**FILE SYSTEM**

Ogni app deve poter salvare e recuperare a lungo termine le informazioni.

Requisiti:

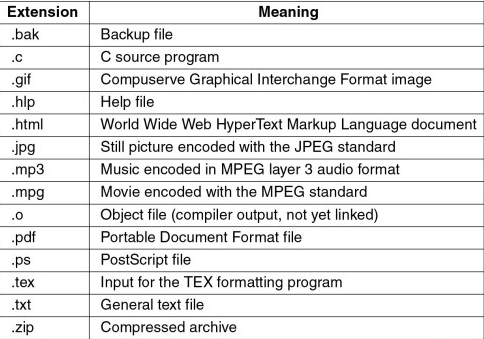
* Grande capacità di memorizzazione
* Sopravvivenza delle informazioni dopo la terminazione del processo
* Accesso in contemporanea da parte di più processi. CANONICI

La memoria principale non rispetta i requisiti->memoria secondaria (es: dischi).

Come salvo le informazioni? Dove? Come gestisco i permessi? Come so quale blocco è libero?

Uso un **file**: un dispositivo ne può contenere tanti, leggibili, duplicabili e creabili, persistenti...

Per studiare un file system ci sono 2 punti di vista:

* **Utente:** servizio per salvataggio e recupero informazioni salvate. Studio struttura, nomina, protezione e operazioni di un file, ignoro i dettagli a basso livello.
* **System designer:** insieme di strutture date e meccanismi per gestire in modo efficiente un dispositivo di memorizzazione per salvare o recuperare informazioni. Studio larghezza dei blocchi logici, implementazione e mappatura degli spazi liberi.

P.O.V.: UTENTE

**FILE:** unità di memoria logica di informazioni.

* Nome: una certa lunghezza, numeri e caratteri speciali ammessi (non tutti), case-sensitive, case-persevering (lascia maiuscolo o minuscolo come scritto, non mette tutto in un modo), supporto o no del ".", estensione (non sempre obbligatoria, a volte più di una).
* File structure: struttura del file, può essere :

sequenza di byte :non strutturata, il SO non conosce il contenuto, flessibilità max, nessun supporto del SO,

sequenza di record: poco usato, unità di misura = record di lunghezza fissa)

Albero di record: ogni record ha un campo chiave, l'albero è ordinato in base alla chaive.

* Tipo di file:

File normali: di testo (sequenze di caratteri stampabili) o binari (non ascii, non leggibili)

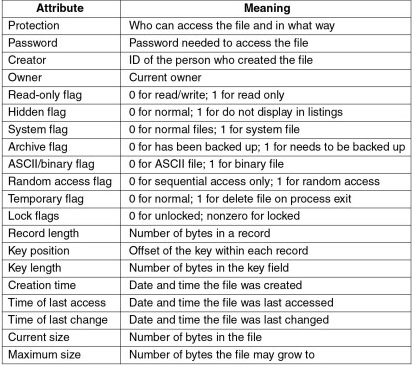
Cartelle: file di sistema per mantenere la struttura di un file system

Devices files: a caratteri speciali o a blocchi speciali.

Alcuni file sono tipati fortemente: il tipo viene riconosciuto automaticamente in un range molto ampio, user-friendly ma poco flessibile

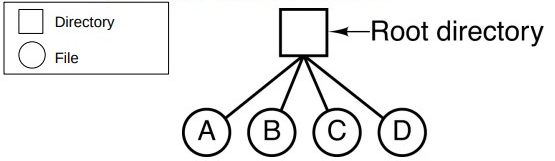
* Accesso:

Sequenziale: bytes e records letti/scritti in ordine inizio-fine, non posso skipparli e per leggere una posizione devo aver letto le precedenti

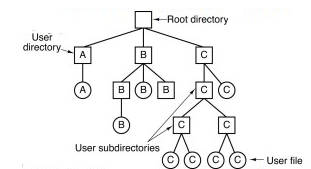
Casuale/Diretto: bytes e records letti/scritti in qualsiasi ordine senza avere accesso alle posizioni precedenti. Usano il seek (incrementa/decrementa di poco il puntatore alla posiz corrente) o un indirizzo di posizione.

* Attributi (metadati): proprietà associate ad ogni file: data e ora di creazione, ultima modifica, permessi...
* Operazioni: set di comandi e funzioni per interagire con i files

Create (aggiunge anche attributi default), delete (libera spazio), open (fetch info da disco a main mem, +1 processi aperti), close (se sta ancora scrivendo salva le info e distrugge strutture dati della main mem, -1 processi aperti), read, write (può aumentare la size, sovrascrivere...), rename, seek, append, getAttributes, setAttributes

**DIRECTORY**: modi per organizzare collezioni di file nel file system, ogni file system contiene una root directory.

Possono avere diverse organizzazioni:

* Livello singolo: la root contiene tutto, facile da implementare ma limitata, non organizzabile e non va bene per i sistemi multi-utente.
* Gerarchia di cartelle: strutturata come un albero con la root come radice, le cartelle possono avere sottocartelle, possono esserci cartelle private nei sistemi multi-utente.

Nome del percorso: modo per specificare il nome di un file in una cartella albero. Sequenza di nomi separata da un path separator “/”. "." = posizione corrente, ".." = parent della posizione corrente.

Nome del percorso assoluto: percorso dalla radice al file, inizia sempre con la radice.

Nome del percorso canonico: semplice e diretto (senza . e ..)

Nome del percorso relativo: relativo alla posizione corrente.

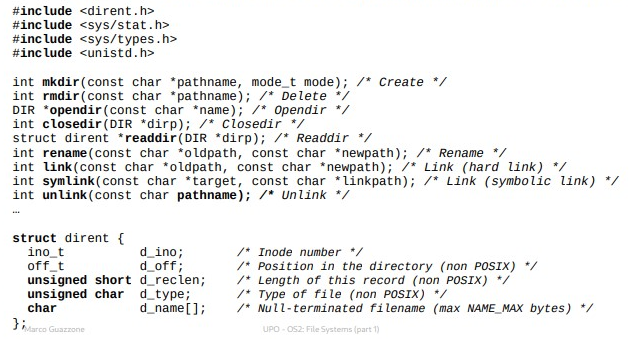
Operazioni: creazione, cancellazione, readdir (leggi la prossima entry in una cartella aperta), rename,...

Crea collegamento:

Lynk: crea un Hard link: crea collegamento che punta alla stessa struttura del file.

Symlink: crea un Soft link: il link è separato in un altro file di tipo link la cui struttura contiene il percorso del file (link simbolico).

Unlink: distrugge un link rimuovendo la entry associata. se hard decrementa il contatore dei link.

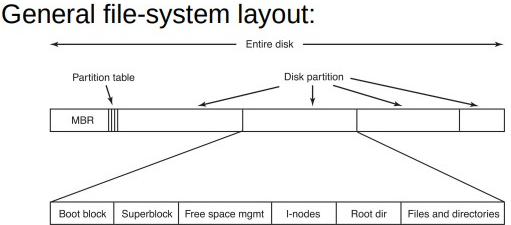


P.O.V.: DESIGNER

**FILE**= collezione di blocchi contigui (blocco: gruppo di settori di disco consecutivi). Per ogni file c'è un file-control block che permette di trovare i blocchi associati al file.

**Cartella**: file di directory entries ognuna mappa il nome di un file nella file-control block.

Salvataggio e ritrovamento: i file systems sono salvati in un disco e quelli in una partizione possono essere diversi da quelli nelle altre partizioni.



**Boot block** (boot sector): contiene il boot loader del SO istallato nella sua partizione. La partizione contiene sempre un boot block anche senza un SO, in quel caso è vuota.

**Superblock** (volume control block): blocco speciale con i parametri chiave di un file system (File system type ID, num di blocchi, dim blocchi...) viene portato in memoria la prima volta che uso il file system.

**Free space management**: informazioni riguardo i blocchi liberi. Ha una struttura che tiene traccia dei blocchi liberi.

**I-nodes** (file control blocks): informazioni riguardo ogni file. Struttura dati che tiene traccia dei metadati e blocchi associati ad ogni file.

**Root dir**: cartella radice, top del file system tree. **File** e **Cartelle.**

**Montaggio del file system**: il file system è salvato su disco ma le strutture dati sono caricate e create in main memory quando sono usate. Utile per file-system management e migliorare performance.

**Smontaggio del file system**: le strutture dati in memoria sono aggiornate durante le operazioni e anche distrutte al tempo di smontaggio. Tutti i cambiamenti devono essere scritti su disco.

Strutture dati in memoria (dipendenti dal SO):

* **tabella di montaggio**: contiene informazioni riguardo ogni file system montato
* **cache delle cartelle**: contiene informazioni sulle cartelle recentemente accedute.
* **Tabella dei file aperti a livello sistema**: contiene una copia dei file control block di ogni file aperto e le loro informazioni. Ogni entry ha un contatore usato dei processi aperti.
* **Tabella file aperti per processo**: contiene file aperti da quel processo.
* **cache dei blocchi**: tiene i blocchi recentemente usati.

Creazione di un file: crea un nuovo file control block, aggiorna la struttura delle cartelle che lo contengono aggiungendo le entry, se apre il file svolge altre operazioni.

Apertura di un file: cerca la struttura delle cartelle per il nome del file dato, quando la trova lo cerca la entry del file nella tabella dei file aperti a liv sistema.

Entry trovata: file già aperto

1. nuova entry nei file aperti x programma
2. setta il ptr a una entry della tabella dei file aperti a liv sistema in una entry della tabella dei file aperti per processo
3. incrementa il count dei file aperti +1.

Entry non trovata: file non aperto

1. nuova entry tab programmi aperti liv sistema leggendo il relativo file control block su disco
2. Setta a 1 il counter nella entry della tab file aperti liv sistema
3. Crea una entry nella tabella file aperti per processo
4. Setta ptr della entry della tab dei file aperti a liv sistema in una entry della tabella dei file aperti per processo.

Return: ptr entry tab. per processo.

Chiusura di un file: rimuovo entry da tab per processo, counter--, if (counter==0): rimuovo entry della tabella

Per le altre operazioni: uso il ptr alla entry della tab per proccesso.

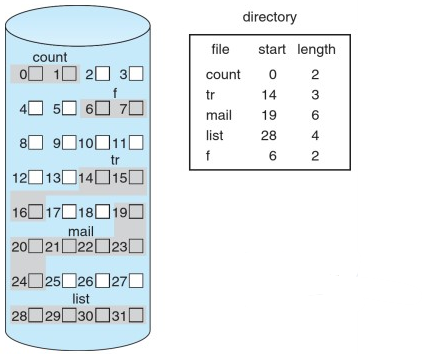
Implementazione file: come alloco blocchi ad un file? Come tengo traccia di quali blocchi sono su quale file? Diversi schemi di allocazione file.

ES: F = dim file in bytes; B = dim blocco in bytes; <alloc-data> = numero di bytes allocati per salvare dati; <mgmt> = numero bytes usati x file allocation management; <alloc-mgmt> = bytes allocati per mgmt; <allocated> = bytes allocati per un file.

Blocchi allocati: N>=F/B

Overhead x gestione spazio = (<alloc-mgmt>/<allocated>)%

Wasted = (1-(<mgmt>+F)/<allocated>)%

**Schema: Allocazione contigua**: salva i files come sequenza contigua di disk blocks.

Considero 20KiB file: 1Kib disk block->File di 50 blocchi; 2Kib disk block->File di 25 blocchi.

Entry delle directory contengono: indirizzo disco del 1° blocco e numero di blocchi.

Ogni file inizia al blocco seguente la fine del precedente

+)Semplice +)supporta accesso sequenziale e diretto +)Seek minimo

-)Frammentazione esterna: se cancello un file rimane un buco non facile da riempire ->Deframmentazione (compatto i dati) costosa.

-)Devo conoscere quanto spazio serve per ogni file: difficile da sapere.

{F' = dim stimata; F = dim attuale B =dim blocco -> num blocchi N = F'B Spreco = (1--F/(n\*B))%

ES: F' = 8Kib; F = 4Kib; B =1Kib ->N=8/1=8 blocchi. Overhead=0, Spreco=50%}

**Schema: Allocazione estendibile**:ogni file consiste in una o più regioni non contigue di blocchi contigui (extents).

Creazione file: alloco sequenza di blocchi liberi.

File cresce = aggiungo dello spazio contiguo.

Gli extents hanno lunghezza variabile o fissa.

Ogni file ha una tabella degli extent per tener traccia degli extent allocati.

Entry x lunghezza var: <indirizzo 1° blocco, numero blocchi>

Entry x lunghezza fissa: <indirizzo 1° blocco>

+)Non devo conoscere la dim in anticipo

+)No frammentazione esterna per lunghezza fissa (riuso un buco estendendolo).

-)Costoso

-)Lunghezza fissa: frammentazione interna; Lunghezza variabile: frammentazione esterna.

{F: dim file; B: dim blocco, P:dim puntatore blocco; E: blocchi x extent.

extent allocati M:F/(E\*B); blocchi x salvare la tab extent T:M\*P/B;

blocchi allocati N:M\*E +T; overhead: ((T\*b )/ (N\*B))%=(T/N)%

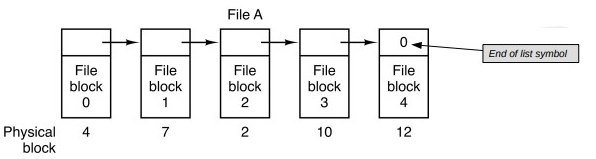
ES: F=4Kib, B=1Kib, P=4B, E=4 blocchi

M=4/4\*1=1 extent allocato. T=1\*4B/1Kib=1. N=1\*4+1=5 Overhead=1/5=20% Spreco=19.92%}

**Schema: Allocazione in linked-list**

I blocchi di file sono organizzati come una linked-list, ognuno contiene dati e ptr al next block.

Le entry delle cartelle contengono solo l'indirizzo del primo blocco (head).



+)Crescita senza limiti +)Accesso sequenziale semplice

+)Nessuna frammentazione esterna: size non dichiarata e deframmentazione non richiesta

-)Performance deboli (tanti seek per sequenziale e scansione per lineare)

-)Overhead per ptr next

-)Non affidabile per perdita ptr: posso usare le linked list doppie ma aumento il costo

{ES: F=4Kib, B=1Kib, P=4B N=F/(B-P)=4kIB/(1KIB-4b)=5

Overhead(P/B)%=0.39% Spreco:(1-(N\*PFf))/(N\*b)%=19.61%}

**Schema: allocazione su "Cluster"**: estende lo schema a liste.

Collezione di blocchi contigui in "cluster" di lunghezza fissa invece che blocchi.

1 Cluster=4 blocchi e nextPtr.

+)Basso overhead +)Performance migliori delle liste: acc seq. meno seek perché i blocchi nel cluster sono contigui e acc random perché scansiono solo i cluster e accedo direttamente ai blocchi.

-)Frammentazione interna nei cluster -)Spazio per ptr -)Perdita ptr

{ES: F=4Kib, B=1Kib, P=4B, Cluster lenght C=4

M=4Kib/(4\*1Kib-4B) N=M\*C Overhead=P/c\*B %; Wasted=1-(M\*P+F)/N\*B %}

**Schema: linked-list +FAT**

Fat: tabella dell'allocazione files, contiene un’entry per ogni blocco, ogni entry contiene il ptr al next o un simbolo se è l'ultimo blocco. Le entries salvano i ptr all'head. Salvata all'inizio di una partizione nel disco.

-)2xSeek di allocazione linked-list ->porto la FAT nella memoria principale.

+)Tutti i vantaggi delle linked list +)No overhead nella FAT +)accesso random + veloce

-)Inaffidabile: rischio perdita ptr->copie di backup della FAT

{es: f=4kIB; B=1Kib; P=4B; disk partition size: D=256GiB

number of FAT entries: M=D/B=268435456

FAT size: M\*P (numero di entry/ptr \* dim ptr)=CIRCA 1gIb

Blocchi allocati N=F/B=4 Overhead=0.39% Wasted=(1-4Kib/(4\*1Kib))%=0}

**Schema: allocazione indicizzata**: ogni file ha la propria tabella delle allocazioni (index block) salvata in 1 o più blocchi contenenti gli indirizzi del blocco.

La entry iesima punta al blocco iesimio, per trovare il blocco “i” uso il ptr all'iesima entry.

L'index block viene caricato in memoria solo quando il file correlato è aperto, e viene scaricato quando il file è chiuso.

+)Occupa meno spazio della FAT +)Stesso schema di allocazione delle liste

+)No overhead nei blocchi (ptr nell'index block) +)Accesso casuale rapido e efficiente

-)Overhead>liste (allocazione index block) -)Spreco>liste (spreco spazio nella index block)

-)Dim file limitata dalla dimensione dell'index block

ES: block ptr size:4B; disk block size:1Kib; index block size: 1 disk block, 4 disk blocks

Con 1Kib index block: 2^10B/2^2B=2^8 entries \* 1Kib=256Kib

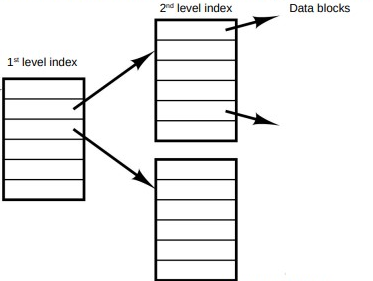
Con 4Kib index block: 2^12B/2^2=2^10entries\*1Kib=1Mib

Quanto dev'essere grande l'index block?

Piccola per ridurre overhead e sprechi ma non troppo in caso non basti.

Estensioni:

* **linked scheme**: un file può avere diversi index block organizzati come linked list, ogni blocco ha una entry per salvare il ptr next, nessun limite di dim
* **multilevel scheme**: ogni livello di blocchi punta a un set del livello inferiore fino all'ultimo livello che punta al blocco del file. Dim determinata dal numero di livelli.



ES: B=4Kib; P=4B, index block size:4Kib.

L-level index: max size = 4KiB\*(2^12B/2^2B)^L=4MiB (livello=esponente)

* **combined scheme (index-node** o **i-node):** index-node (index-block principale).

Le prime n entries del nodo sono ptr diretti al blocco, contengono indirizzi ai blocchi dei file, le altre sono ptr indiretti che contengono inidrizzi di indici multilivello

{ES:12 ptr a blocchi diretti +3 ptr indiretti singoli, doppi o tripli.}

-)Max dim file: determinata da struttura e dimensione degli index-node.

Per N ptr diretti e M indiretti, ogni index block ha L entries.

Numero di data blocks indirizzabili: A=N +∑ (i=1->M) L^ i

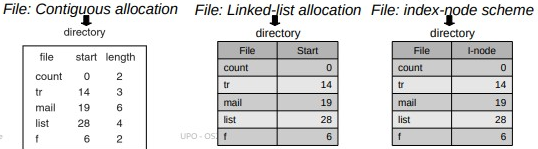
Dim max file S=A\*bytes per blocco

{ES: N=10, M=3, L=256, B=1Kib -> A=10+ 256^1 + 256^2 + 256^3=16843018

S=A\*1Kib=16.06 GiB

+)Flessibile

**Cartella**: collezione di entries (una per ogni file contenuto nella cartella) che mappano il nome di un file nelle informazioni necessarie per trovare il suo blocco.



Una cartella può contenere gli attributi di un file o possono essere nel file control block.

Come gestisco nomi lunghi di file nelle entries?

1. Non accetto nomi lunghi
2. Uso un campo di lunghezza fissa e setto la sua dim molto grande->semplice ma spreco.
3. Entry con lunghezza variabile: inizia con un header contenente la lunghezza della entry e degli attributi (info su proprietario, creazione...) poi c'è un campo di lunghezza variabile che termina con un carattere speciale.

-)Frammentazione esterna->deframmento.

1. Entry con lunghezza fissa e tengo i nomi in un'altra area: ogny entry ha un ptr alla heap.

+)No frammentazione esterna

-)Frammentazione interna

Operazioni nelle cartelle: ricerca file, creazione e cancellazione.

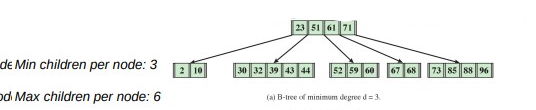
Implementazione cartelle:

* lista lineare: ricerca = scansione lineare. Semplice ma inefficiente
* hashtable: ogni cartella ha una hash table che mappa il nome di un file in un ptr alla entry per il file. Le entries sono organizzare come una lista.

ricerca = calcolo valore hash e possibile scansione delle liste di collisione.

+)Facile -)Complesso -)Overhead per la tabella.

* B-tree: albero bilanciato che ha in ogni nodo un numero variabile di chiavi e figli (>=d, <=2d). Le chiavi sono in ordine crescente. Figlio sx minore e dx maggiore.



Operazioni: ricerca, inserimento e cancellazione della chiave.

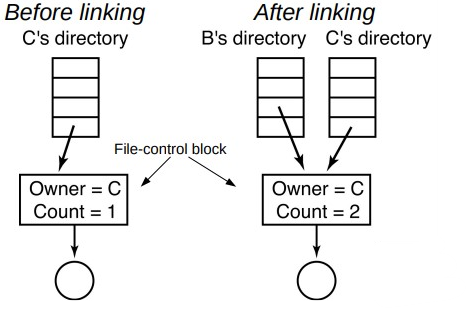
Costo max = costo dei nodi = O(log n).

* Approccio ibrido: pochi file-> lista lineare. Tanti file->B-tree.
* Caching: mappo il percorso per un ptr per una entry. Prima di cercare un nome controllo se è in cache, se c'è lo prendo se no lo cerco e poi salvo in cache.

**FILE CONDIVISI**

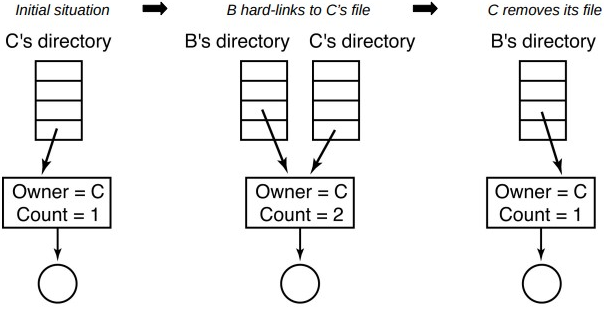
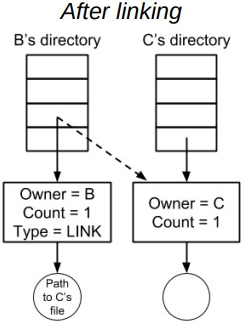
File che appare contemporaneamente in più cartelle di diversi users.

Le cartelle coinvolte contengono dei link al file condiviso che cambiano l'organizzazione da un albero ad un DAG.

Come rendo le modifiche valide/attive per entrambi gli users?

* Hard linking: i disk block sono nel file-control block del file condiviso. Hard links sono entries di cartella che puntano al file-control block.

Se il proprietario cancella il file, l'altro utente avrà una entry che punta ad un file non valido o diverso ed è difficile rimuovere tutte le entries -->link-count: se ==0: rimuovo il file-control block perché nessuna entry punta a quel file, altrimenti se C rimuove il file ma link-count>0: link-count--



C può contattare l'amministrazione per cedere il file che non usa più.

* Symbolic linking: creazione di un file di tipo link (link simbolico/soft) e mettendolo nella cartella di un altro user. Il file contiene solo il path al file originale e non influenza il link-count.

Rimuovo file originale->link inutile.

+)Soft links possono puntare alle cartelle.

+)Non sono file veri e propri->più facili da trovare.

+)Alcuni SO permettono hard-linking solo agli users.

+)Possono linkare a diverse partizioni, file systems o pc.

-)Invalido se rimuovo il file

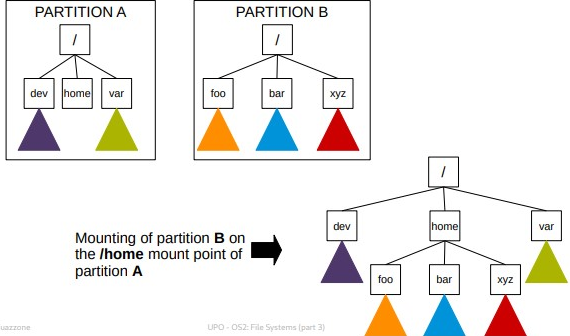
-)Overhead per accedere al link e per salvarlo

Creano + percorsi per un file->più tempo nelle ricerche->come lo gestisco?

a)ignoro: subisco il costo e posso finire in un loop; b)supporto limitato: zip dei link; c)supporto completo: tar

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Link | Posso rimuovere file originale? | Posso spostare file originale? | Influenza link counter? | Istanzia nuovo i-node quando lo crei | Posso spostare il file link? |
| Hard | No | Sì | Sì | No | Sì |
| Soft | No | No | No | Sì | Sì |

**Montaggio di un file system:**  da fare prima di accedere al File system, aggancia il file system ad un punto di montaggio e lo rende disponibile per il sistema.



**Smontaggio di un file system:** sganciamento del file system dal punto di montaggio.

**Punto di montaggio**: cartella dell'host o punto di accesso indipendente, solitamente è una cartella vuota, se non vuota: 1)Oscuro i file precedenti 2)Monto il F.S. 3)Rendo visibili anche i file precedenti.

Unix: root file system "/" è sempre montato all'avvio (boot time) e posso attaccarci o staccarci altri file system.

Prima di fare una operazione di montaggio: SO controlla presenza di un file system: legge il superblocco e controlla che il formato sia valido (se non valido controllo la consistenza) --> Guarda nella mount table che il F.S. sia montato e del tipo giusto.

ES: montaggio in Unix: 1)controllo di consistenza nuovo F.S.

2)Crea struttura dati per tenere di metadati del nuovo FS

3)aggiorno la mount table aggiungendo una entry con ptr alla copia del superblocco

4)Attacca il nuovo FS al punto di montaggio settando 2 campi della copia del nodo: flag che indica che è un punto di montaggio, campo che punta alla entry della mount table.

ES: attraversamento punto di montaggio: 1)Riconosco il punto di montaggio per la flag

2)SO legge le entry della tabella e analizza il superblocco

3)SO può iniziare ad attraversare le cartelle secondo il formato del file system

ES: montaggio in windows: permesso con l'introduzione del file system NTFS v.3.0 (New Technology File System)

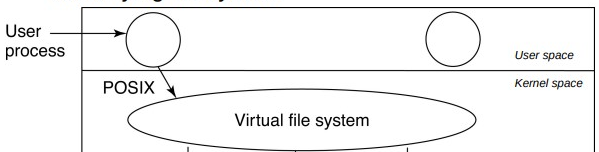
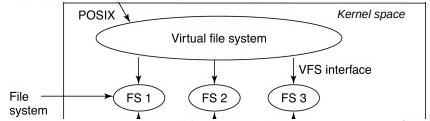
Prima di NTFS v.3.0: schema a 2-liv: namespace + path name; file system montati in un namespace separato denotato da una lettera e colonna, es: X:\path\file.

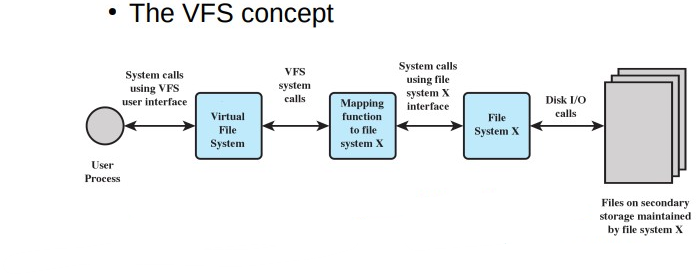
Con NTFS v.3.0: SO può montare un file system in ogni punto (volume mount point) con la struttura esistente di NTFS, il FS è supportato da ogni FS di Windows e il mount point ha un riferimento alla root dir.

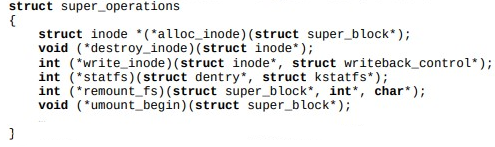
**VIRTUAL FILE-SYSTEM (VFS)**: integra più file system in una struttura singola ed omogenea, i suoi FS possono essere remoti o in partizioni di dispositivi locali.

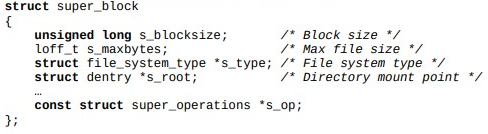
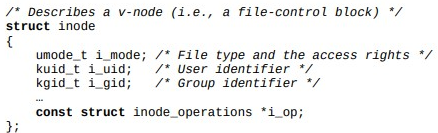
Astrazione: isola aspetti comuni ad ogni FS in un seprato layer del software, il layer è come un ponte: esibisce interfaccia comune astratta a liv superiori e chiama le funzioni dei FS dei livelli inferiori.

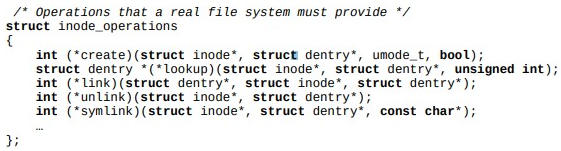
VFS ha un'interfaccia superiore al processo utente e una inferiore al FS concreto, ogni FS concreto ha le funzioni richieste dal VFS

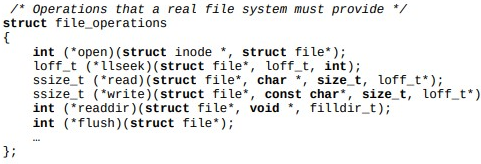
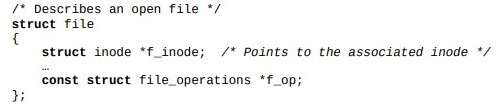
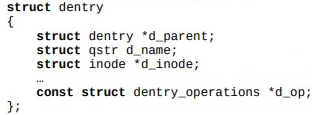
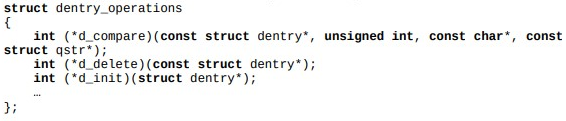
 



Astrazioni usate dall'interfaccia VFS:

* **superblocchi**: descrivono un FS montato
* **v-node**: descrivono un file-control block associato ad un file specifico

****

* **file**: descrive un file aperto associato al processo
* **dir entry**: descrive una entry di una cartella

Ognuno ha un ptr a una tabella di ptr a funzione con tutte le possibili operazioni di quel tipo.

VFS ha delle strutture dati interne, tra cui:

* **mount table**: tiene traccia dei file montati
* **tab file aperti nel sistema**: tiene traccia di tutti i file aperti
* **tab file aperti per processo**

Il VFS non salva su disco, crea e gestisce delle strutture in memoria, è il software associato con un FS concreto che fa il fetch e salva i metadati e dati del FS dal/sul disco.

Aggiungere supporto per un concreto FS in VFS:

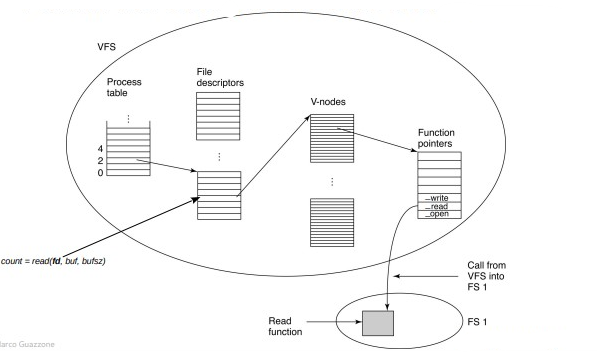
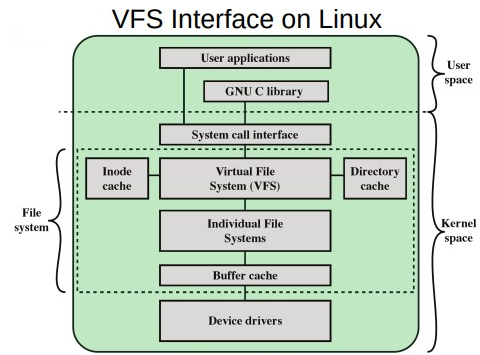
* FS nuovo: scrivo funzioni con la stessa firma richiesta da interfaccia VFS
* FS esistente: scrivo funzione "wrapper", la firma è la stessa delle funziooni dell'interfaccia VFS. L'implementazione consiste nel creare 1 o + chiamate al file system concreto

Un FS montato dev'essere registrato con VFS prima di essere usato.

ES: montaggio FS: 1)registra il FS

2)Creazione e aggiornamento strutture dati (creo superbloco, metto una entry con un suo ptr nella mount table e aggiorno i v-node) 3)Posso usare il FS

ES: process calls fd = open(“/usr/include/unistd.h”, O\_RDONLY)

1. Analisi /usr ->FS già montato
2. Mount table: VFS trova il superblocco
3. Superblocco: VFS trova la cartella root del FS montato
4. Cartella root: segue il path include/unistd.h
5. Trova il file, crea nuovo v-node in mem che contiene un ptr
6. VFS chiama la funzione open e mette il risultato nel v-node
7. Creazione nuova entry nella v-nodes table in cui savare i v-node
8. VFS crea nuovo file che punta alla entry associata al v-node (crea salvataggio del return)
9. crea entry del processo corrente nel fd table in cui salvare il fie creato
10. Return del fdal chiamante, così può usarlo per altre operazioni e vedere file/v-node/ptr associati.

**Gestione dello spazio su disco**

I file sono collezioni di blocchi di disco di lunghezza fissa--> Ma quale lunghezza?

**DIMENSIONE DEI BLOCCHI**

Con blocchi troppo grandi i file di piccola dimensione sprecano una grande quantità di spazio su disco (frammentazione interna). Con blocchi troppo piccoli i file di grandi dimensioni si estendono su più blocchi, portando così a un sovraccarico di spazio maggiore e una perdita di tempo (a meno che non venga utilizzato uno schema di allocazione contiguo.

Blocco di 1KB: contiene 30-50% dei file.

Blocco di 4KB: contiene 60-80% dei file, 93% spazio usato dal 10% dei file (i più grandi). Leggere un file grande con tanti piccoli blocchi sarebbe lento (tanti accessi a disco).

La dimensione dei blocchi influisce sulla velocità dei dati (data rate direttamente proporzionale con unità di misura bytes trasferiti al secondo) e efficienza in spazio inversamente proporzionale. Pertanto cerco equilibrio sulla scelta della dimensione del blocco.

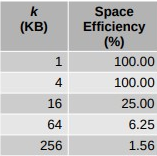
**ES:** bytes per traccia:1MB; rot speed:7200RPM (al minuto); seek time medio:5msec.

Quanto tempo ci vuole per trasferire K BYTES?

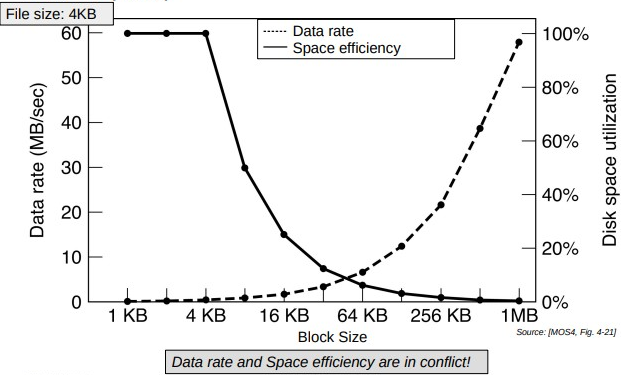
Rot. time: 60\*10^3msec/7200RPM=1minuto/4200RPM-

Avg ritardo di rotazione=8.33MSEC/2=4.165msec

Avg tempo trasferim. di k bytes: k\*(8.33msec/10^6).

Avg tempo di accesso per trasferire k B: seek time + ritardo rot + tempo trasf k = 5+4.165+ k\*(8.33msec/10^6). Dominato da sekk time e rotation delay

Dato una dimensione file, meno spazio alloca e più è efficiente.



Potremmo scegliere una dim di 64KB che è all'incrocio ma entrambi troppo bassi.

Prima sceglievo 1-4KB, ora meglio >4KB e accettare gli sprechi.

ES: In Linux posso creare un FS con mkfs.<type> (Es:mkfs.ext2 oppure mke2fs -t <type>)

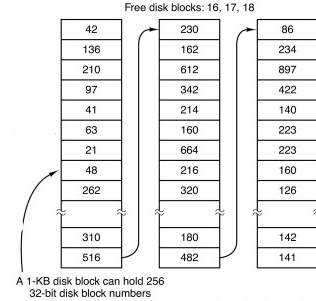
ES: per conoscere dim attuale dei blocchi:

$sudo blockdev --getbsz /dev/sdaX

$sudo tune2fs -l /dev/sdaX | grep -i 'block size'

$stat -f /path/to/mount/point/of/fs/on/dev/sdaX

**Come tengo traccia dei blocchi liberi?**

#APPROCCIO 1: **LINKED-LIST**: creo una lista di blocchi liberi (free list) dove ogni nodo è un disk block che contiene ptr a blocchi liberi e nextptr

/\*\*ES: disk block:1KiB e disk-block ptr:4B. Quanti ptr in un nodo di lista?

Un nodo contiene 1Kib/4B=2^10B/2^2B=2^8=256ptr-1 next=255 ptr a nodo

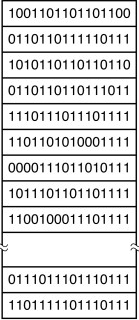
ES: disk size:1TiB, disk block:1Kib, disk-block ptr: 4B. Max nodi nella free list? max spazio occupato nella free list?

Ptr per nodo: 2^10/2^8 -1=255ptr.

Blocchi nel disco: 1T/1Kib=2^40/2^10=2^30 blocchi

Max nodi per lista: blocchi/ptr=2^30/255=2^22 circa

Spazio occupato dalla ista: 2^22\*2^10=2^32 \*\*/



#APPROCCIO 2: **BITMAP**: 1 bit k per ogni blocco del disco, se k=1->libero else usato.

/\*\* disk size: 1TiB, disk block size:1KiB, dim della bitmap? spazio occupato \*bitmap?

\* blocchi su disco: 1TiB/1Kib=2^(40-10)=2^30blocchi

\* bitmap=2^30blocchi\*1 bit/blocco =2^30 bits=2^27bytes=128MiB

\* Blocchi per contenere tutta la bitmap: 2^27B /1KiB=2^(27-10)=2^17 blocchi

\* Spazio occupato:2^17blocchi\*1Kib per blocco=2^(17+10)=2^27 \*\*/

+)Richiede MENO SPAZIO della free list

-)Peggiore a disco quasi pieno: pochi dischi lberi->scansione lista meno veloce.

Se i blocchi liberi arrivassero in gruppo (allocazione contigua)?

Meglio tener traccia dei gruppi di blocchi invece che dei singoli. Disco vuoto=1 grande spazio, migliore. Frammentazione alta: tanti spazi, peggiore.

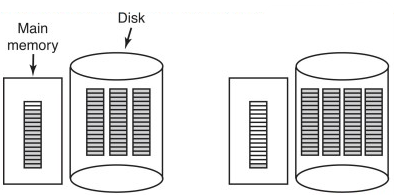
Le strutture per la gestione degli spazi liberi sono salvate su disco ma tenute in memoria per ridurre operazioni I/O. Con la free-list possiamo tenere in mem solo la head e lasciare gli altri su disco

/\*\*Creazione file: 1)Prende un nodo in memoria 2)Finisce i ptr, legge un nodo da disco

\* Anche in caso di cancellazione file, libero il blocco e lo aggiungo ai nodi in memoria.

\* Nodo pieno->scrivo su disco e creo nuovo nodo \*\*/

/\*\*ES: nella main Mem ho spazio per 2 entry, arriva un file da 3 blocchi-->riempio il nodo di 2 --> porto il nodo su disco -->ne creo un altro in cui metto la entry rimanente.



Quando i nodi in memoria sono quasi vuoti, una serie di file temporanei può causare molte operazioni. Come riduco le operazioni?

Divido il nodo pieno ottenendo nodi su disco pieni (eccetto l'ultimo mezzo vuoto) per ridurre l'uso del disco e tengo i nodi in memoria mezzi pieni per gestire creazione ed eliminazione con poco I/O su disco.

Nodo in memoria pieno = lo scrivo su disco e leggo un nodo mezzo pieno dal disco (o lo creo).

Bitmap: posso salvare in mem solo qualche blocco invece di tutta la bitmap, ma devo caricare il blocco giusto, aggiorno i bits e mantengo in memoria e/o scrivo su disco quel blocco e quello precedente.

**Quote di disco**: sono meccanismi usati per evitare che lo user consumi troppo spazio su disco.

L'amministratore assegna ad ogni user una quota massima di spazio da occupare sia per i blocchi di disco sia per i file e inoltre impone due limiti per non superare lo spazio da occupare che ci è consentito, se è un soft limit potrei anche superarlo, se è un hard limit non posso.

Quota record: sono voci per ogni user della tabella delle quote, salvo su disco il quota file, carico parzialmente in memoria quello che mi serve. Quando un utente fa log out o chiude i file, il quota record viene scritto su disco.

Per ogni file aperto: il file control block nella tabella dei file aperti contiene ID del proprietario e un puntatore al record in memoria. ID serve per caricare il record corretto e cambiare le dim dei file.

Ogni volta che aggiungo un blocco ad un file aperto, il numero di blocchi caricati sul proprietario del file aumenta nel record della quota, se lo tolgo diminuisce e controllo il limite.

Creo/aumento file-> controllo-> warn se violo un soft, arop aw violo un hard.

**Back-up:** back-up del file system per gestire il recupero dei dati da disastri (crash, fuoco, acqua, catastrofi...) o errori umani (rimozione non sicura chiavetta, cancellazione...)

Richiedono molto spazio e tempo.

Lo faccio solo di alcuni files (con file di configurazione...) e ci sono alcuni che devo evitare per non andare in crash (file speciali). Fare backup di file che non sono cambiati dall’ultimo backup è una perdita di spazio e di tempo. Due tipi di backup:

**Back-up differenziale:** in questo caso si fa il backup dei file che sono stati modificati dall’ultima volta che è stato fatto l’ultimo backup completo. Un vantaggio di questo tipo di backup è che il recupero dei file è semplice inoltre è affidabile mentre, uno svantaggio è che richiede molto spazio di memorizzazione. Per esempio, se si fa un backup completo al giorno 0 poi si modifica il file A al giorno 1, si modifica il file B al giorno 2 e si il backup differenziale al giorno 2 si includono i file A e B, dopodiché si modifica il file C al giorno 3 e si fa il backup differenziale al giorno 3 si includono i file A, B e C poiché non si è ancora fatto un altro backup completo per questo si includono ancora i file A e B nel backup del giorno 3.

**Back-up incrementale:** in questo caso si il backup dei file che sono stati modificati fino a quell’istante. Richiede meno spazio ma è meno affidabile di quello differenziale (se uno si perde->perdo tutti) e complesso da recuperare (devo recuperarne tanti).

/\*\*ES: back-up differenziale: completo al giorno 0 + differenziali giornalieri.

Giorno 0: back-up completo. Giorno 1: cambia F2->dff. su F2

Giorno 2: nessun cambiamento->diff F2. Giorno 3: cambia F3,F6->diff F2,F3,F6

Recupero al giorno 3: giorno 0+giorno 3\*\*/

/\*\*ES: back-up incrementale: completo al giorno 0+daily incrementale.

Giorno 0: completo. Giorno 1: cambia F2->incr F2 Giorno 2: cambia nulla->niente

Giorno 3: cambia F3, F6-> incr F3, F6\*\*/

🡪Perché non si effettua un backup singolo completo? Si potrebbe fare così un backup completo e poi un backup incrementale o differenziale ma non è una buona scelta poiché con il passare del tempo i backup incrementali o differenziali diventerebbero più dispendiosi sotto l’aspetto dell’affidabilità, del recupero dei dati e dello spazio rispetto a un backup completo, quindi, in conclusione conviene fare periodicamente un backup completo.

La compressione aiuta a risparmiare spazio, ma richiede tempo ed un bad block può rovinare la decompressione.

🡪Si devono supportare backup online (file system montato) o offline (file system smontato)?

Back-up **online**: può non essere aggiornato o perdere file spostati.

Back-up **offline**: nessuna perdita ma non sempre posso spegnere il PC per ore. (troppo tempo)

Per questo motivo sono stati creati degli algoritmi per poter fare dei backup (**Snapshot**).

**Snapshot**: congela un momento il file system, le modifiche vengono fatte su copie dei file o di directory (copy on write) per permettere il back-up.

🡪Dove dovrebbero essere salvati i backup? Su un altro dispositivo (per evitare catastrofi) lontano da persone non autorizzate.

Su un disco si possono eseguire due tipi di backup:

**Scaricamento (dump)** in un back-up:

**🡪dump** **fisico**: effettua il backup di tutti i blocchi (usati e non) e inizia dal blocco 0 del disco e li scrive tutti in ordine fino all'ultimo, si ferma quando l’ultimo blocco del disco è stato copiato. I vantaggi di questo approccio sono: indipendenza dal file system, è semplice e veloce, anche per file system non montati. Non si appoggia a nessun file system e non vengono modificati i metadati.

-) C’è un problema dei blocchi non usati: fare il dump di blocchi non usati è inutile, per evitarli devo accedere alla struttura dati dei blocchi liberi. Il programma di back-up deve conoscere i dettagli del file system e rottura del mapping 1-1 dei blocchi tra disco su cui salvo e quello su cui faccio il in-up. Più complicato tenere l'ordine.

-) Problema dei bad blocks: devono essere gestiti dal SO in caso appaiano dopo la formattazione a basso livello. Se letti danno errore quindi devo riconoscerli oppure ignorare gli errori di lettura.

-) Problema file di sistema interni: ci sono file che devo ignorare perché sono file di sistema che potrebbero non essere validi in futuro quindi devo distinguerli ed ignorarli.

-) Incapace di ripristinare su file system diversi dall'attuale, di fare back-up selettivi, differenziali o incrementali.

**🡪dump logico**: parte da 1 o più specifiche cartelle e fa un dump ricorsivo su tutti i file che trova.

Il dump logico deve includere tutte le cartelle (anche quelle non modificate) che si trovano nel percorso di un file o directory modificato per rendere possibile il ripristino su un altro PC, per permettere di incrementare i ripristini e gestire file con lo stesso nome ma path diverso.

+) Può saltare file e cartelle selezionate, fare back-up differenziali/incrementali e ripristinare singoli file

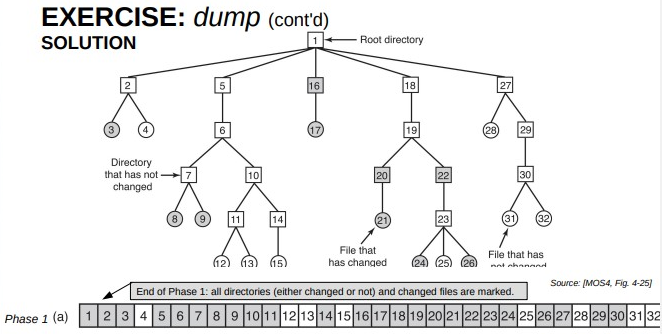
-) Richiede conoscenza della struttura interna, più lento del backup fisico.

🡪(esempio backup logico) Algoritmo di **dump basato su UNIX**: esamina files, decide di quale fare il back-up e copia i file in una memoria esterna. Solitamente c'è un comando per il dump e uno per il restore.

Mantiene una bitmap indicizzata per numero di i-node ci sono uno o più bit associati agli i-node.

Ci son 4 fasi del dump:

1. Esamino le entries delle cartelle ricorsivamente: marchio la bitmap di tutte le cartelle modificate e non modificate e dei file modificati trovati. Alla fine della visita tutti i file modificati e tutti le directory modificate o non sono marchiate.
2. Attraverso ancora l'albero per togliere il marchio alle cartelle senza file o sottocartelle modificati/e. Alla fine di questa visita sono marchiati solo file e directory modificate.
3. Scansione degli i-node nella bitmap in ordine numerico e dump di tutte le cartelle marcate. Dopo che ho scaricato una cartella, tolgo il marchio. Quindi dopo questa fase avremmo marcati solo i file che erano stati marcati e non avremmo più directory marcate.
4. Scansione dei nodi nella bitmap in ordine numerico e dump dei file marcati. Alla fine di questa fase non ci dovrà essere nessun file e nessuna directory marcata.









**Ottimizzazione:** fase 1 e 2 possono essere unite e formare una singola scansione, non lo faccio con le altre fasi per non complicare il processo di ripristino.

**Ripristino** **di un file system:** creo un FS vuoto, ripristino il Back-up completo più recente e gli incrementali successivi (prima ripristino le cartelle e poi i files).

#Problema 1: la lista dei blocchi liberi non viene scaricata non essendo un file, devo ricostruirla dai file ripristinati.

#Problema 2: link: un file linkato viene ripristinato 1 volta sola-> ricreo il link

#Problema 3: file di sistema interni: file speciali, pipes, socket... non devo fare il dump

#Problema 4: file sparsi file che possono avere dei buchi: la loro dimensione fisica è maggiore della loro dimensione logica, il file system li rappresenta come blocchi di 0. Non tutti i file system gestiscono i file con buchi, in questo caso si ripristina un nuovo file di dimensione intera.

**Consistenza di un File System**

Parliamo dell’affidabilità dei file system e degli algoritmi per garantire quest’ultima. Gli algoritmi che vediamo risolvono i problemi di inconsistenza. Per motivi di efficienza il file system di solito tiene i blocchi cambiati in memoria e li scrive sul disco più tardi, ad esempio, in fase di smontaggio del file system.

**Stato di inconsistenza**: il file system può rimanere in questo stato a causa di crash del sistema, arresto animalo, smontaggio...

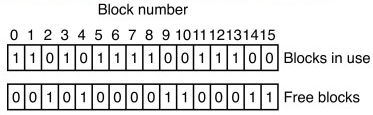
Molti S.O. hanno strumenti per controllare la consistenza dei file system UNIX ha **fsck.**

**fsck**: È uno strumento usato per controllare consistenza di blocchi e dei file in Unix.

**Consistenza blocchi**: fcsk ha 2 tabelle, la block in-use table e la free-blocks table  **che non sono usate dal file system ma solo da fsck**, per ogni tabella c'è una entry per ogni possibile blocco con un contatore che parte da 0.

Procedimento di fsck: fsck legge gli i-nodes->trova il numero di blocco e il file allocato-> count della *bock-in-use table* +1 -> esamina free-bocks data -> trova blocco libero e incrementa count.

Un File system è consistente se ogni blocco ha valori diversi nelle due tabelle.



**CASI DI INCONSISTENZA:** Se nella tabella dei blocchi non usati e nella tabella dei blocchi usati abbiamo lo stesso valore allora siamo in uno stato di inconsistenza oppure se abbiamo in una sola cella di entrambe le tabelle un valore maggiore di uno.

**Tipi di errori di inconsistenza:**

**Missing block:** Abbiamo che nella tabella dei blocchi liberi e nella tabella dei blocchi usati c’è lo stesso valore in questo caso 0 (quindi c’è inconsistenza). Questo è possibile per un crash. Per risolvere questo problema bisogna aggiornare sia la tabella dei blocchi liberi usata dal file system aggiungendo quel blocco poiché è un blocco libero e anche aggiornare la tabella dei blocchi liberi usata da fsck (si imposta nella tabella dei blocchi liberi 1 quel blocco).

**Blocco libero-usato:** Abbiamo che nella tabella dei blocchi liberi e nella tabella dei blocchi usati c’è lo stesso valore in questo caso 1 (quindi c’è inconsistenza). Per risolvere questo problema bisogna aggiornare sia la tabella dei blocchi liberi usata dal file system togliendo quel blocco perché è usato e anche aggiornare la tabella dei blocchi liberi usata da fsck (si imposta nella tabella dei blocchi liberi 0 quel blocco).

**Doppio blocco libero**: In questo caso abbiamo un blocco nella tabella dei blocchi liberi che è maggiore di 1. free>1 (linked list) ->Crash dopo aggiunta in free-block list ma prima di controllo duplicati. Per correggere tale errore fsck ricostruisce la struttura dati e rimuove le entries doppie.

**Doppio blocco usato:** In questo caso abbiamo un blocco nella tabella dei blocchi usati che è maggiore di 1. in-use>1. Errore Grave.

1-Alloca un blocco per ogni duplicato (count-1). Quindi crea un blocco per tutti i file che vogliono quel blocco, quindi, alloca n-1 blocchi con n numero di file che vogliono quel blocco. Il numero di file è indicato nella tabella dei blocchi usati. Quindi si allunga la tabella dei blocchi usati e la tabella dei blocchi non usati.

2-Copia il contenuto dei blocchi duplicati nei nuovi blocchi liberi.

3-Per ogni file associato ai blocchi duplicati: fsck alloca una delle copie del blocco e la dealloca dal blocco duplicato.

4-Reporto l'errore per ispezionare il danno.

**Consistenza file**: fsck usa 1 tabella (files-in-use table) per tenere traccia di quanto spesso ogni file appare nell'albero delle cartelle.

Partendo dalla cartella root fsck discende l'albero analizzando ogni cartella e per ognuna, per ogni i-node nella cartella incrementa il corrispondente contatore nella tabella file-in-uso.

Il contatore può essere >1 a causa degli hard-link.

Il file system è consistente a livello di file se:

**Consistenza a livello-file:** Per ogni file, il relativo contatore nella tabella dei file in uso concorda con il contatore di collegamento nell'i-node corrispondente

Per alcuni file, i valori del contatore di collegamento e del contatore di utilizzo dei file non coincidono

Due tipi di errori:

1– Il contatore di link nell'i-node è troppo alto

2– Il contatore di link nell'i-node è troppo basso

**Inconsistenza a livello-file:** contatore dei file in uso è diverso dal link counter

1. **Link counter > file in uso:** quando il file in uso scenderanno a 0, il link counter sarà maggiore e perciò non verrà rimosso -> non critico ma genera una perdita di spazio. La soluzione per correggere questo errore FSCK corregge link-counter.
2. **Link counter < file in uso:** quando il link counter scende a 0 rimuovo blocchi dell'i-node e quindi alcune cartelle punteranno ad un i-node non usato. FSCK: corregge il link-counter.

**Controlli**: consistenza file e blocchi sono spessi integrati per ragioni di efficienza, ma sono possibili anche altri controlli:

Controllo stato i-node: controlla corruzione e altri problemi (ogni i-node ha un campo valido). Errori difficili da risolvere->i-node sospetto e pulito da FSCK (blocchi rilasciati e struttura dati aggiornata).

Controllo cartelle: per ogni i-node in una cartella controllo che abbia un legal number e che sia allocato, che "." e ".." siano presenti e i permessi dei file.

Controllo superblocchi: fsck fa un controllo per trovare superblocchi sospetti e in caso usare delle copie alternative dei superblocchi.

Controllo dei blocchi danneggiati: verifica la presenza di puntatori a blocchi danneggiati durante la scansione dell'elenco di tutti i puntatori di blocco negli i-node.

**File system con journaling**: ha l'obiettivo di essere robusto e resistente ai fallimenti (ovvero agli errori).

**Funzionamento:** il file system tiene un log delle azioni da fare prima di farle, se viene smontato in modo errato, al prossimo montaggio il S.O. controlla nel log per vedere cosa doveva fare e finisce il lavoro. Usato da Microsoft, Linux, Apple...

Consideriamo i passi da fare per rimuovere un file:

Rimozione files:

1. Rimuovo la entry della cartella.
2. Rilascio l'i-node tra gli i-node liberi.
3. Restituzione di tutti i blocchi di disco al pool dei blocchi di disco liberi.

L'ordine importa solo in caso di smontaggio scorretto.

#Scenario 1: rimuove la entry di directory e poi avviene un fallimento, Quindi le i-node e i blocchi sono inaccessibili e perciò non allocabili. C’è uno spreco di memoria.

#Scenario 2: rimuove la entry di directory rilascia i-node e poi avviene un fallimento. Quindi non posso accedere ai blocchi. C’è uno spreco di memoria.

#Scenario 3: rilascia i-node e poi avviene un fallimento la conseguenza sarà che una entry delle cartelle puntano a file errati o i-node non assegnati, perciò, non posso riassegnare i blocchi dei file la conclusione è che c’è uno spreco di memoria.

#Scenario 4: rilascio blocchi file e poi avviene un fallimento quindi le entry puntano ad un i-node valido che punta a blocchi liberi o riassegnati la conseguenza è che ci saranno diversi files condivisi tra bocchi.

Vediamo una soluzione a questi possibili scenari. Per evitare tali problemi, il file system di journaling utilizza un **registro write-ahead (journal).**

Soluzione: **Write-ahead log (journal):**

1. il file system scrive un log entry (transazione) mettendo in lista le operazioni da eseguire su disco
2. la transazione viene scritta sul *journal file* su disco(commited)
3. eventualmente viene letto da disco per verificare che sia corretto,
4. una log entry viene scartata se si verifica un crash durante la scrittura.
5. Quindi esegue le operazioni appena registrate sul disco.
6. Dopo che le operazioni sono state completate con successo, la voce di registro può essere cancellata

**ES: rimozione files:** Compio transazione su journal con le 3 operazioni da fare.

Fine transazione -> svolgo le operazioni in lista rimuovendo il file. Quando sono concluse cancello il log. Caso di smontaggio errato: recupero dal log le operazioni pendenti e le rifaccio.

Il concetto di transazione viene da DBMS:

* **Begin**: crea nuova transazione e comprende anche gli aggiornamenti
* **Commit**: controlla che la transazione corrente sia scritta, scrive "commit" a fine transazione, fa il "write behind" applicando aggiornamente alle transazioni passata e rimuove la transazione dal log
* **Rollback**: rimuove la transazione dal log
* **Failure recovery**: se i log è vuoto non fa niente, altrimenti scarta la transazione senza commit

Le operazioni devono essere **idempotenti**: possono essere ripetute più volte senza errori.

ES: NON "aggiungi il blocco appena libero a fine della free list" ma "cerca la lista dei blocchi liberi e aggiungi il blocco n solo se non è già presente".

QUANDO AGGIORNO IL DISCO CON LE TRANSAZIONI?

a)sempre->troppo costoso

b)periodicamente + se troppe, log quasi pieno o FS sta per essere smontato.

**Log circolare:** gestito come un buffer circolare, ogni entry non vuota è una transazione completa e appaiono dalla più vecchia.

Tutte le operazioni di una transazione completata sono su disco->log entry = "done" -->la entry è un **checkpoint** (posso riusarlo).

Il ripristino parte dalla transazione seguendo *l'ultimo checkpoint* e proseguendo fino alla entry del checkpoint o una entry vuota.

Problemi di performance: scritture doppie (anche il log al journal) e in diverse parti del disco (richiedono il seek).

Soluzioni: allocazione sequenziale dei log (meno seek), aggiornamento dei log batch (una transazione unica per un gruppo/batch), mantenimento dati in RAM, log solo dei metadati.

FSCK serve ancora? Sì, per riparare situazioni inaspettate e errori di smontaggio e per fare altri controlli.

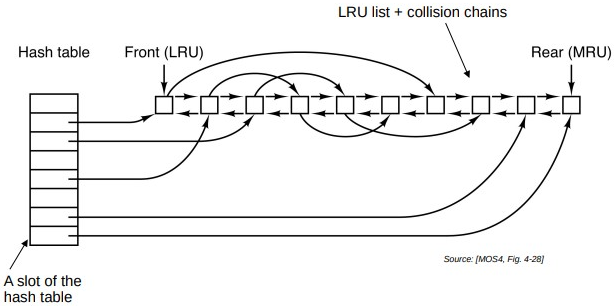
Troppi accessi-->performance basse-->servono ottimizzazioni:

* **Block caching:** riduce gli accessi a disco, usa una cache/buffer (collezione di blocchi logicamente sul disco ma in memoria per le performance).

Per ogni read/write: controllo se il blocco è in cache (se non c'è lo leggo da disco e aggiungo) e poi il blocco può essere letto o scritto.

??Come controllo velocemente se un blocco è in cache??

Uso hashtable con concatenazione separata: computo il valore hash nell'indirizzo e guardo nella tabella, tutti i blocchi con indirizzo = sono in una lista. Complessità O(1)-O(n).

??Come seleziono la vittima in cache??

Cache piena->devo rimuovere qualcosa --> algoritmo: FIFO, LIFO, LRU (aggiornato ad ogni utilizzo)... LRU-->bidirectional list: Least recent<--->Most recent.

!!Problema: cache = modifiche in cache perse (stato inconsistente).

Sol: uso LRU: quando un blocco viene inserito nella LRU list:

??Userò nuovamente quel blocco tra poco??

No-->lo metto a inizio lista (lru). Sì-->lo metto a fine lista (mru).

??Il blocco è fondamentale per la consistenza del file system??

Lo sono tutti tranne i blocchi di dati, gli altri (non dati) li scrivo subito su disco.

Alcuni blocchi di dati rimangono in cache per troppo tempo ed è pericoloso perché sono importanti. Posso rispondere con diversi approcci:

**#Approccio UNIX**: uso write-back caches (cache di risposta): un blocco modificato viene segnato come "dirty" e scritto si disco più tardi. Il programma del SO aggiorna periodicamente (spesso 30 sec) i blocchi dirty invocando una chiamata di sistema "sync" che sposta i blocchi "dirty" su disco.

+)Efficiente -)Poco affidabile -)Troppo coplesso

**#Approccio Windows**: write-through chaches: simile a Unix, la sua system call si chiama FlushFileBuffers -->non scrive dopo ma subito.

-)Performance basse (richiede molto I/O disco, non buono per tante operazioni)

+)Molto affidabile +)Semplice

**Block read-ahead (prefetching)**: tecnica per ottenere blocchi in cache prima che siano necessari, aumenta l'hit-rate.

Usa il principio di località spaziale: se k -->k+1 e k+2 -->perfetto per accesso sequenziale, dannoso per accesso casuale -->Bit (aggiornabile) per distinguere accesso sequenziale/casuale --> sequenziale=read-ahead (località spaziale), casuale = stop read-ahead.

**Ridurre movimenti braccio (seek)**:

#Tecnica 1: località dai dati: metto vicini i blocchi a cui probabilmente accedo.

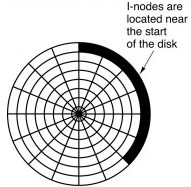
Scelta del blocco libero: quello più vicino al blocco precedente.

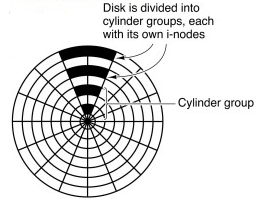
+)Efficiente con le bitmap (tiene traccia del blocco precedente)

-)Non efficiente nelle linked-list: scansione di tutta la lista dei blocchi vicini.

PREALLOCAZIONE SPECULATIVA: serve un blocco-->alloco un cluster (+ blocchi) e quando il file è chiuso libero i blocchi del cluster non usati. Usa 1 blocco, ne riserva n-1. Valido con bitmap o linked list.

ES: settore: 512B, blocco:1KiB, allocazione di un cluster di 2KiB (2 blocchi)->libero i non usati.

#Tecnica 2: nei sistemi che usano gli i-node, le letture richiedono almeno 2 accessi: 1 per i-node e 1 per data block. Gli i-nodes vengono messi a inizio della partizione-->tanta distanza dai blocchi.

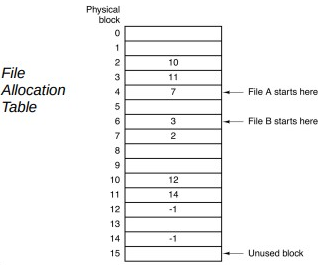
#2.1 I-node in mezzo al disco-->meno seek time.

#2.2 divido il disco in cilindri e ognuno con i-nodes, data blocks e free-blocks

Creazione file: scelgo qualsiasi i-node.

Allocazione blocco: cerco uno dello stesso cylinder group, altrimenti uno del cilindro vicino.

**/\*\*QUESTA PARTE NON C'E' NELLE REGISTRAZIONI, NON SO SE LA ABBIA SPIEGATA\*\*\**

**FAT File System**: file system che usa lo schema di allocazione file "linked-list con FAT"

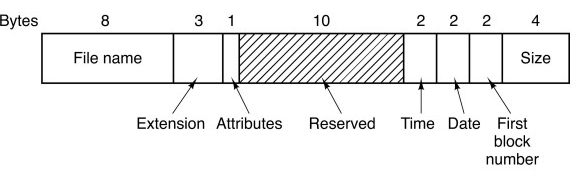
Versioni: 8-bit, fat12, fat16 (numero di bit delle entries), vfat (supporto per nomi lunghi), fat32 (32 bit ma ne usa 28), exfat (32-bit entries e usati tutti).

Il disco è diviso in blocchi di disco chiamati cluster e la FAT descrive quale cluster vengono usati da quale file e tiene traccia dei cluster liberi.

Organizzazione di una partizione disco:

* **Boot sector**: contiene il boot loader e altre informazioni riguardo il resto dei superblocchi (quante copie della FAT, grandezza settore, numero cluster...)
* **FAT #1...FAT #N**: 1 o + copie della FAT, di solito 2 e ognuna salvata con allocazione continua.
* **Root directory**: entries di cartella per la cartella radice. Salvata nella data range area.
* **Data range**: contiene clusters.

Entry della cartella(Fat12/16): dim fissa di 32B, nome file ed estensione = 8+3B.



Attributi: 1Byte:

* read-only
* archivio: if 1: il file è stato cambiato che un programma di backup deve archiviarlo.
* nascosto: non è mostrato nella listing
* sistema: nascosto + non può essere cancellato col comando DOS "del"

Riservato: 10B per uso futuro (prossima versione).

Tempo e data: 2+2B, ultima modifica.

First block number: ptr alla entry FAT corrispondente al primo cluster del file. Max 2^16 entries.

Size: 4B, dimensione del file in bytes. Max = 2^32=4GiB

FAT32: prova superare alcuni limiti del 12 e 16

* Partition size: dipende da Cluster e numero di FAT entries. Fat12=16MiB, FAT16=max 2 GiB.

FAT32->32-bit entries (uso solo 28)->max 2TiB.

* Altri problemi: max file size 4GiB e assenza supporto per nomi lunghi.

File size limitata da: max dim partition (2^bit FAT entries \* dim cluster) e struttura per le directory (campo size).

Entries. max file size in FAT32: min(max partition size, campo size).

VANTAGGI FAT32: supporta partizione disco più larghe e meno frammentazione interna.

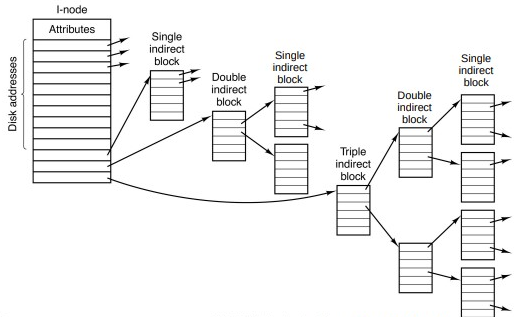
/\*\*ES: 2 GiB

\* FAT16: devo usare clusters di 32KiB, spreco se molti file sono <32Kib.

\* FAT32: posso usare diverse dim dei cluster-->4-KiB Cluster:

\* lunghezza Fat: dim partizione/dim cluster=2^31-12=2^19

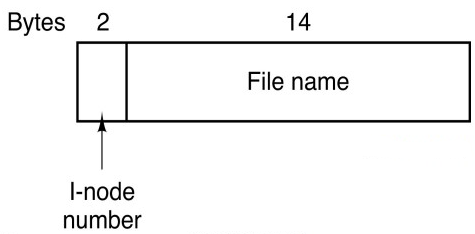
\* dimensione Fat: lunghezza \* dim entry = 2^19+2=2^21=2MiB. {le entry FAT32 sono lunghe 4B}

\* Overhead: 2MiB/2GiB<0.1%. \*\*/

ES: UNIX V7: implementazione FS con struttura cartelle ad albero e allocazione indicizzata dei file basata sugli i-nodes.

**Struttura i-nodes**:

* Attributi: dim file, info creazione, ultimo accesso e ultima modifica, proprietario, gruppo, permessi, count link.
* 10 ptr diretti a blocchi disco (utile per piccoli file)
* 3 ptr indiretti: 1 singolo (punta al blocco di ptr indiretti), 1 doppio (punta a blocco di singoli) e 1 triplo (punta ai doppi).
* Max file size = max blocco indirizzabile \* dim blocco.

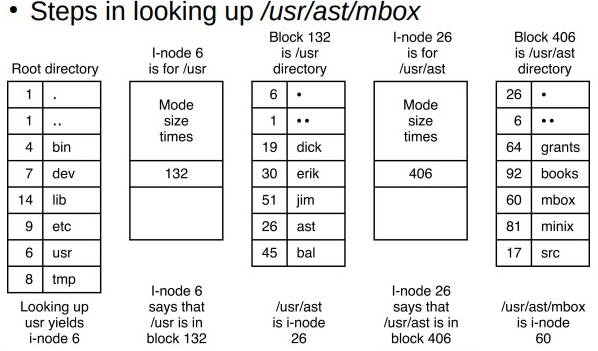
Dove max blocco indirizzabile=10+[dim blocco/dim ptr] +[dim blocco/dim ptr]^2+[dim blocco/dim ptr]^3

**Directory entry:**

numero i-node: 2B-> max 2^16 files.

Nome file: 14B->max 14 caratteri.

Entry speciali per "." e ".." per cartella curr e padre.



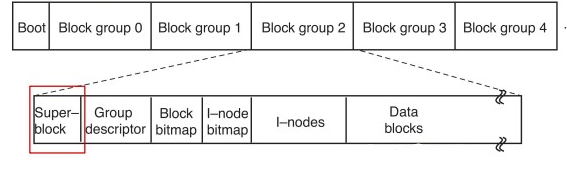
**LINUX EXT FILE SYSTEM**:creato per superare i imiti del Minx.

**Ext:** Max dim partizione: 2GB con lunghezza nome file: max 255 caratteri + Virtual File System

**Ext2:**

* allocazione FILE tramite schema indicizzato basato su i-node e preallocazione speculativa dei blocchi.
* allocazione cartelle: schema lineare
* struttura blocchi liberi: bitmap
* lunghezza file name: max 255 car.
* Limiti: max partition file = 2TiB->32TiB e max file size=16-32GiB

Formato da gruppi di blocchi: sequenza contigua di blocchi possibilmente allocato nello stesso cilindro o in quello adiacente ->+ Efficienza: poco seek, + Affidabilità:copia back-up del superblocco.



Superblocco: descrive il layout dell'intero FS. Contiene: numero totale i-nodes, disk blocks e ogni block group contiene una copia del superblocco stesso per rimpiazzarlo.

Group descriptor: descrive il gruppo, locazione di: bitmap, numero blocchi/nodi liberi, cartelle...

Block/i-nodes bitmaps: tiene traccia di quelli usati o liberi, ogni bitmap è salvata in un singolo disk block. Max bits: block size in byte\*8

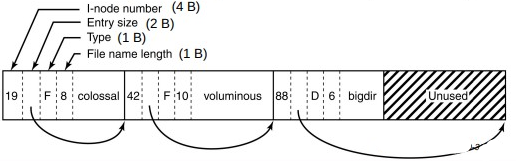
I-nodes: nodo lungo 128B e in una struttura con 12 ptr diretti (simile ad Unix v7).

Data blocks: blocchi in cui devono essere allocati file e cartelle.

ALGORITMO ALLOCAZIONE BLOCCHI:

* i-node: prova ad allocare le nuove cartelle nello stesso gruppo del padre e i file in quello della cartella. Non c'è spazio (gruppo pieno)->gruppo non pieno più vicino possibile.
* data block alocation: prova il gruppo in cui risiede il suo i-node e usa preallocazione speculativa per i file regolari. Serve spazio->guardo la bitmap.

Directory enries: organizzate come una lista lineare, ogni entry e nome file hanno dim variabile-->scansione tutta la lista e frammentazione interna.

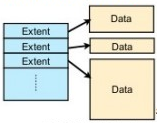
* numero i-node: numero del i-node collegato al file/sottocartella identificata dalla entry
* dimensione entry: grandezza della entry in bytes-->permette di trovare la entry della cartella successiva
* tipo: regolare, cartella, dispositivo, link...
* lunghezza nome file
* Spazio rimanente: avanzo del file name + padding

Entry rimossa->buco-->allungo la dim della entry della precedente cartella (se non ci sono precedenti riempio di 0).

**Ext 3:** estende il 2 +

* Sistema di journaling
* Allocazione diretta: lista lineare di default come ext2 + Hashed B-tree opzionale
* File più grandi: fino a 2 TiB invece di 32 GiB.
* Compatibilità: backward (monto ext2 come un ext3) e forward (monto un ext3 come ext2)

**Ext 4:** estende il 3 +

* allocazione file: schema indicizzato su i-nodes + extents
* allocazione diretta: Htree come schema di default ma lista lineare ancora disponibile.
* Miglliori performance/scalabilità/affidabilità: max 64000 sottocartelle (32000 in ext3), partizione di 1 EiB=2^60B e file fino a 16TiB. Journaling con controllo somma e possibile disattivazione.
* Compatibilità: backward ma solo parzialmente forward (solo se le nuove features del 3 non sono usate).
* **extents**: lo schema di mapping indiretto dei blocchi usato fino a ext3 non è efficiente (buono per files piccoli ma costoso). Per accedere ad alcuni blocchi devo seguire una catena di ptr indiretti->tanto seek.

Un extent è un set di blocchi logicamente contigui col file e fisicamente contigui con i dispositivi di salvataggio. Il numero di blocchi in un extent è variabile.

In ext4 l'extent è uno schema di allocazione usato come default.

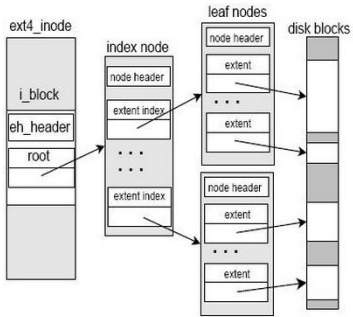
+)I/O veloce (blocchi contigui)

-)poco overhead tempo/spazio: accesso ai metadati attraverso meno blocchi.

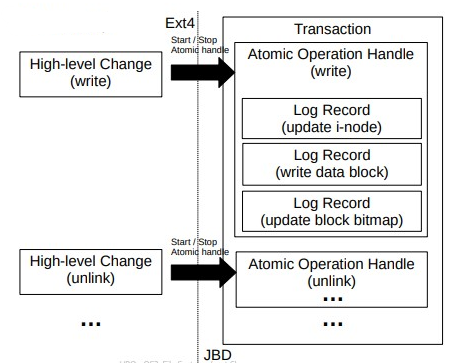
1 extent: max 128MiB di dati con blocchi di 4KiB

1 nodo: max 4 extents.

Per file larghi uso un albero extent di profondità costante: la radice salvata nella struttura i-node e punta al index node (nodo interno), ogni index node può puntare ad un altro o ad un leaf node. Gli extent sono salvati nei leaf nodes e puntano ai data blocks. Max depth=5.



**Journal block device (JBD)**: fornisce un livello kernel indipendente dal File system per maneggiare il journal. il Journal può essere salvato sullo stesso dispositivo del FS o su un altro.

Journal: file trattato come un buffer circolare in una regione contigua di un dispositivo.

JBD supporta 3 strutture principali:

* **log record**: descrive una operazione del FS a basso livello
* **atomic operation handle**: gruppo di log records che forma un cambiamento ad alto livello che deve essere fatto atomicamente.
* **transazione:** gruppo di atomic operation handle

Per efficienza il JBD non scorre le singole operaton handle ma direttamente le transazioni

Stati della transazione:

* **running**: la transazione è attiva e può accettare un nuovo atomic op handle. 1 per volta
* **locked**: transazione non ne accetta di nuovi ma ne esiste un ancora non finito (sono pasati almeno 5 secondi o non c'è spazio nel journal)
* **flush**: tutte le atomic handle sono complete, posso scrivere la transazione sul journal.
* **Commit**: tutti i log record degli atomic op. handles sono scritti su disco ma la transazione non è segnata come finita.
* **finished**: posso compiere le operazioni su disco-->quando le compio le segno come checkpointed (posso rimuoverle).

Scarto sia le incomplete che le non-checkpointed.

I cambiamenti possono coinvolgere: meta-dati del File-system (superblocchi, gruppi, nodi, bocchi bitmap...) o dati (i non meta-dati).

Modalità di journaling:

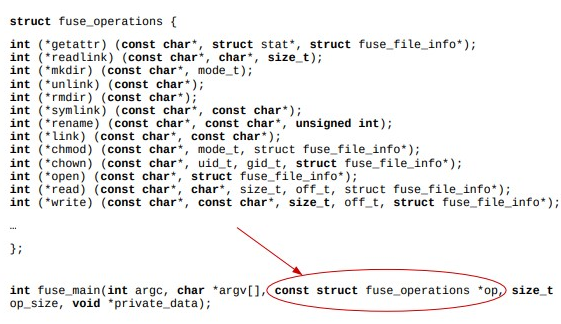
* **journal**: fa log di dati e metadati, richiede + accessi a disco ma perde meno aggiornamenti. Lento ma sicuro
* **ordered**: log solo dei metadati ma i dati vengono scritti prima. Mode default, - corruzioni.
* **Writeback**: log solo dei metadati. Non ci sono garanzie su ordine di scrittura dei file su disco. Rischioso ma veloce.

**FUSE**: File system on userspace: implementazione del file system nello user-space.

In Linux è un VFS. Il modulo kernel FUSE comunica con unn processo a liv. utente che implementa il FS nel FUSE attraverso la libreria libfuse.

VFS ridireziona le richieste del client al FS concreto (FUSE kernel module) che usa i comandi della libfuse e ridireziona il comando al programma di iplementazione che lo esegue e manda il return al client.

Il FS deve registrarsi alla libreria con il metodo fuse\_main().



+)Facile da aggiustare e fare extent, non ricompilo il kernel.

+)Non crasha con i bug

+)Molto portatile

+)Disponibile in più linguaggi di programmazione

-)Lento