNAMENTO del puntatore all'interno del file

return stato

end;

Per estrarre i dati da un file si fa uso di **BUFFER INTERNI** che appartengono al S.O. ➔ *disk cache* eventualmente unificata con la page cache

VANTAGGIO:
possibilità di ridurre il numero di operazioni di I/O su disco prelevando le informazioni dai blocchi già presenti in memoria centrale

(segue **SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE**)

OPERAZIONE WRITE

routine WRITE (connection-ID, num-byte, out-buffer): stato
begin
VERIFICA se il connection-ID è valido
if NOT VALIDO then
indicazione di errore; return stato-errato;
VERIFICA se il file è aperto in scrittura
if NOT VALIDO then
indicazione di errore; return stato-errato;
ESTENDE il file se necessario allocando un nuovo blocco/cluster e AGGIORNANDO le informazioni presenti nella tabella dei file attivi di sistema
if NOT SPAZIO then
➔ errore; return;
CALCOLO dell'indirizzo fisico
COPIA di num-byte dati da out-buffer ai buffer interni
INVIO comandi di scrittura al controllore del dispositivo se scrittura sincrona, altrimenti se scrittura asincrona ➔ vedi dopo!
stato=risultato scrittura
AGGIORNAMENTO del puntatore all'interno del file
return stato
end;

Anche per la scrittura si utilizzano i **BUFFER INTERNI**

VANTAGGIO:
possibilità di SCRITTURA RITARDATA per ridurre il numero di operazioni di I/O su disco

Le scritture vengono effettuate solo sul buffer
===> il buffer non viene copiato sul disco, ma solo ***MARCATO*** come **SCRITTO**

Quando servirà un buffer e non ce ne sono di liberi si seleziona una VITTIMA ➔ strategie analoghe al rimpiazzamento di pagina nella gestione memoria virtuale
➔ se il buffer selezionato è MARCATO, lo si deve scrivere sul disco
➔ ci può essere una scrittura periodica dei buffer MARCATI, in seguito alla quale viene tolta la marcatura

(segue **SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE**)

OPERAZIONE SEEK

routine SEEK (connection-ID, posizione-logica)
begin

VERIFICA se il connection-ID è valido
if NOT VALIDO then
    indicazione di errore; return;

CALCOLO della posizione richiesta
if NOT VALIDO then
- indicazione di errore; return;
- nessuna indicazione di errore ⇐ utilizzato in UNIX/Linux

AGGIORNAMENTO del puntatore all'interno del file

end;

NOTA BENE: in UNIX/Linux, la primitiva di lseek() prevede 3 parametri dato che il parametro indicato come "posizione-logica" viene espresso da offset e origine dello spostamento (inizio file, posizione corrente del file pointer e fine del file)

Se un **file** è stato creato di tipo **sequenziale** (o è memorizzato su un dispositivo sequenziale o è un dispositivo sequenziale, come la tastiera o il video) l'operazione di SEEK

- non ha effetto

oppure

- produce una indicazione di errore

(segue **SCHEMI FUNZIONALI DELLE OPERAZIONI SUI FILE**)

OPERAZIONE CLOSE

routine CLOSE (connection-ID)
begin
VERIFICA se il connection-ID è valido
if NOT VALIDO then
indicazione di errore; return stato-errato;
COPIA il descrittore del file sul disco nel caso sia stato modificato
ELIMINA dalla TFA del processo le informazioni corrispondenti e AGGIORNA il contatore d'uso della tabella dei file attivi di sistema (e nel caso ELIMINA le informazioni)
ANNULLA la definizione di connection-ID
end;

In certi sistemi, la **CHIUSURA** dei file avviene in modo **automatico** alla terminazione dei processi (**UNIX**) ➔ chiaramente alla terminazione di un processo la TFA viene deallocata e quindi automaticamente le informazioni sono eliminate, ma devono essere aggiornati i contatori d'uso delle due tabelle di sistema ed eventualmente eliminate le informazioni non più necessarie!

OPERAZIONE DELETE

routine DELETE (nome-file)
begin
RICERCA nella directory del nome-file;
if NOT TROVATO then errore; return;
VERIFICA del diritti sul file del processo/utente
if NOT AUTORIZZATO then
indicazione di errore; return;
DEALLOCAZIONE spazio per il file
CANCELLAZIONE del descrittore del file ⇐ in UNIX, I-NODE torna libero
end;

Invece che la cancellazione del descrittore, ci può essere un meccanismo che invalida il descrittore in modo da rendere semplice il recupero del file in caso di cancellazione accidentale ➔ ad esempio, usato da MS-DOS

(segue **GESTIONE DEI FILE**)

GENERALIZZAZIONE DEL CONCETTO DI FILE

Il concetto di FILE può essere usato anche come ASTRAZIONE del SISTEMA di INGRESSO/USCITA

Questo vuol dire che tutte le periferiche (ad esempio, 💻 e ⌨) sono trattate come file

L'utente può usare un UNICO ed UNIFORME insieme di servizi per trattare i file e l'I/O
➔ **I/O INDIPENDENTE DALLE PERIFERICHE**

Sistema di I/O ***indipendente*** dai dispositivi ➔ FORMA di collegamento RITARDATO tra i programmi ed i dispositivi

I programmi usano una **ASTRAZIONE** dei dispositivi
➔ standard input e standard output

Il collegamento fra l'**ASTRAZIONE** e il dispositivo avviene a tempo di esecuzione usando la ***RIDIREZIONE*** (UNIX/Linux e MS-DOS) mediante:

- *il linguaggio comandi con appositi metacaratteri*
  ESEMPIO: in UNIX/Linux,
  < ridirezione dello standard input
  > ridirezione dello standard output
- *chiamate di sistema*
  ESEMPIO: in UNIX/Linux,
  close(0) e **open** del file da cui si vuole leggere come standard input
  close(1) e **creat** del file su cui si vuole scrivere come standard output

Chiaramente esiste una **associazione di default**:
standard input ➔ tastiera e standard output ➔ video

(segue GENERALIZZAZIONE DEL CONCETTO DI FILE**)**

In UNIX, per fornire questa ASTRAZIONE UNICA le periferiche stesse sono viste come **FILE**

➔ File della sottodirectory **/dev** in UNIX

FILE organizzati a blocchi ➔ dischi
FILE organizzati a caratteri ➔ terminali e stampanti

Su tutti questi **dispositivi/file** si utilizzano le OPERAZIONI di OPEN, CLOSE, READ e WRITE

L'OPERAZIONE di SEEK ha senso solo per i file/dispositivi organizzati a blocchi

I dispositivi sono FILE che vengono creati alla ***installazione*** del sistema

ESEMPIO: su LICA02

`soELab@Lica02:~$ ls -l /dev` (la lettera evidenziata a inizio di ogni riga significa: c ➔ dispositivo a carattere, b ➔ dispositivo a blocchi)

```
…
crw------- 1 root root      5,   1 Nov 17 12:32 console
…
crw-rw-rw- 1 root root      1,   3 Nov 17 12:32 null
…
crw-rw-rw- 1 root tty       5,   2 Dec  3 19:06 ptmx
…
brw-rw---- 1 root disk      8,   1 Nov 17 12:32 sda1
brw-rw---- 1 root disk      8,  17 Nov 17 12:32 sdb1
…
lrwxrwxrwx 1 root root         15 Nov 17 12:32 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root         15 Nov 17 12:32 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root         15 Nov 17 12:32 stdout -> /proc/self/fd/1
…
crw-rw-rw- 1 root tty       5,   0 Nov 29 15:20 tty
…
```

NOTE:

1) /dev/null usato per "scartare" output non interessante;
2) /dev/ptmx usato per i terminali virtuali;
3) /dev/sda1 e /dev/sdb1 dischi montati (si veda slide 37 sulla Gestione dei file)
4) /dev/tty usato per "forzare" l'ouput sul terminale corrente

(segue **GENERALIZZAZIONE DEI SERVIZI PER FILE**)

In alcuni S.O. questa **uniformità** di trattamento viene estesa anche ad altri concetti

Infatti, in alcuni sistemi (ad esempio **UNIX/Linux**) viene fornito un meccanismo di ***PIPE*** per fare comunicare due processi ➔ 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

➔ Possibilità di connettere l'uscita di un comando/programma con l'ingresso di un altro

QUINDI,
senza alcuna necessità di modifiche,
un programma può **leggere** da
- una tastiera (dispositivo)
- un file
- un altro comando

e analogamente,
un programma può **scrivere** su
- un video o una stampante (dispositivi)
- un file
- un altro comando

ESEMPIO: UNIX `$ls | more`

La shell (`$`) (processo P0) crea una sotto-shell (processo P1) con la primitiva *fork()* e poi attende la sua terminazione con la primitiva *wait()*:
- P1 crea la pipe p con la primitiva *pipe()*;
- P1 crea una sotto-shell (processo P2) con la primitiva *fork()*, quindi …

P1	**P2**
close(0);	close(1);
dup(p[0]);	dup(p[1]);
close(p[0]); close(p[1]);	close(p[0]); close(p[1]);
execlp("more", …);	execlp("ls", …);

NETWORK FILE SYSTEM

Il Network File System (NFS)[6] è un buon esempio di **File System di rete** basato su protocolli client-server

L'NFS è stato definito e implementato dalla SUN sulla base dei **protocolli TCP/IP** (o UDP/IP a seconda della rete di comunicazione)
➔ definito per la prima volta per il **S.O. Solaris**

L'NFS risulta disponibile nella maggior parte delle distribuzioni di UNIX e Linux e in alcuni S.O. per PC

Nel contesto dell'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 è 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 è basata su una **relazione client-server**
➔ una stazione può essere sia un client che un server

Affinchè una directory remota (*DR*) sia accessibile in modo trasparente a una certa stazione cliente, questa deve prima effettuare una operazione di **MONTAGGIO** di DR nel file system (logico) locale della stazione cliente
➔ la directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito

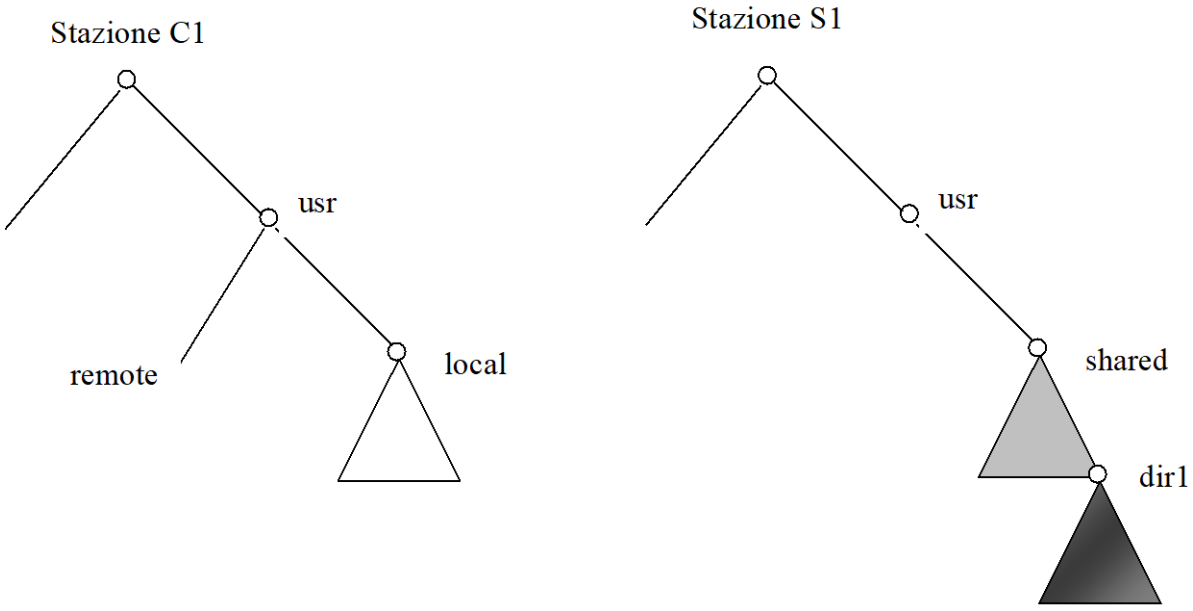
Dopo tale operazione, la directory remota risulta accessibile in **modo trasparente** tramite il file system locale

[6] Attenzione a non confondere l'NFS con NTFS (New Technology File System) di Windows

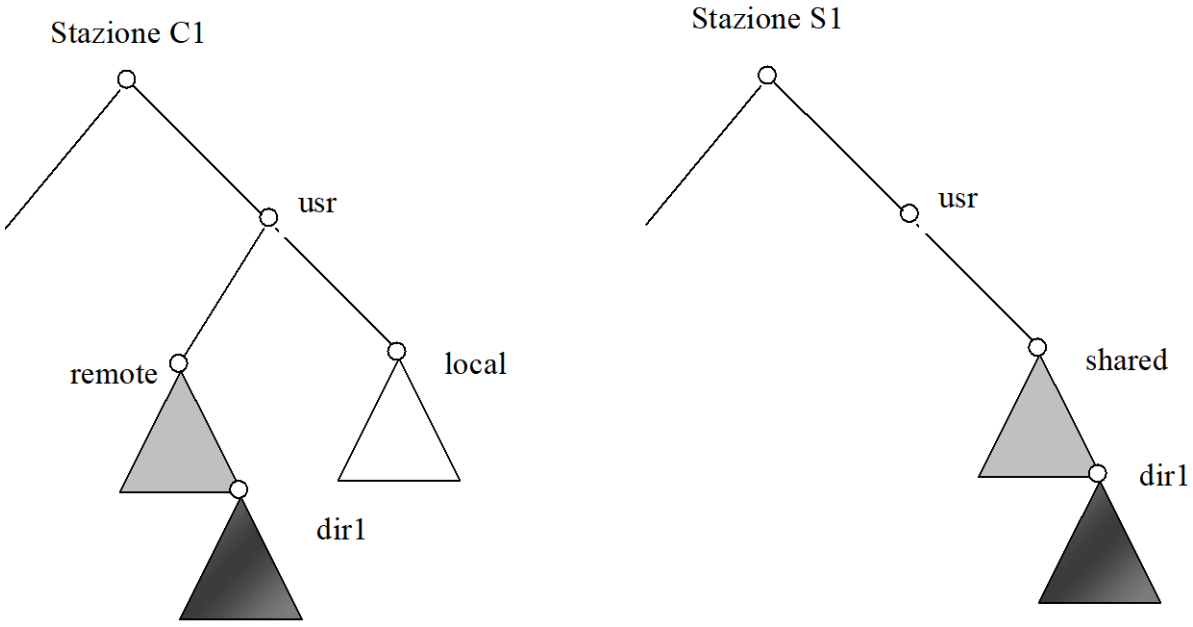
(segue **NFS**)

Supponiamo ad esempio di avere due stazioni, C1 e S1, C1 che si comporta come client e S1 che si comporta come server

Inizialmente, C1 e S1 possono accedere solo ai file system locali



Si consideri quindi di effettuare il montaggio di S1:/usr/shared in C1:/usr/remote



(segue **NFS**)

Gli utenti di C1, possono quindi accedere a qualsiasi file ad esempio di dir1 usando il percorso /usr/remote/dir1

NB: se remote avesse contenuto qualcosa, dopo il montaggio, questo contenuto non sarebbe più accessibile, fino a che non si effettua lo smontaggio ➔ stesso comportamento del montaggio "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

OSSERVAZIONE.
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**

REALIZZAZIONE

Uno degli scopi nella progettazione dell'NFS era quello di operare in un ambiente eterogeneo di stazioni, sistemi operativi e architetture di rete

Questa indipendenza si ottiene usando le primitive RPC costruite su uno specifico protocollo di rappresentazione dei dati detto **XDR** (**eXternal Data Representation**)

Per chi fosse interessato a conoscere ulteriori dettagli su NFS può consultare le slide AppendiceNFS.pdf [Operating System Concepts – 9th Edition (dal cap. 12), Silberschatz, Galvin and Gagne ©2013]