Implementazione di una directory

Una directory ha le stesse esigenze dei file in termini di nome, metadati, contenuto ecc..
Si necessita in particolare di avere un modo per implementare l'astrazione della directory, ad esempio si può pensare ad un **elenco dinamico** il cui generico elemento deve implementare un file, ma come è fatto il generico record di questo elenco dinamico ?

Di certo si necessita di memorizzare un nome per identificare il file, mentre il resto del record dipende da come viene implementato il file stesso.

Ad esempio nel caso di utilizzo di FAT si necessita di memorizzare i metadati ed il primo blocco a cui fare riferimento nella FAT.

Se il caso d'uso prevede un i-Node allora si necessita di mantenere, oltre al nome del file, l'i-number che farà riferimento ad un preciso i-Node che contiene i metadati del file.

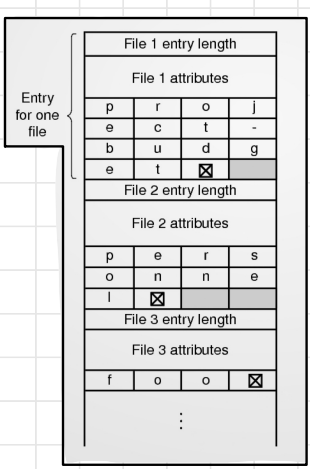
- **Primo modo :**

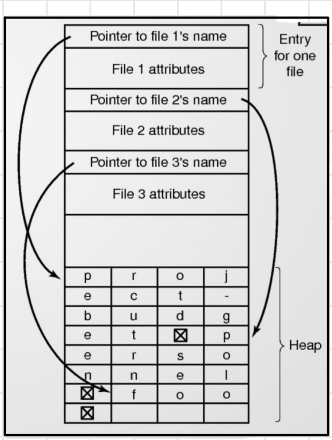
A prescindere da come si implementa il file si possono avere delle strategie differenti per organizzare la generica voce nell'elenco della directory. Si può pensare di tenere nella generica voce dell'elenco un campo "**entry_lenght**" che determina quanto è lunga l'intera entry del file (dell'intera voce). Si potrebbe pure usare questo campo per navigare le varie voci della directory, infatti è nota la lunghezza del salto per saltare alla entry del file successivo.
Si ha poi un campo a dimensione fissa che rappresenta gli attributi (metadati) del file stesso. Segue un altro campo di lunghezza variabile che rappresenta il nome del File delimitato da uno speciale carattere di terminazione.
Il nome del file è un campo dinamico e può continuare a crescere fino ad una certa soglia che se non occupata dal nome del file viene comunque allocata per mantenere un **allineamento** in memoria, indotto dalla dimensione fissa delle varie entry dei file.

- **Secondo modo :**

Si potrebbe organizzare la generica entry nell'elenco tramite un campo di dimensione fissa costituito dai metadati del file stesso ed un puntatore ad una area dinamica dell'elenco, ovvero un **heap** . L' heap così descritto contiene i nomi dei file, separati da un carattere di terminazione. Il riutilizzo delle entry diventa molto semplice in quanto non si creano buchi tra le varie entry man mano che vengono eliminati dei file e viene facilitata la compattazione. L'heap potrebbe essere un oggetto problematico in quanto necessita di allocazione contigua ma dato che sono stati rimossi i campi di lunghezza fissa, tutto sommato presenta una dimensione potenziale piccola.

Entrambi i modelli sono rappresentativi delle stesse informazioni.





Prestazioni

Si potrebbe pensare di ricorrere ad una ricerca lineare per navigare una directory, il che risulta dispendioso e lento.
Si necessita di rendere il più efficiente possibile la ricerca di un file all'interno di una directory :

- **Tabella hash**

Si pensa di implementare la ricerca tramite tabelle hash ad indirizzamento aperto. In ogni directory viene implementata una tabella hash di dimensione n e per inserire un file nella directory viene fatto l'hash del nome del file in un valore nel range da 0 a n-1. Le voci dei file seguono la tabella hash e se una posizione risulta già utilizzata viene generata un'altra posizione costruendo così una lista concatenata di posizioni con lo stesso valore di hash. La ricerca del file segue appunto questa lista ed molto rapida, tuttavia questo approccio necessita di un'amministrazione più complessa e viene usato solo in caso di directory molto grandi.

- **Cache**

Un altro approccio per effettuare ricerche in grandi directory prevede di mettere in cache il risultato delle ricerche. Prima di avviare una ricerca si verifica se il nome del file è già presente in cache. Questo approccio è particolarmente efficace in uno scenario nel quale la maggior parte delle ricerche viene fatta su un numero piccolo di file.

Condivisione di un file nel filesystem

Quando più utenti vogliono accedere allo stesso file si necessita di gestire l'accesso concorrente. Si potrebbe pensare di creare un riferimento al file all'interno della directory degli utenti che vi vogliono accedere.

- **Soft-link (simbolic-link)**

Quando viene referenziato un file su una determinata directory da più utenti, viene creato un oggetto speciale simbolico chiamato **simbolic link** nella directory che ha richiesto il file, con un nome ed un contenuto. L'entità appena creata viene vista in modo particolare dal sistema operativo. Il contenuto del simbolic link è il path dell'oggetto richiesto. Il SO esegue il follow link quando guarda al contenuto del simbolic link, ovvero segue il percorso specificato ed esegue l'operazione sulla destinazione specificata dal link simbolico, ovvero sul file originale.

Se il file viene spostato dalla directory, il contenuto del simbolic link non viene aggiornato e per tanto il collegamento presente nella directory potrebbe non essere più valido. In particolare se viene rimosso il file originale, il contenuto del simbolic-link diventa inconsistente e l'apertura provoca un errore.

L'accesso al file tramite simbolic link viene comunque gestito in maniera coerente con i permessi che si hanno sul file. Questo tipo di gestione della condivisione diviene pesante perchè nonostante la chiamata di open( ) sia solo una, si necessita di fare following di tutti i gli oggetti soft-link, i quali possono puntare ad un oggetto sia di tipo file ma anche di tipo directory, quindi ancora navigabili.

- **Hard-link (link fisico)**

Quando viene referenziato il file da più utenti si necessita di creare un secondo riferimento nelle altre directory. Nel caso in cui il file è implementato con i-Node viene creato nella seconda directory un riferimento, detto **hard-link,** allo stesso i-Node e questo approccio non crea altri oggetti, quindi ha un minore overhead.

Nel caso in cui il file referenziato viene rimosso, l'hard-link all'i-Node non viene rimosso, ma continua a rimanere allocato se ci sono altre directory che vi fanno riferimento. Nell'i-Node è riportato un **contatore** che tiene conto del numero di directory che hanno un hard-link a quel file e questo contatore viene decrementato ogni volta che una directory che lo contiene, rimuove il file referenziato. L'i-Node viene deallocato nel caso in cui non vi siano più directory, da qualche parte, che lo contengono.

Problema con gli hard-link

Si possono anche creare più hard-link (riferimenti) allo stesso oggetto nella stessa directory purchè essi abbiano nomi differenti, ed essi risulteranno indistinguibili senza possibilità di capire chi è il riferimento originale. Questo può creare problemi se si creano degli hard link ad oggetti directory, in particolare se una sotto-directory creasse un riferimento alla directory superiore, il riferimento stesso conterrebbe la sotto-diretory con un ulteriore riferimento alla directory superiore ecc.. creando un loop. Se l'algoritmo di navigazione non riconosce questi loop indotti dai cicli creati, esso stesso andrà in loop. Questo problema si presenta perchè la utility di sistema non riesce a distinguere il riferimento originale alla directory da quelli creati dopo, problema che non si manifesterebbe con un soft-link in quanto il SO riuscirebbe facilmente a distinguere i riferimenti creati dopo. Nel caso di backup con soft-link viene ricopiato l'oggetto in se senza attraversarlo in modo tale da ricostruire coerentemente la gerarchia senza cicli.
Con gli hard link non è possibile creare i **cross-file system link** ovvero hard link allo stesso oggetto tra file system differenti, poichè gli indirizzi fisici punterebbero ad oggetti differenti poichè in due spazi differenti.

Gestione dei blocchi liberi

Si deve tenere conto di una gestione di blocchi di disco liberi ed allocati in maniera efficiente :

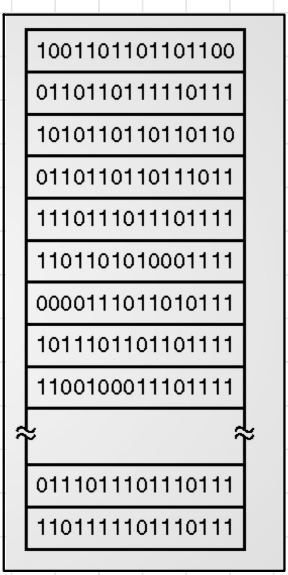
- **Bitmap**

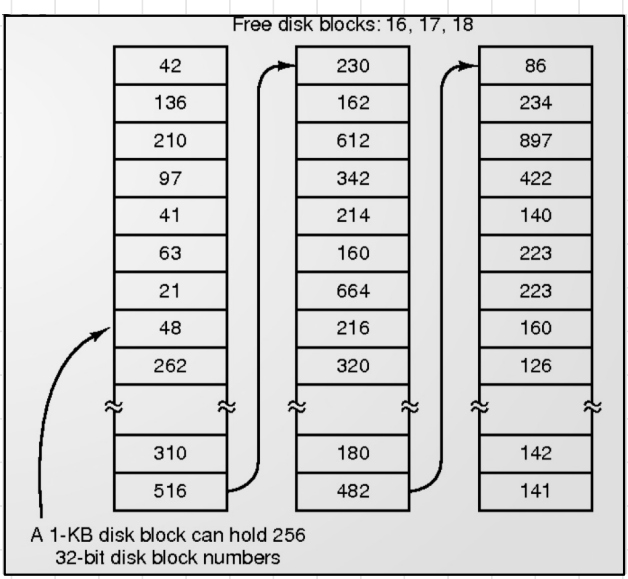
Questo approccio fa uso di una bitmap come struttura dati. La bitmap risiede sul disco ed ha un bit per ogni blocco del disco, in particolare un blocco libero è mappato in un bit a zero ed uno allocato in un bit ad 1. La dimensione della bitmap è statica ed indipendente dallo stato di allocazione del file system.

- **Liste concatenate (free-list)**

Un approccio simile a quello adottato nella paginazione prevede una lista dove il generico elemento è un blocco libero. Nel caso in cui il disco sia formattato la lista conterrà tutti i blocchi del disco, essendo tutti vuoti, mentre all'aumentare del numero di blocchi allocati, la dimensione della lista diminuisce. La dimensione della lista dipende quindi dallo stato di allocazione del file system ed occupa meno spazio della bitmap solo se il disco è quasi pieno.

Si potrebbe pensare di compattare ulteriormente la free-list se i blocchi liberi tendono ad essere molti e consecutivi, infatti in questo scenario si tiene traccia di sequenze di blocchi e non di blocchi singoli.





Controlli di consistenza

A seguito di una caduta di tensione o di un crash di sistema la logica dell'implementazione dell'astrazione deve poter garantire la consistenza dei dati. Lo stato di inconsistenza è dato dall'interruzione dell'esecuzione di un'istruzione, in generale chiusure anomale del file system. Apposite **utility** di sistema si occupano di fare dei controlli di consistenza dei dati, ad esempio un controllo potrebbe riguardare lo **stato di allocazione** dei blocchi.

- **1) controllo : vettori dei blocchi**

Si sfruttano due vettori di interi per implementare questi controlli, uno per i blocchi allocati ed uno per i blocchi liberi. Questi vettori sono inizialmente sono tutti a zero ed ogni volta che si trova un i-Node che fa riferimento ad un blocco, esso sarà allocato e quindi incremento di uno la posizione del vettore degli occupati relativa a quel blocco.
Si confrontano poi gli stati dei vettori con le strutture dati (liste o bitmap) che tengono conto dei blocchi liberi o occupati. Il risultato atteso è che tutte le posizioni dei vettori contengano zero o uno e che siano uno il complementare dell'altro, in quanto un blocco o è libero o è allocato.
Se una posizione contiene 0 in entrambi i vettori significa che quel blocco è sia libero, sia allocato e questo è un problema marginale. Il problema più grave per l'integrità dei dati è rappresentato da una posizione che vale 2 ad esempio, significherebbe che lo stesso blocco è allocato per due file (due i-Node vi fanno riferimento). La soluzione in questo caso è duplicare il blocco in condivisione involontaria e segnalare all'utente l'anomalia.

- **2) controllo : riferimenti negli i-Node**

Un altro controllo riguarda invece gli i-Node stessi, i quali al loro interno tengono un contatore di directory che vi fanno riferimento, ovvero che contengono il file a cui si riferisce l'i-Node (hard-link). Si fa una scansione di tutti gli i-Node e si incrementa una posizione di un vettore ogni volta che si trova un riferimento all'i-Node e alla fine si valuta la coerenza con il contatore dei riferimenti all'interno dell'i-Node stesso, per valutare se c'è un riscontro di dati.

- **3) Journaling**

Il journal è una struttura dati che si trova su disco insieme al file system. L'idea di base è quella di tenere in un file di log ciò che il il file system sta per fare, prima ancora che lo effettui, in modo tale che, in caso di crash durante l'esecuzione di un'operazione, il sistema stesso sia in grado di minimizzare i danni. Si tiene conto delle operazioni elementari necessarie allo svolgimento delle macro-operazioni da compiere nel journal. Vengono quindi scritte delle voci di log per ogni micro-operazione e solo dopo che le operazioni descritte sono state effettuate e concluse con successo vengono eliminate dal file di log.

In caso di crash le operazioni interrotte sono ancora presenti nel journal. La strategia prevede che vengano reiterate tutte le operazioni nel journal, quindi al più reiterano le istruzioni sui metadati che sono state interrotte. Le operazioni devono essere necessariamente **idempotenti** affinché il journal funzioni, ovvero se reiterate non vanno a provocare danni. Questo è fondamentale per garantire l'integrità del file system in caso di crash durante l'esecuzione di un'operazione. Se le operazioni sul journal non fossero idempotenti, potrebbero causare danni al file system se venissero ripetute.

Avere un incongruenza sui metadati, quindi sulle strutture che ne tengono conto, potrebbe provocare grossi danni ai file di tutte le directory, ad esempio a seguito di incongruenze sui dati presenti nella FAT si potrebbero modificare blocchi relativi ad altri file. Per questo motivo mantenere l'integrità dei metadati è più importante del mantenere l'integrità dei file stessi, in quanto danni a questi ultimi sarebbero localizzati ai singoli file danneggiati.

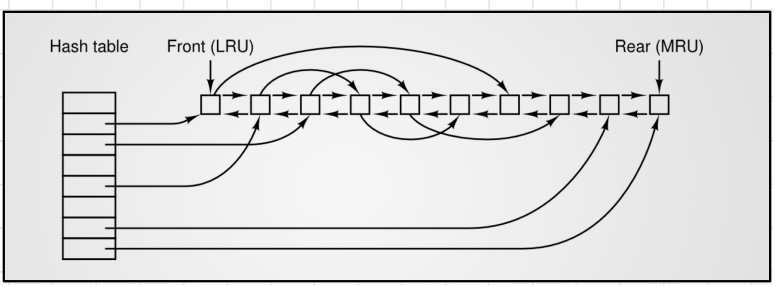
Cache del disco (buffer cache)

L'accesso al disco è molto più lento rispetto alla memoria e come conseguenza di ciò, il file-system viene progettato tenendo conto di possibili ottimizzazioni per migliorare le prestazioni. A tal proposito si fa uso di una cache del disco (buffer cache) implementata via software. Questa cache è una raccolta di blocchi appartenenti logicamente al disco, ma che sono tenuti in RAM per ragioni di prestazioni.

La dimensione allocata per la cache del disco deve essere limitata per non causare sofferenza ai processi ed allo stesso tempo si necessita di valutare algoritmi per la gestione della cache del disco.

Struttura basata su Chained-hash table

La cache può contenere anche molti blocchi e scorrerli con una scansione lineare sarebbe poco efficiente.
Si usa una tabella hash per organizzare la struttura della cache del disco, per gestire l'accesso efficiente ai blocchi di dati. Quando viene richiesto l'accesso a un blocco, un metodo abituale consiste nel generare un valore di hash a partire dal **numero del blocco** e dall'**indirizzo del disco** e cercare il risultato nella tabella hash. Tutti i blocchi con lo stesso valore hash sono collegati in una lista concatenata in modo che si possa seguire la catena di collisioni. Se il blocco di dati è presente, viene restituito immediatamente al processo richiedente, altrimenti il sistema cerca il blocco di dati sul disco e lo carica in cache, aggiornando la tabella hash di conseguenza per ridurre i tempi di accesso alle successive letture di quel dato.



LRU modificato

Quando un blocco deve essere caricato nella buffer cache ma questa risulta piena, si necessita di selezionare un blocco da rimuovere, ed eventualmente da aggiornare sul disco se modificato. Questa situazione è analoga alla paginazione e di fatto sono applicabili tutti gli algoritmi di sostituzione già visti. L'algortimo di sostituzione che più si avvicina all'ottimo è LRU, infatti il blocco referenziato meno recentemente viene scartato e sostituito da un altro blocco richiesto al disco, che viene inserito nella posizione MRU (most-recently used), in quanto appena referenziato.

Tuttavia LRU può presentare problemi di inconsistenza del file system legati a scenari si crash improvvisi del sistema. La cache in questione può lavorare sia in lettura che in scrittura dei blocchi del disco, ovvero può essere modificato il contenuto dei frame della cache del disco. Le cache in tal senso possono lavorare in modo sincrono o asincrono.

La scrittura sincrona garantisce l'integrità dei dati in caso di guasti del sistema o di perdita di alimentazione, ma può ridurre le prestazioni complessive del sistema a causa del tempo aggiuntivo necessario per scrivere i dati su entrambi i dispositivi. Lavorando in modo asincrono, si pensa invece di **ritardare l'aggiornamento** del contenuto sul disco, in modo da guadagnarci sul numero di accessi fatti a seguito di sovrascritture continue dei blocchi in istanti di tempo molto vicini, poiché si fa solo l'ultima scrittura. Ritardare le scritture tuttavia è controproducente in caso di crash del sistema, soprattuto se riguarda i metadati.

La modifica dell'LRU riguarda una scrittura dilazionata su disco in caso di dati e vengono sincronizzati in maniera immediata blocchi che riguardano **metadati** . Nonostante la misura atta a mantenere l'integrità del file-system non è comunque opportuno tenere a lungo i blocchi di dati nella cache prima di scriverli.

Free-behind & Read-ahead

La tecnica di Free-behind è utilizzata nella gestione della cache del disco per rimuovere dalla cache un blocco di dati che è stato appena scritto e che non è probabile che venga riletto subito dopo. Quando un blocco di dati viene scritto nella cache del disco, viene aggiunto alla lista LRU, che tiene traccia dell'ordine di utilizzo dei blocchi di dati nella cache. Tuttavia, invece di rimuovere il blocco meno utilizzato dalla lista LRU, la tecnica di Free-behind prevede, in maniera contro intuitiva, di **rimuovere il blocco appena scritto**, poiché si ritiene che non ci siano probabilmente altre richieste di lettura per quel blocco.

La tecnica di Read-ahead, invece, prevede di **anticipare la lettura dei dati** successivi in modo da migliorare le prestazioni dell'I/O. In pratica, quando viene eseguita una lettura di un blocco di dati, il sistema prevede che potrebbero essere richiesti i dati successivi e quindi carica in cache anche questi dati. Il caricamento di queste ulteriori informazioni sul buffer-cache non comporta un aumento nel tempo di accesso al disco in quanto è un costo che è già stato pagato.