File System

Per memorizzare dati a lungo termine, il sistema operativo fornisce l'astrazione del **file** attraverso un **file system**. I file sono l'unità logica di informazione creata da un processo e le informazioni contenute in un file sono persistenti.
Dal punto di vista utente, file system è il servizio fornito dal sistema operativo che permette di facilitare le operazioni sui file. Dal punto di vista del designer di sistemi, è una collezione di strutture dati e meccanismi usati dal sistema operativo per memorizzare e cercare informazioni.

1 File

1.1 Denominazione

Ogni sistema operativo ha le proprie regole per la denominazione dei file: di solito sono ammesse stringhe di 1-8 caratteri alfanumerici e alcuni caratteri speciali. Alcuni sistemi operativi sono case sensitive, altri no.
Molti SO supportano nomi a due parti, con le due parti separate da un punto (parte1.parte2): la seconda parte è detta **estensione del file**, indica una caratteristica del file. In UNIX, sono solo una convenzione, mentre su Windows hanno un significato: le estensioni indicano con quale programma il file può essere aperto.

1.2 Struttura

Tre modi comunemente usati per strutturare file:

- sequenza di byte: un file è una sequenza di byte non strutturata, il significato è imposto dai programmi a livello utente
- sequenza di record: un file è una sequenza di record a lunghezza fissa
- albero di record: un file è un albero di record, ognuno dei quali contiene un campo chiave

1.3 Tipi

La maggior parte dei SO supporta almeno due tipi di file: **file regolari** e **directory**.
I file regolari contengono dati e possono essere file di testo o binari, le directory sono file di sistema che mantengono la struttura del file system. Altri tipi di file sono di file a caratteri e file a blocchi, che servono per I/O.
I file sono divisi in varie sezioni: la prima è l'header, contenente il **numero magico** che identifica il formato.

1.4 Accesso

Tipi di accesso:

- accesso sequenziale: i byte sono letti/scritti in ordine a partire dall'inizio
- accesso diretto: i byte possono essere letti/scritti in qualsiasi ordine; il SO fornisce una funzione di seek per accedere al file in qualsiasi posizione

1.5 Attributi

Gli attributi sono le proprietà associate ad ogni file: data di creazione, dimensioni e permessi sono attributi.

1.6 Operazioni

Operazioni possibili su file:

- create: crea un file vuoto
- delete: elimina un file
- open: apre un file
- close: chiude un file
- read: legge dati dal file dalla posizione corrente
- write: scrive dati su file nella posizione corrente
- seek: cambia la posizione del puntatore
- append: aggiunge dati alla fine di un file
- get attributes: legge proprietà di un file
- set attributes: modifica proprietà di un file
- rename: rinomina file

2 Directory

2.1 Organizzazione

L'organizzazione può essere a directory singola o a gerarchia: la prima ha una directory contenente tutti i file, la seconda più directory ordinate ad albero.

2.2 Path name

Il path name è il modo per specificare il nome di un file all'interno di un albero di directory: è una sequenza di nomi di directory separate da un carattere detto **path separator**.
La directory corrente è rappresentata con **"."**, la directory genitore con **".."**.
Diversi tipi di path:

- assoluto: path dalla root directory al file, inizia con separatore e può contenere . e ..
- canonico: path diretto, non può contenere . o ..
- relativo: path relativo alla directory corrente, può contenere . e ..

2.3 Operazioni

Operazioni possibili su directory:

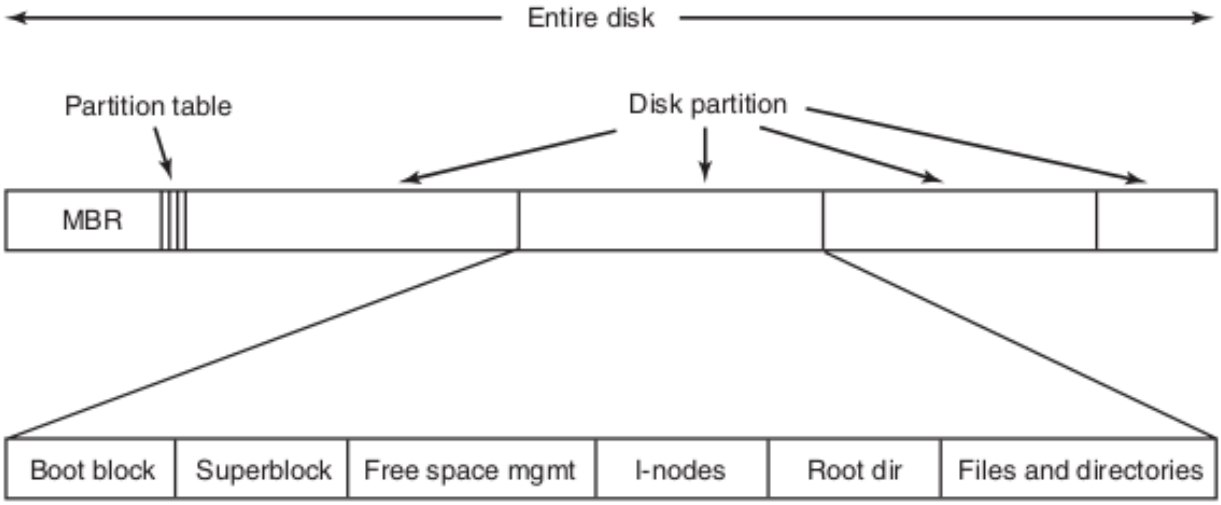
- create: crea directory vuota
- delete: elimina una directory vuota
- opendir: apre directory
- closedir: chiude directory
- readdir: legge directory
- rename: rinomina directory
- link: crea link ad un file
- unlink: rimuove link al file

Il linking è una tecnica che permette ad un file di apparire in più di una directory. Possono essere hard link o soft link (detto anche symbolic link).

3 Implementazione del file system

3.1 Layout

Una disco può essere diviso in più partizioni. Una partizione è divisa in più settori.
Il settore 0 è detto **Master Boot Record**(MBR) e contiene il boot loader del SO. Alla fine del MBR c'è la tavola di partizione.
Il **superblock** è un blocco che contiene i parametri chiave del file system ed è portato in memoria a boot time.
Free space mgmt contiene info sui blocchi liberi del file system.
I-nodes contiene info su ogni file.
Root dir è la root directory.



Molte strutture dati sono caricate e create nella memoria centrale quando il file system viene usato (montato), sono distrutte quando viene smontato. Queste strutture dati dipendono del SO. Tipicamente sono:

- mount table: contiene info su tutti i file system montati
- open file table del sistema: contiene copie del file control block di ogni file aperto
- open file table dei processi: contiene, per ogni file aperto da un processo, puntatori per ogni voce della file table del sistema
- cache della struttura delle directory
- cache dei blocchi del disco

Alla creazione di un file, il SO crea un nuovo file control block e aggiunge una nuova voce nella directory.
All'apertura di un file, il SO cerca nelle directory il file appropriato e controlla se esiste una voce nella system-wide open file table: se esiste, crea una nuova voce nella per-process file table e incrementa il counter dei file aperti, altrimenti crea una nuova voce nella system-wide open file table e pone il counter a 1. L'operazione di open ritorna un pointer all'voce appropriata nel per-process file table.
Alla chiusura di un file, l'voce nella per-process file table è rimossa e il counter decrementato: se il counter nella system-wide file table diventa 0, anche quella voce è rimossa.

3.2 Implementazione dei file

Un file è un insieme di blocchi. Esistono vari modi per allocare i file.
Nel metodo ad **allocazione contigua**, il file è memorizzato come una sequenza contigua di blocchi: l'voce nella directory contiene info sul numero di blocchi da cui è composto il file e indirizzo del primo blocco. Questo metodo è facile da implementare e garantisce maggiori performance, ma causa frammentazione esterna: i buchi lasciati da file eliminati potrebbero essere troppo piccoli per poter memorizzare nuovi file. Inoltre è necessario sapere, alla creazione del file, quanti byte questo occuperà.
Il metodo di **allocazione a extents** è simile a quello contiguo: ogni file consiste in una o più regioni di blocchi contigui detti extents. Ogni file necessita di una tavola per tener traccia degli extents allocati. Con extents di lunghezza fissa si va incontro a frammentazione interna, con quelli a lunghezza variabile a frammentazione esterna.
Nell'**allocazione a liste concatenate**, i blocchi di un file sono organizzati in una lista concatenata, l'voce nella directory contiene solo l'indirizzo del primo blocco. Mentre non causa frammentazione esterna, questo metodo non garantisce buone performance e presenta overhead, poichè ogni blocco deve riservare spazio per il puntatore al prossimo blocco (qualche byte).
L'**allocazione a cluster** accorpa blocchi contigui in cluster di dimensioni fisse e li collega. Performa meglio dell'allocazione a liste concatenate ma va sempre incontro a frammentazione interna(un cluster può essere pieno solo in parte).
L'allocazione a liste collegate può essere estesa usando una **File Allocation Table** (FAT): i puntatori ai blocchi successivi sono memorizzati nella tabella, con una voce per ogni blocco. Per velocizzare l'operazione di seek, la tabella è portata in RAM; rimane ancora possibile la perdita di dati legata al danneggiamento dei pointer.

Con l'**allocazione a indici** ogni file ha la sua tavola d'allocazione (chiamata index block), memorizzata in uno o più blocchi separati, contenenti gli indirizzi dei blocchi. L'voce del file nella directory contiene l'indirizzo dell'index block. L'index block è portato in RAM, ma solo quando il file associato è aperto. La dimensione del file è limitata dalla dimensione dell'index block. Ci sono varie estensioni dell'allocazione a indici: lo **schema a liste collegate** ha index block organizzati come una lista collegata e lo **schema a indici multilivello** ha index block che puntano ad altri index block e così via, fino a puntare ai file block. Il numero di livelli determina la dimensione massima del file. L'efficienza dipende dalle dimensioni del file e dal blocco che si vuole accedere. Nello **schema i-node** eiste un index block principale, l'i-node: le prime n voce sono puntatori diretti ai blocchi, gli altri sono puntatori indiretti, che puntano ad indici multilivello.

3.3 Implementazione delle directory

Una directory è una collezione di voce, una per ogni file contenuto nella directory. In alcuni SO, l'voce contiene anche attributi dei file, in altri sono memorizzati nel file control block.
Come gestire file con nomi lunghi? Usare campi di lunghezza fissa consuma molto spazio, quindi si usano campi di lunghezza variabile: ogni file ha un header di lunghezza fissa e un cammpo di lunghezza variabile per il nome del file. Ogni nome è terminato da un carattere speciale. Questo approccio causa frammentazione esterna; si possono invece tenere i nomi dei file in un area separata, ad esempio l'heap alla fine di una directory: ogni voce contiene un puntatore nell'heap dove può trovare il nome del file. Ciò causa frammentazione interna ma è risolvibile con la compattazione.
L'operazione fondamentale è cercare file: può essere reso efficiente tramite una hashtable. Ogni directory ha una hashtable che mappa il nome del file al puntatore dell'voce della directory dove può essere trovato. In alternativa, si può usare una struttura ad albero: ogni nodo contiene un numero variabile di chiavi e figli, e le foglie hanno tutte la stessa profondità. Di solito viene usato uno schema ibrido: se la directory contiene pochi file, viene usata una lista, altrimenti viene usato un albero.
I nomi dei file sono memorizzati in una cache: questa è ispezionata prima di ogni ricerca, quindi se contiene l' indirizzo del file, questo è locato immediatamente. Se il nome del file non è contenuto nella cache, si risale alla directory genitore finchè non viene trovata una voce nella cache.

3.4 Condivisione file

File condivisi apppaiono in directory diverse appartenenti ad utenti diversi. Per ottenere ciò si usa la tecnica del **linking**: le directory coinvolte contengono un link al file condiviso. Il linking è gestito tramite un **grafo diretto aciclico**.
Se le voce contenessero indirizzi a disco, i cambiamenti del file apparirebbero solo all'utente che lo ha modificato. La prima soluzione è quella di usare **hard linking**: i blocchi sono listati nel file control block del file condiviso (in UNIX, gli hard link ad un file puntano al suo i-node). Se però un utente rimuovesse il file, gli altri utenti avrebbero una voce che punta ad un file control block non valido, e la rimozione dei link non validi sarebbe un'operazione non pratica. Di solito, il file control block contiene un counter che tiene traccia di quante directory voce puntano al file: quando il counter raggiunge lo 0, il SO può rimuovere il file control block.
La seconda soluzione è quella di usare **symbolic linking**: il SO crea un nuovo file di tipo LINK, che contiene solo l'indirizzo al file a cui è collegato. A differenza degli hard-link, i symbolic-link non causano incremento o decremento del counter e sono trasparenti alle applicazioni. Solo l'utente che ha creato il file ha il puntatore al file control block del file condiviso, quindi il file control block

è distrutto solo quando il proprietario elimina il file.
I symbolic link possono linkare a directory (ciò può causare loop) e sono file speciali (gli hard link sono file regolari); i file simbolici possono linkare file attraverso macchine diverse, su diversi file system, ma richiedono più overhead rispetto agli hard link, dato che è necessario un ulteriore file control block per ogni symbolic link.
Alcuni programmi non riconoscono il link, mentre altri forniscono solo supporto limitato.

3.5 Mounting/unmounting

Prima di poter accedere ai file su un file system, questo deve essere montato. Montare un file system collega questo ad un **mount point** e lo rende disponibile al sistema. Di solito il mount point è una directory vuota, ma alcuni sistemi operativi permettono il mounting su directory contenenti file. Su UNIX, il **root file system** "/" è sempre montato a boot time, e ogni altro file system può essere collegato o scollegato dal root file system.
Per completare un'operazione di mounting, il SO deve verificare che il dispositivo contenga un file system valido. Se lo è, il SO memorizza nella **mount table** che il file system è stato montato. Ad esempio, su UNIX, il SO:

- controlla la validità del file system
- crea strutture dati che contengono metadati del nuovo file system (superblock)
- aggiorna la mount table con la nuova voce
- collega il nuovo file system al mount point settando flag del mount point e puntatore alla voce nella mount table

3.6 Virtual File Systems

Un **virtual file system** integra multipli file system in una singola struttura omogenea: possono essere file system memorizzati nelle partizioni del drive locale oppure file system remoti. L'idea chiave è l'astrazione, ovvero isolare gli aspetti che sono comuni a tutti i file system in uno strato che fa da ponte tra gli strati superiori e quelli inferiori.
Il VFS possiede un'interfaccia "superiore" che si interfaccia con i processi utente ed una inferiore che si interfaccia con i file system concreti (in kernel space). Le astrazioni tipicamente usate dalla VFS sono:

- superblock: descrive il file system montato
- v-node: descrive il file control block associato ad uno specifico file
- file: descrive un file aperto associato ad un processo
- directory entry: descrive una specifica voce in una directory

Inoltre, la VFS contiene strutture dati interne, quali mount table, system-wide open file table e per-process open file table. Non memorizza nulla su disco, gestisce tutto in memoria.
Ad esempio, per aprire un file data system call *open*, la VFS cerca il superblock del file system nella mount table, trova la root directory e cerca il giusto path tramite funzioni del file system concreto; una volta trovato il file corrispondente, crea un v-node che contiene puntatori alle funzioni per manipolare file, esegue la funzione di *open*, crea una nuova voce nella v-node table e nella file descriptor table, ritornando il file descriptor appena creato.

4 Gestione e ottimizzazione dei file system

4.1 Gestione dello spazio su disco

La dimensione dei blocchi è di solito determinata durante la formattazione ad alto livello. Qual è la giusta misura? Blocchi troppo grandi sono proni a frammentazione interna e rischiano di sprecare spazio su disco. Blocchi troppo piccoli richiedono molto overhead e leggere file composti da molti blocchi richiede maggior tempo. La dimensione dei blocchi influenza il data rate (bytes trasferiti al secondo) e spazio usato/allocato. Il tempo d'accesso è dominato dal tempo di seek e dal delay rotazionale; più grandi sono i blocchi, maggiore sarà il data rate, ma minore sarà l'efficienza in spazio (viceversa per blocchi piccoli).
Per tenere traccia dei blocchi liberi sono usati due approcci: liste collegate e bitmap. Il primo consiste in una lista contenente blocchi contenenti puntatori a blocchi liberi (e nodo successivo della lista). Il secondo consiste in una mappa di bit dove ogni bit corrisponde ad un blocco su disco. La bitmap richiede spazio costante indipendentemente del numero di blocchi liberi, mentre la lista è più efficiente quando il disco è quasi pieno. I blocchi liberi di solito sono collocati uno di fianco all'altro, quindi la lista potrebbe tenere conto solo della grandezza di questa fila continua di blocchi liberi.
Le strutture dati per la gestione dello spazio libero sono di solito memorizzate su disco ma portate in memoria per ridurre le operazioni di I/O. Con lo schema a lista si può tenere un solo nodo in memoria, fino a quando non si esauriscono i puntatori a blocchi liberi: questo metodo può però risultare in molte operazioni I/O nel caso il nodo in memoria sia quasi vuoto. Per limitare il problema è possibile dividere un nodo pieno per evitare di avere nodi quasi vuoti in memoria. Per le bitmap, è possibile memorizzare in memoria solo alcuni blocchi al posto dell'intera bitmap.
Le **disk quotas** sono un meccanismo per evitare che gli utenti di un sistema multiutente consumino troppo spazio. Allo scopo, eiste una *quota table* con voce per ogni utente. Per ogni file aperto, il corrispondente file control block nella system-wide open file table contiene l'ID del proprietario e un puntatore alla quota. Ogni volta che un blocco viene aggiunto ad un file aperto, il numero di blocchi totali per l'utente è incrementato e vengono controllati i limiti. Se il soft limit viene superato, viene visualizzato un avviso e il numero di avvisi rimanenti è decrementato (periodo di grazia).

4.2 Backups

Creare backup è un'operazione lunga e costosa in termini di spazio. Di solito non è necessario fare un backup dell' intero sistema, ma solo dei file veramente importanti e che sono stati modificati dall'ultimo backup.
ll **backup differenziale** consiste nel fare backup dei soli file modificati dall'ultimo backup totale: rende facile il recupero dei file ma richiede più spazio. Il **backup incrementale** consiste nel fare backup delle modifiche intercorse dall'ultimo backup (totale o parziale): occupa meno spazio ma rende il recupero dei file più complesso e meno affidabile.
Comprimere i dati permette di risparmiare spazio ma un singolo errore può invalidare un intero backup, quindi questa operazione deve essere considerata con precauzione.
Creare backup quando un sistema è online può causare inconsistenze, quindi sono stati creati algoritmi per fare snapshot dello stato del file system: quando uno snapshot è creato, ogni seguente modifica a file e directory è applicata alle nuove copie dei blocchi originali.
Un disco può essere copiato su un disco di backup in due modi: dump fisico e dump logico. Il **dump**

fisico inizia al primo blocco del disco (o partizione) e procede sequenzialmente: è indipendente dal file system utilizzato, è molto veloce, non modifica metadati e può copiare file system non montati. In questo modo, però, sono copiati anche i blocchi vuoti: per evitarlo, il programma di backup deve sapere a priori come il file system è implementato, e questo processo rompe la mappatura uno ad uno dei blocchi tra dischi. Inoltre, si rischia di copiare blocchi corrotti e file appartenenti al SO che non dovrebbero essere copiati.
Il **dump logico** inizia da una o più directory specificate e copia ricorsivamente tutti i file e directory presenti; deve includere tutte le directory contenute nel path di una directory o file modificato, per rendere possibile il recupero del suddetto file. Il dump logico è più lento del dump fisico, ma è più preciso.
Un algoritmo di dump basato su UNIX mantiene una bitmap indicizzata al numero di i-node ed opera in 4 fasi:

1. Tutte le velocizzare (a partire dalla directory specificata) sono esaminate, e per ogni file modificato il suo i-node è segnato nella bitmap
2. L'albero delle directory è ripercorso ricorsivamente, tenendo conto delle directory che non sono da copiare
3. Gli i-node della bitmap sono scansionati e le directory da copiare sono copiate
4. Gli i-node della bitmap sono scansionati e i file da copiare sono copiati

Con questo metodo, il recupero del file system è molto semplice: prima sono recuperate le directory e poi i file. Ci sono però dei problemi: la lista dei blocchi liberi non viene copiata quindi dev'essere ricostruita dopo ogni dump, i file linkati devono essere copiati una volta sola, i file speciali e i blocchi vuoti contenuti negli sparse file (file con "buchi") non devono essere copiati.

4.3 Consistenza

Per aumentare efficienza, il sistema mantiene i blocchi modificati in memoria e li scrive su disco solo successivamente. Un file system può ritrovarsi in uno stato inconsistente a causa di crash o spegnimento improvviso del sistema.
I SO hanno programmi per controllare la consistenza del file system; in UNIX è *fsck*: *fsck* controlla sia la consistenza del blocchi che quella dei file.
Per controllare la consistenza dei blocchi, *fsck* usa due tabelle, una per i blocchi in uso e una per blocchi liberi. Per prima cosa, *fsck* legge tutti gli i-node, legge gli indirizzi allocati per ogni file e incrementa il contatore corrispondente nella tavola dei blocchi in uso, poi esamina la tabella dei blocchi liberi per trovare tutti i blocchi non in uso. Un file system è consistente a livello di blocchi se ogni blocco ha il contatore corrispondente settato a 1 in una tavola e 0 nell'altra. Si possono verificare diversi tipi di incosistenza:

- blocco con contatore a 0 in entrambe le tabelle, *fsck* lo aggiunge alla tabella dei blocchi liberi
- blocco con contatore a 1 in entrambe le tabelle, *fsck* può rimuoverlo da uno delle dei tabelle
- blocco con contatore > 1 nella tabella dei blocchi liberi, *fsck* ricostruisce la tabella dei blocchi liberi per eliminare l'inconsistenza

- blocco con contatore > 1 nella tabella dei blocchi in uso, *fsck* alloca un blocco libero, copia il contenuto del blocco duplicato in quel blocco e inserisce la copia in uno dei file

fsck controlla la consistenza dei file, usando una singola tabella che tiene traccia di quante volte un file appare nell'albero di una directory (una voce per ogni i-node). Un file system è consistente a livello file se per ogni file il contatore correlato presente nella tabella combacia con il contatore dei link nell'i-node corrispondente. Possono presentarsi due tipi di errori:

- contatore dei link di un i-node ¿ del contatore nella tabelle dei file in uso: quando tutti i link ad un i-node sono rimossi dalle directory, il contatore sarà ¿ 0 e l'i-node non verrà rimosso, quindi *fsck* setta il contatore dei link con il valore corretto presente nella sua tabella

- contatore dei link di un i-node ¡ del contatore nella tabelle dei file in uso: quando il contatore dei link raggiunge lo 0, il file system lo marca come non in uso, ma alcuni file punteranno ancora a tale i-node, quindi *fsck* setta il contatore dei link con il valore corretto presente nella sua tabella

Sono possibili altri controlli: controllo dello stato degli i-node (se è corrotto, i blocchi associati sono rilasciati e l'i-node eliminato), directory (segnala directory con numero di file sospetto o eccessivi permessi), superblock e bad-blocks (blocchi che contengono bad-sectors).

4.4 Journaling file systems

L'obiettivo principale è garantire robustezza in caso di problemi, mantenendo un log di ciò che il file system sta per fare prima di farlo (**write-ahead log**), in modo da poter riprendere da quel punto in caso di crash del sistema. Per rendere il jounaling funzionale, le operazioni loggate devono essere idempotenti, ovvero possono essere ripetute senza causare danni. Per garantire maggiore efficienza, vengono usati **log circolari**. Ogni voce non vuota rappresenta una transazione da svolgere; quando tutte le operazioni sono state svolte, la voce è marcata come completata (checkpoint). Il recupero inizia dalla transazione seguente l'ultimo checkpoint e procede finchè non viene trovata una voce vuota.
Il jounaling introduce problemi di efficienza, dato le scritture su disco sono raddoppiate. Per ovviare a questo problema, il log è allocato sequenzialmente (meno opeazioni di seek) e contiene solo metadati (non i dati dei file). Programmi come *fsck* servono anche in un file system dotato di journaling, dato che permettono di ricoverarsi da situazioni inaspettate e forniscono ulteriori controlli.

4.5 Performance

Il **block caching** è una tecnica utilizzata per ridurre l'accesso a disco. Una **block cache** è un insieme di blocchi che logicamente appartiene al disco ma tenuto in memoria per motivi di efficienza: per ogni operazione di lettura/scrittura, viene prima controllata la presenza del blocco nella cache; se non è presente, viene letto da disco e aggiunto alla cache.
Per verificare velocemente se un blocco è contenuto in cache, si usa una hashtable. Per selezionare una vittima dalla cache, sono usati gli stessi algoritmi impiegati nella paginazione: nel caso di politica LRU, la vittima è il blocco acceduto meno recentemente. Questa politica è implementata tramite lista bidirezionale, con i blocchi usati meno di recente di fronte e quelli usati recentemente nel retro. Nel caso di crash, però, le modifiche in cache vanno perdute, lasciando il file system in

uno stato inconsistente: è quindi assegnata priorità maggiore a blocchi critici (file control block, tabella dei blocchi liberi...), i quali vengono scritti immediatamente su disco. In questo modo, potrebbe verificarsi la situazione in cui i blocchi di dati non sono frequentemente scritti su disco. In UNIX, la soluzione è usare **write-back caches**: quando un blocco in cache è modificato, è marcato come "dirty" e il programma *update* scrive periodicamente i blocchi "dirty" su disco. Tale sistema è poco affidabile e complesso da implementare. In passato, venivano usate **write-through caches**, dove i blocchi modificati venivano subito scritti su disco, a discapito dell'efficienza.
Con il metodo **block read-ahead**, i blocchi sono portati in cache prima del loro utilizzo per incrementare l'hit-rate della cache: quando il blocco di un file viene letto, gli altri blocchi corrispondenti al file sono portati nella cache. Questa strategia funziona perfettamente quando i file sono letti sequenzialmente, ma è dannosa quando sono letti randomicamente, quindi il SO tiene traccia dei pattern d'accesso di ogni file e decide se è meglio usare la suddetta strategia o meno.
Esistono varie tecniche per ridurre il movimento del braccio del disco. La prima consiste nell'allocare blocchi che sono accceduti sequenzialmente in modo che siano vicini tra loro. L'operazione di allocazione dei blocchi liberi per un file è veloce se il SO tiene traccia dei blocchi in uso e non tramite una bitmap, ma lenta se usa una lista. Può essere possibile allocare un gruppo di blocchi quando ne è richiesto uno solo, e rilasciarli nel caso non servano.
La seconda consiste nell'allocare gli i-node (nei sistemi che li posseggono) a metà del disco, invece che all'inizio; oppure dividere il disco in *cylinder groups*, ognuno con i suoi i-node, blocchi dati e tabella dei blocchi liberi: quando bisogna allocare un nuovo blocco, si cerca di trovarne uno disponibile nello stesso cylinder group in cui è contenuto l'i-node.