Sistemi Operativi 1

anno 2008

Progetto numero 3

ROMFS e VFS

di

Danilo Tomasoni

Denisa Ziu

Martino Salvetti

Indice

1. Caratteristiche generali di un file system in ambienti Linux

Un file system è il mezzo logico con cui un sistema operativo memorizza e recupera i dati da un dispositivo di archiviazione come ad esempio un hard disk o un CD-ROM.
Anche se può sembrare banale è bene esprimere sin dall'inizio i concetti generali di un file system e partire quindi dalle operazioni fondamentali da esso implementato:
-creazione e cancellazione dei file
-apertura dei file per la lettura e scrittura
-chiusura di file
-creazione di directory destinate a contenere gruppi di file
-elencazione del contenuto di una directory
-eliminazione di file da una directory.
Il file system consiste di due parti distinte: un insieme di file, ciascuno dei quali contenente dati correlati, e una struttura della directory, che organizza tutti i file nel sistema e fornisce le informazioni relative.
Un file è una sequenza ordinata di elementi, dove un elemento può essere una parola (word) in linguaggio macchina, un carattere o un bit, in relazione all'implementazione. Un programma, o un utente, può creare o modificare un file solo tramite l'utilizzo del file system. Per i sistemi Unix un file, a livello del file system, in genere è privo di formato. Tutta la formattazione viene eseguita da moduli di livello superiore o da programmi forniti dall'utente. Dal punto di vista degli utenti un file ha un unico nome, e tale nome è simbolico. Una directory è uno speciale file gestito dal file system, contenente un elenco di varie voci, chiamate entry. Per un utente, un'entry ha l'aspetto di un file, a essa infatti si può accedere tramite il nome simbolico, che è il nome di file dell'utente. Il nome di un'entry deve essere unico solo all'interno della direcory in cui è presente; pertanto, in Linux, lo spazio dei nomi di file si riferisce a una sola directory. In realtà, ciascuna entry è un puntatore, che può essere di due tipi: l'entry può puntare direttamente a un file (che può essere anche una directory), oppure puntare ad un'altra entry, situata nella stessa directory o in un'altra directory. Una entry che punta ad una entry di un'altra directory prende il nome di collegamento.

1.1 Directory con struttura a grafo aciclico

Il file system può essere molto ampio. Molti sistemi registrano milioni di file in terabyte di spazio dei dischi. Per gestire tutti questi dati è necessaria un'adeguata organizzazione di essi. I sistemi operativi Linux mantengono una struttura della directory a grafo aciclico. Un grafo aciclico permette alle directory di avere sottodirectory e file condivisi. Il file e la directory possono essere in due directory diverse. Ciò significa che se un file è condiviso esiste un'unica copia del file. La condivisione è di particolare importanza se applicata alle sottodirectory; un nuovo file apprare automaticamente in tutte le sottodirectory condivise.
I file e le sottodirectory condivisi vengono realizzati tramite i collegamenti (link). Esistono due tipi di collegamenti. Il primo tipo si riferisce agli **hard link,** che sono di fatto una copia

di una voce di directory, hanno nomi diversi ma puntano allo stesso inode e quindi condividono esattamente lo stesso dato (oltre agli stessi permessi, data di modifica, owner ecc.). Dal momento che originale e link sono indistinguibili e condividono lo stesso inode, non si possono fare hard link fra file system diversi. Pur essendo la gestione dei link comunque molto rapida da parte del kernel, gli hard link sono leggermente più "economici" in quanto risparmiano più spazio su disco (stesso i-node, stessi dati) e richiedono una lettura in meno sul file system.
Il secondo tipo fa riferimento ai **symbolic link**, che sono dei piccoli file che contengono un puntamento ad altri file o directory. Questi file hanno i-node autonomo e possono puntare a file di altri file system (sia locali, che di rete). Si possono facilmente visualizzare con un normale ls -l e se viene cancellato o spostato il file a cui puntano rimangono "stale": continuano ad esistere ma puntano a qualcosa che non esiste.

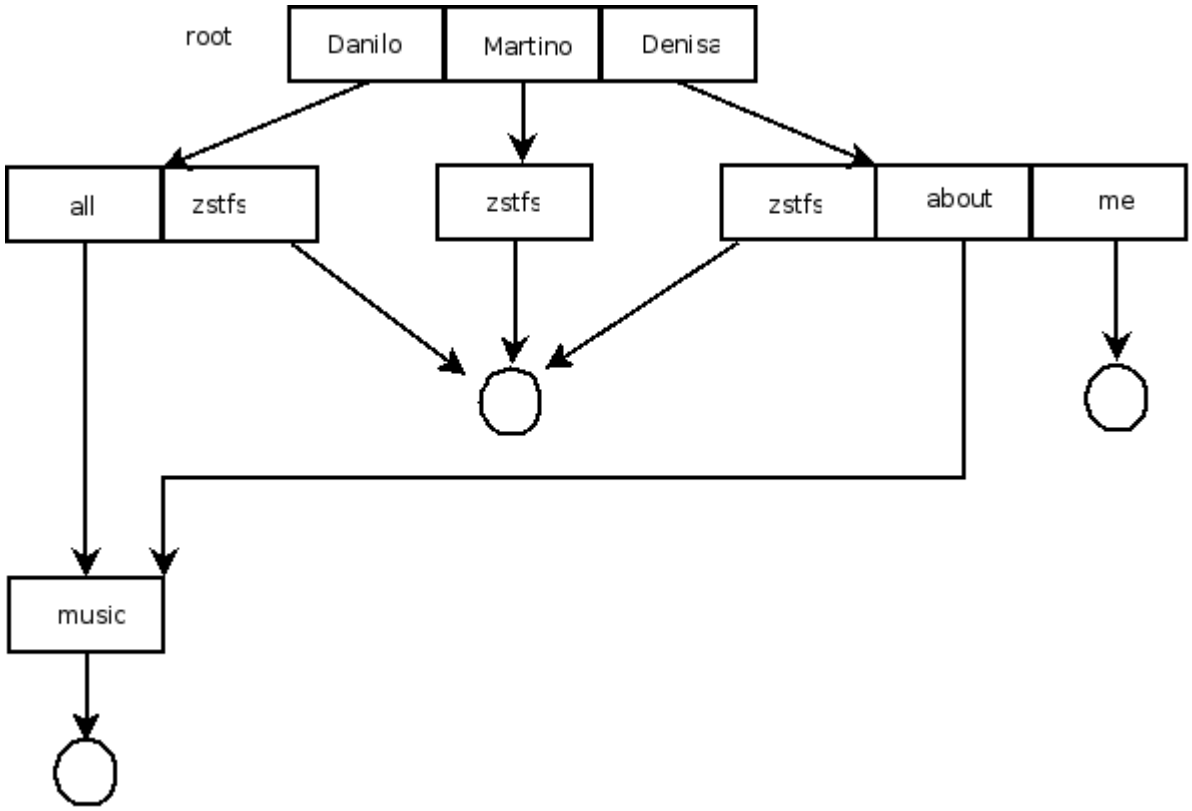


Figura 1.1 Esempio di struttura della directory a grafo aciclico

2. La relazione tra il kernel di Linux e il file system: il Virtual File System

Il compito di organizzare le strutture di controllo del file system non è di competenza delle applicazioni e nemmeno compito specifico del kernel. Il sottosistema dei file del kernel di Linux ha un'idea astratta della natura di un file system. Il kernel di Linux ignora completamente le specifiche del file system nativo di Linux ext2 o di reiserfs: tutti i file system sono trattati allo stesso modo. L'interazione tra il kernel e i vari tipi di file system viene gestito tramite il Virtual File System. Questo livello di astrazione conesnte di implementare ogni operazione di manipolazione dei file come chiamatate di sistema del kernel che agiscono su oggetti istanziati della struttura di controllo, e inoltre consente l'allocazione e la deallocazione dinamica delle strutture di controllo per i file aperti, sia a livello dell'intero sistema sia per i singoli processi.

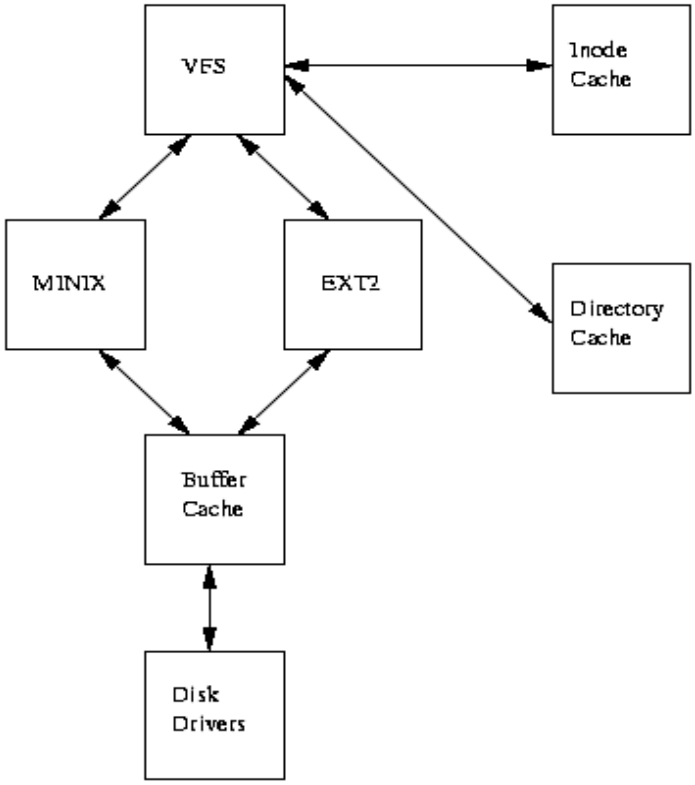


Figura2.1 Organizzazione di base del VFS.

2.1 La cache dei buffer

Quando i file system vengono montati inoltrano numerose richieste di lettura e scrittura di blocchi di dati ai dispositivi a blocchi. Tutte queste richieste vengono trasmesse ai driver di dispositivo sotto forma di strutture dai buffer_head tramite chiamate di routine standard del kernel. Queste strutture forniscono delle informazioni necessarie di cui i driver dei dispositivi a blocchi hanno bisogno; l'identificatore di dispositivo identifica il dispositivo stesso, mentre il numero del blocco indica al driver quale blocco leggere.
Tutti i dispositivi a blocchi vengono considerati come collezioni lineari di blocchi di uguale dimensione. Per rendere più rapido l'accesso ai dispositivi fisici a blocchi, Linux gestisce una cache di buffer a blocchi. La cache dei buffer è composta da due parti funzionali. La prima riguarda le liste dei buffer a blocchi liberi. C'è una lista per ogni dimensione di buffer supportata; i buffer a blocchi liberi vengono accodati all'ingresso di queste liste quando sono creati per la prima volta o quando sono stati eliminati. La seconda parte funzionale invece è costituita dalla cache stessa. Essa è struttura come tabella hash. L'indice hash è generato dall'identificatore del dispositivo proprietario e dal numero di blocco del blocco di dati.

2.2 Gli oggetti del file system

Il VFS interagisce con una serie di oggetti per memorizzare, accedere e gestire i dati su dispositivi a blocchi. Questi oggetti vengono riportati di seguito.
FILE. Definisce le informazioni su un file attualmente aperto e sul processo che vi accede. La struttura di questo oggetto è detenuta solo nel kernel e scompare alla chiusura del file.
INODE. Rappresenta un oggetto base presente all'interno del file system. Può essere costituito da un file normale, una directory, un collegamento simbolico oppure altri oggetti. Può sembrare che file e inode siano simili. In realtà alcuni oggetti hanno degli inode ma non hanno mai dei file (es. collegamenti simbolici), per contro alcuni file sono privi di inode (es. i pipe, i socket). Inoltre un file possiede informazioni di stato che un inode non possiede; in particolare la posizione all'interno del file in cui verrà effettuata la successiva operazione di lettura o scrittura (f_pos).
I FILE SYSTEM. E' una raccolta di più inode, con un inode particolare, noto come root. L'accesso agli altri inode avviene a partire dalla root. Ogni file system è rappresentato da una struct super_block e ha una serie di metodi memorizzati in struct super_operations. Oltre a queste informazioni il VFS è a conoscenza dei tipi diversi di file system, grazie alla struct file_system_type contenente soltanto un metodo, read_super, che istanzia un super_block.
NOMI. L'accesso a tutti gli inode presenti in un file system avviene per nome. Il processo di ricerca che consente di risalire dal nome all'inode può essere oneroso, per questo il livello di VFS gestisce una cache (dcache) di nomi attualmente attivi e utilizzati di recente. La dcache è struttura in memoria come albero; ogni nodo dell'albero corrisponde a un inode in una data directory avente un determinato nome. Se un qualsiasi nodo dell'albero dei file si trova nella cache, ogni predecessore di tale nodo si trova anch'esso nella cache. Ogni nodo

dell'albero è rappresentato da una struct dentry avente una serie di metodi memorizzati in struct dentry_operations. Le dentry fanno da intermediari tra i file e gli inode. Ciascun file punta alla dentry aperta, mentre ciascuna dentry punta all'inode cui fa riferimento. Da questo deriva che, per ogni file aperto, la dentry di tale file, e tutti i genitori di tale file, vengono copiati nella cache.

2.2.1 File

I file vengono rappresentati mediante la *struct file*, avente una serie di metodi memorizzati in *struct file_operations.*
La struttura file viene definita in linux/fs.h. I campi della struttura che vengono richiamati dal romfs sono:
-f_dentry: registra l'entry dcache che punta all'inode corrispondente a questo file;
-f_pos: registra la posizione corrente del file, corrispondente all'indirizzo che sarà utilizzato per la successiva richiesta read;
Le funzioni per i file che appaiono nel romfs sono:
-readdir: ha il compito di leggere le entry di directory dal file, che sarà presumibilmente una directory, e di restituirle utilizzando la funzione di callback filldir_t. Questa funzione ha come argomenti l'handle void* che stato passato assieme al puntatore a un nome, alla lunghezza del nome, alla posizione nel file in cui questo nome è stato trovato, e al numero di inode associato al nome. Se la funzione filldir restituisce un valore non nullo, readdir assumerà di aver completato il suo compito, e ritornerà. Quando readdir raggiunge la fine della directory, dovrebbe ritornare il valore 0. In caso contrario, può ritornare dopo che solo alcune entry sono state consegnate a filldir, in tal caso deve restituire un valore non nullo. In caso di errore restituisce un valore negativo.

2.2.2 Inode

E' l'unità fondamentale alla base di qualsiasi file system in Unix. C'è un inode per ogni file; un file viene identificato univocamente dal file system su cui risiede e dal suo numero di inode sul file system.
Per accedere ad un file il VFS cerca il nome nel file system per trovare a quale inode esso punta. Tutti i componenti che vengono cercati, siano essi file o directory, restituiscono un numero di inode. Un volta ottenuto questo numero, la funzione iget() trova e restituisce l'inode specificato. La funzione iput() viene usata in seguito per rilasciare l'accesso all'inode. Queste funzioni sono simili a malloc() e free(), con la differenza che un inode aperto può detenere più di un processo alla volta.
Linux gestisce una **cache degli inode** attivi e di quelli utilizzati di recente. Ci sono due percorsi mediante i quali è possibile accedere a tali inode. Il primo è tramite la dcache. Ogni dentry della dcache fa riferimento ad un inode, pertanto tiene tale inode nella cache. Il secondo percorso ricorre alla tabella hash dell'inode. Ogni inode ha un valore hash basato sull'indirizzo del superblocco del file system e sul numero dell'inode. Gli inode aventi lo

stesso valore hash vengono quindi concatenati, formando una lista a doppio collegamento.
Anche in questo caso il VFS prevede una struttura inode. I campi di tale struttura che vengono usati dal romfs sono:
-i_ino: inizializzato dal VFS per indicare quale inode leggere (viene richiamato nelle funzioni: romfs_read_inode, romfs_readdir e romfs_lookup);
-i_op: puntatore alla struttura contenente l'inseme delle operazioni che si possono compiere per un inode;
-i_sb: puntatore alla struttura del superblocco;
-i_size: dimensione del file;
-i campi i_atime, i_ctime, i_mtime che indicano rispettivamente i tempi di ultimo accesso, creazione e modifica dei file. Nel romfs vengono posti tutti uguale a zero.
-i campi i_gid, i_uid che contengono il groupid e lo userid del proprietario. Anche questi campi sono pari a zero nel romfs.
-i_mode: contiene il tipo di file (regolare, directory, speciale, ecc.)
-i_nlink: contatore agli hard link di un determinato inode. Questo contatore viene decrementato ogni volta che si cancella un collegamento. Il romfs è un file system di sola lettura per cui questo campo viene posto a 1 e rimane sempre tale.
La funzione inerente alla struct inode_operations richiamata dal romfs è la *lookup (struct inode *,struct dentry *, struct nameidata *)*. Questa funzione viene chiamata quando il VFS cerca un inode nella directory padre. Il nome da cercare si trova nella dentry. Questo metodo chiama la d_add() per inserire l'inode trovato nella dentry. Se la ricerca fallisce, ciò viene segnalato dalla restituzione di una dentry negativa, con un puntatore di inode NULL.

2.2.3 I File System

Linux ricava le informazioni sui nuovi tipi di file system mediante chiamate a *register_filesystem* (e le ignora dopo le chiamate all'omologa unregister_filesystem). Essa viene richiamata dal romfs da module_init, quando quindi viene caricato il file system come modulo. Il VFS chiamerà quindi il metodo *get_sb()* dello specifico file system. La funzione get_sb() del romfs ritorna il metodo get_sb_dev (monta un file system residente in un block device), il quale a sua volta richiama la funzione *romfs_fill_super* che inizializza in maniera opportuna la struttura del superblocco (rappresenta il file system).
Alcuni dei campi che fanno parte della struct super_block e che quindi vengono usati dal romfs sono:
-s_magic: registra un numero di identificazione che è stato letto dal dispositivo, per confermare che i dati presenti sul dispositivo corrispondono al file system in questione;
-s_flags: riguarda i flag applicati ad un determinato file system. Il romfs usa il flag MS_RDONLY, indica che il file system di riferimento è stato montato come di sola lettura. Nessuna scrittura o modifica indiretta sarà consentita;
-s_root: fa riferimento alla root del file system. Nel romfs viene creato caricando l'inode root dal file system e passandolo a d_alloc_root;
-s_op: puntatore alla struct super_operations.
Le funzioni della *struct super_operations* prese in considerazione dal romfs sono:

-alloc_inode: viene chiamato per allocare memoria per la struct inode e quindi per inizializzarla;
-destroy_inode: rilascia le risorse allocate dalla struct inode;
-read_inode: richiamato per leggere uno specifico inode. Nell'argomento struct inode* passato a questo metodo, saranno inizializzati i campi i_sb e i_dev, e in particolare i_ino, per indicare quale inode deve essere letto e quale file system deve leggerlo;
-statfs: quando il VFS necessita di informazioni statistiche sul file system;
-remount_fs: richiamato quando un file system viene rimontato.

2.2.4 Nomi

Il livello del VFS si occupa di tutta la gestione dei percorsi dei file e li converte in entry della dcache prima di consentire al file system sottostante di vederli. L'unica eccezione è quella costituita dalla destinazione di un collegamento simbolico, che viene passato inalterato al file system sottostante, che deve quindi interpretarlo. La dcache è costituita da lotti di struct dentry. Ciascuna dentry corrisponde a un componente nome di file del file system e all'eventuale oggetto associato con tale nome. Ciascuna dentry fa riferimento al rispettivo genitore, che deve esistere nelle dcache.dentry e inoltre deve registrare le relazioni di montaggio del file system.
I campi della struct dentry che interessano il romfs sono:
-d_inode: puntatore all'inode correlato a questo nome. Se questo campo è NULL siamo di fronte ad una entry negativa, il che implica che il nome non esiste;
-d_name: può puntare al campo d_iname, il quale memorizza i primi 16 caratteri del nome del file, per facilitare il riferimento. Se il nome corrisponde esattamente, d_nam.name punta qui, altrimenti punta a un'area di memoria allocata separatamente.

3. ZSTFS: una versione compressa del ROMFS

3.1 Perchè dividere i file in blocchi

Un importante caratteristica diversifica il nostro file system dal romfs: la suddivisione dei file in blocchi.
L'idea di dividere i file in blocchi nasce dalla necessità di non decomprimere i file per intero se si legge solo una parte di essi. Questo metodo diventa particolarmente significativo per file di grandi dimensioni in quanto permette l'allocazione in memoria principale solo di una parte di dati e quindi consente un uso più efficiente della RAM.
Per tenere sotto controllo l'insieme dei blocchi di un file usiamo un array dinamico contenente i puntatori ai vari blocchi nel hard disk. L'array viene dichiarato nella struct romfs_inode_info (int *index). I file possono avere blocchi di dimensione variabile. L'index non viene inserito nel caso di un file piccolo quanto la dimensione di un blocco.
Nella figura 3.1 viene riportato un esempio di un file diviso in cinque blocchi. Dallo schema si può notare che l'array allocato in memoria contiene sei indici al posto di cinque. Questo

non è un errore grafico, in genere l'ultimo indice ci servirà per calcolare la fine dell'ultimo blocco.

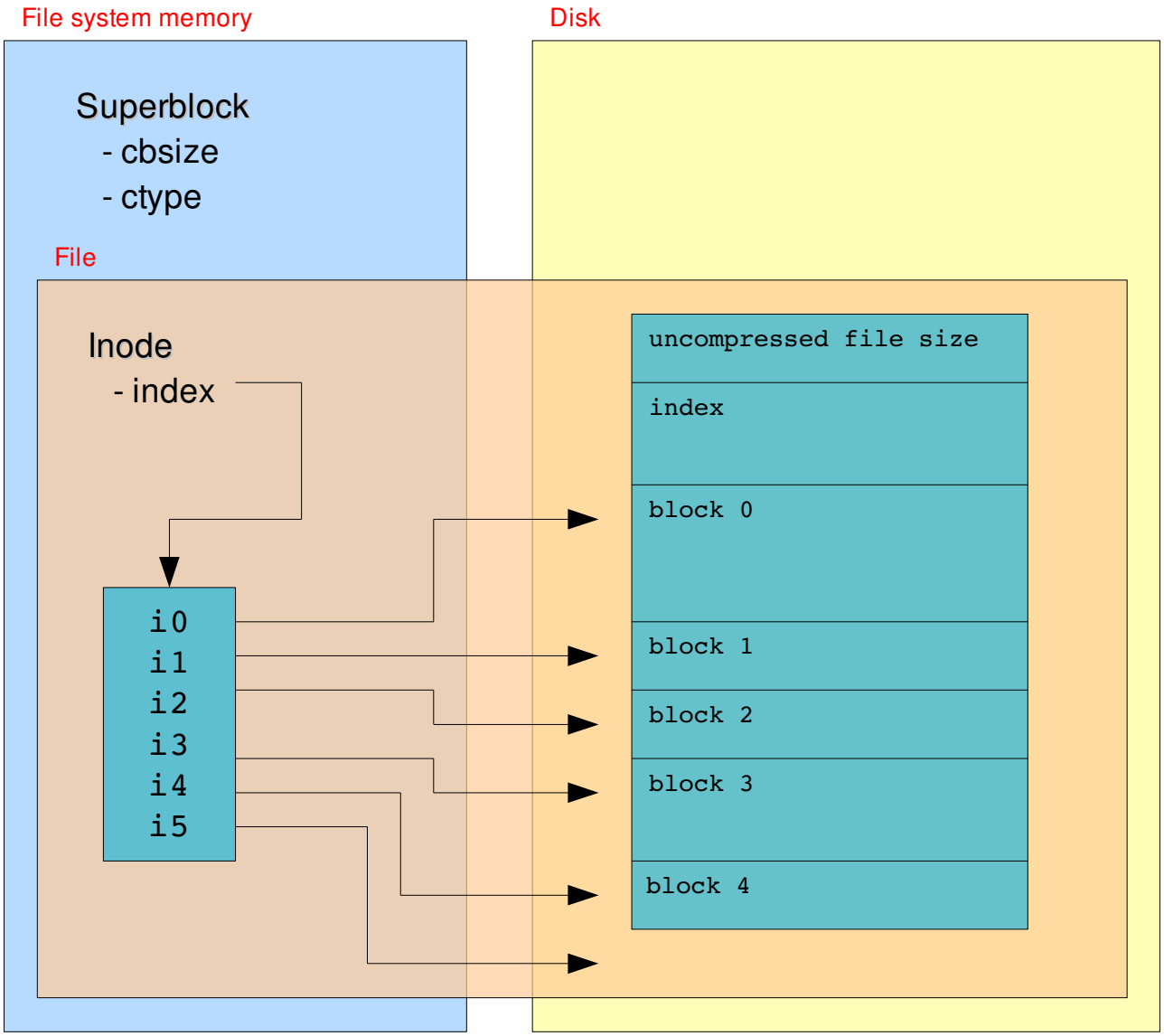


Figura 3.1: Rappresentazione grafica della suddivisione in blocchi di un file sia nel hard disk che nella memoria principale.

3.2 STRUTTURA: Zstfs vs Romfs

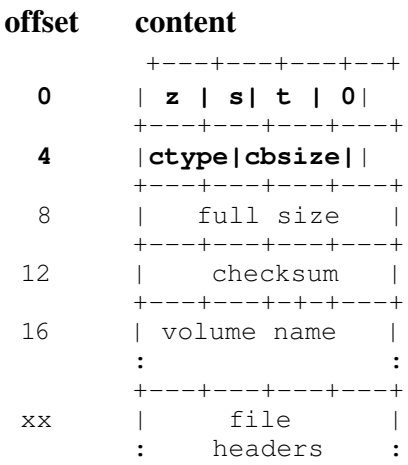


Figura 3.3.1 Header del Zstfs

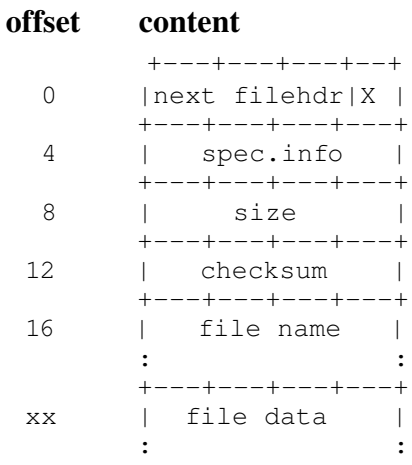


Figura 3.3.2 Header dei file di Zstfs

Per modificare il romfs non si poteva che partire dalle fondamenta. Il zstfs porta pochi ma signficicativi cambiamenti alla struttura del piccolo file system di sola lettura.

Le modifiche inerenti al header del file system sono:
--->i primi quattro byte ora identificano il nuovo file system;
--->i successivi quattro byte (di cui usiamo solo 12 bit) contengono due importanti informazioni: il 5° byte, in cui viene memorizzato *ctype,* fa riferimento al tipo compressione, il 6° byte (di cui vengono usati solo 4 bit) , in cui viene memorizzato *cbsize,* serve a calcolare la dimensione dei blocchi da comprimere. La dimensione dei blocchi può variare da 1024 byte a 32 MB e viene così calcolato: $1024x2^{cbsize}$. I restanti 4 bit sono pari a zero.
Ctype e cbsize sono contenuti nel campo *info* della *struct romfs_super_block*. Per ricavare i due valori separatamente abbiamo usato le due Macro CTYPE e CBSIZE. La prima non fa altro che shiftare il campo info a destra di 24 bit per ottenere il primo byte; mentre la seconda shifta il campo info di 20 bit a destra (per eliminare gli zeri), pone i primi 12 bit in and con il valore 15 (in binario 1111) in modo da ricavare solo i 4 bit contenenti il valore necessario, infine shifta il numero 1024 del valore dei 4 bit (operazione di moltiplicazione).
I due valori così separati vengono memorizzati nella *struct zst_sb_private* (come int cbsize e char ctype). La struttura zst_sb_private viene puntata da s_fs_info, campo della struct super_block del Virtual File System. s_fs_info viene puntata a sua volta dal puntatore i_sb della struct inode, in questo modo si potrà sempre accedere ai campi della struttura zst_sb_private tramite il relativo inode di un file.
Tutti gli altri campi relativi al header del romfs rimangono uguali. Full size rappresenta la

dimensione totale del file system, il checksum riguarda la verifica dei primi 512 byte, il volume name indica l'etichetta del file system, e poi iniziano tutti i file.

In riferimento agli header dei file non abbiamo applicato modifiche per cui i valori che rimangono sono:
--->l'offset del prossimo file header, vale 0 se non ci sono altri file; gli ultimi 4 bit indicano il tipo di file. I primi 3 dei 4 bit (quelli meno significativi) sono necessari per i prossimi 4 byte (spec. info) e possono avere questi valori:
-0 (hard link): spec.info punta al file header da leggere;
-1 (directory): spec.info puna al header del primo file della cartella;
-2 (regular file): spec.info non viene usato;
-3 (symbolic link): spec.info non viene usato;
-4 (block device): spec.info si riferisc al major/minor number;
-5 (char device): spec.info si riferisce al major/minor number;
-6 (socket): spec.info non viene usato;
-7 (fifo): spec.info non viene usato.
Il quarto bit indica se il file è eseguibile.
--->size: memorizza la dimensione del file, nel nostro caso del file compresso;
--->checksum: è una sequenza di bit per verificare l'integrità dei dati;
--->file name: è il nome del file che può arrivare ad un massimo di 128 byte;
--->file data: sono i dati contenuti nel file.

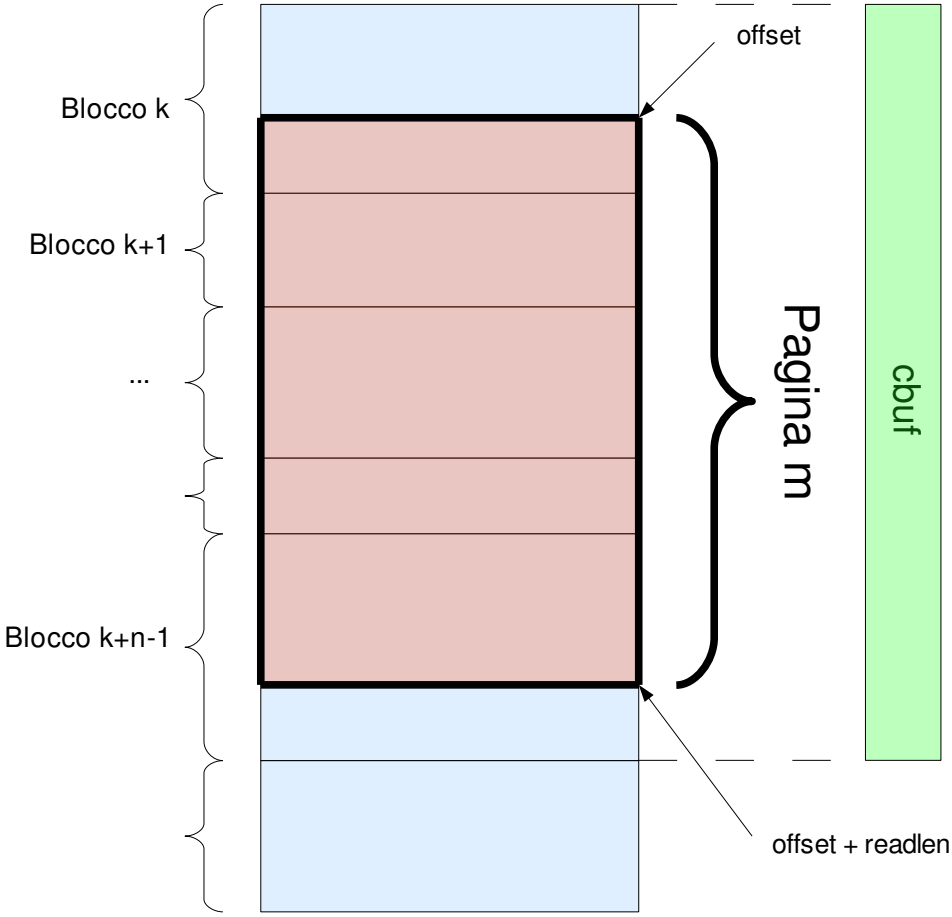
3.3 Zstfs: un' evoluzione del Romfs

I cambiamenti più rilevanti si possono riscontrare nelle due funzioni **romfs_read_inode** e **romfs_readpage**.

La prima funzione prende come parametro un inode del Virtual File System e riempie la struttura dell'inode con le relative informazioni, tra cui anche la size (dimensione del file non compresso). Nel caso in cui l'inode è un file regolare viene letto l'index dal'hard disk (assoluto rispetto all'intero file system), convertito in little endian (se è richiesto), e allocato in memoria.
Si è scelto di non memorizzare l'indice nel file system per i file di piccole dimensioni (cioè più piccoli di un blocco) per evitare ridondanza, esso sarebbe infatti composto da un unico puntatore che indica l'inizio dei dati, informazione già conosciuta. Probabilmente questa ottimizzazione è inutile considerando l'esiguo overhead per questo tipo di file. Inizialmente è stata fatta questa scelta pensando che la maggior parte dei file sono di piccole dimensioni.
Tutti gli altri tipi di file (cartelle, link simbolici, fifo e altri file speciali) non sono compressi e quindi vengono gestiti esattamente come nel romfs.
La funzione romfs_read_inode ha il compito di caricare l'header di ogni file, essa svolge tutte le normali operazioni di romfs e in più si occupa di caricare l'indice dei blocchi compressi. Esso viene salvato in un array dinamico (la dimensione dell'indice è in funzione

della dimensione del file prima che venga compresso) e immagazzinato nella struttura romfs_inode_info (che contiene anche altre informazioni sugli inode), in modo che possa essere recuperato con facilità nella funzione romfs_readpage. La dimensione dell'indice può essere calcolata in modo molto semplice: dimensione file non compresso/cbsize + 1. La cella in più serve per l'ultimo puntatore fittizio: esso non punta ad un blocco, ma è necessario per poter calcolare la dimensione dell'ultimo blocco.

La funzione romfs_readpage prende come parametro un file e una page e legge i blocchi decompressi relativi a quel file.
Per decomprimere i blocchi vengono allocati due buffer temporanei cbuf e dbuf che verrano poi deallocati alla fine dell'operazione.
Per poter recuperare una pagina è necessario leggere e decomprimere tutti i blocchi che la compongono. I blocchi sono di dimensione variabile (in quanto compressi), le pagine invece hanno una dimensione fissa (in genere 4KB), eccezion fatta per l'ultima di ogni file.
Si può notare che più i blocchi compressi sono grandi, e maggiore è la quantità di dati che vengono decompressi inutilmente. Si consideri anche che all'aumentare della dimensione dei blocchi mediamente aumenta l'efficienza della compressione. Per questo è necessario trovare un compromesso che tenga conto di entrambi gli aspetti.
Per poter localizzare in tempo costante i blocchi che corrispondono alla pagina ci si serve dell'indice alloccato in memoria nella funzione romfs_read_inode(). L'indice è situato in memoria centrale quindi l'operazione è molto veloce.
Dividendo l'offset per il campo cbsize si ottiene il numero del blocco in cui inizia la pagina richiesta (Pagina m in questo caso). Con un'operazione simile si può ottenere il numero dell'ultimo blocco da leggere: (offset+readlen)/cbsize.
Ora che si conoscono i blocchi (da k a k+n-1) è sufficiente consultare l'indice per sapere la loro reale ubicazoine su disco e la loro dimensione: index[i] è il puntatore all'inizio del blocco i e index[i+1]-index[i] è la sua dimensione. Entrambe queste informazioni sono memorizzate per tutti i blocchi di ogni file (anche l'ultimo).



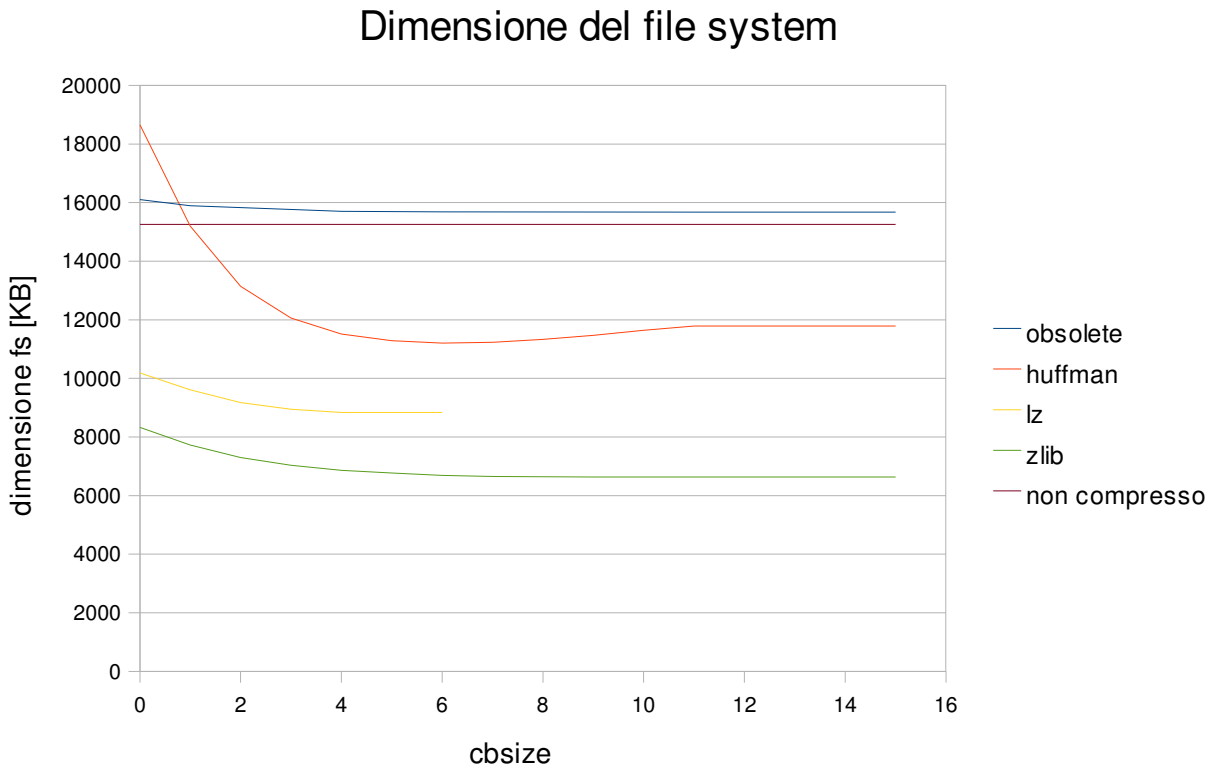
Tutti i blocchi vengono copiati nel buffer temporaneo cbuf per la decompressione. Alla fine dell'operazione solo la pagina richiesta vine inserita nalla page cache.
Una possibile ottimizzazione di zstfs consiste nell'inserire nella cache tutte le pagine contenute nei blocchi letti, in modo da evitare di ri-decomprimere gli stessi dati più di una volta (questo fenomeno non verrebbe eliminato completamente ma sensibilmente ridotto).

3.3.1 Big Endian e Little Endian

Romfs è un file system portabile, (come il kernel Linux) ed è quindi stato scelto uno standard per memorizzare i dati indipendente dal sistema che ha generato lo specifico fs.
Romfs usa big endian e il sistema x86 usa internamente una rappresentazione little endian. Il kernel mette a disposizione la funzione be32_to_cpu() per effettuare questa conversione (solo se necessario) degli interi a 32bit.
Per seguire le scelte tecniche di romfs anche l'indice dei file è stato salvato nello stesso formato, mantenendo la portabilità anche in zstfs (sebbene esso non sia stato testato su altre architetture).

3.3.2 Benchmark

Abbiamo realizzato dei test per verificare il comportamento della dimensione del fs al variare della dimensione dei blocchi. Inoltre abbiamo potuto confrontare le prestazioni dei vari algoritmi (solo per quanto riguarda lo spazio occupato e non per la velocità di decompressione o la memoria occupata).
Le funzioni rappresentate in questo grafico dipendono dal tipo dei dati della cartella sorgente, questi dati non possono rappresentare il comportamento generale dei vari algoritmi.
La cartella usata in questo esempio contiene file di testo e alcuni archivi compressi per un totale di 14,9MB.



Sull'asse delle ascisse è stato rappresentato cbsize e non la dimensione reale dei blocchi per avere una migliore rappresentazione dei dati. Bisogna quindi considerare che all'aumentare di cbsize la bimensione dei blocchi aumenta esponenzialmente, e con essa la memoria richiesta per la lettura di ogni singola pagina. Per questo motivo non si deve valutare solo la dimensione finale del file system per scegliere la dimensione dei blocchi.
L'algoritmo di Huffman ha un'andamento particolare: per valori maggiori di 6 di cbsize (cioè con blocchi maggiori di 64KB) aumenta la dimensione del fs. Questo fatto deve essere duvuto all'algoritmo stesso considerando che l'header del zstfs diminuisce all'aumentare della dimensione dei blocchi. Un altro fatto che emerge e` che per valori di cbsize inferiori a 2 l'algoritmo di huffman non ottine nessuna compressione dei dati, anzi il loro volume

cresce. Questo e` dovuto all'albero di decisione salvato in ogni blocco: esso rappresenta un considerevole overhead che puo` essere abbattuto solo con alti livelli di compressione dei dati.

3.4 Genzstfs

Genzstfs è un programma che ha il compito di generare file system di tipo zstfs. Non dispone di interfaccia grafica ma solo di una semplice TUI. I parametri che si possono specificare sono:

- cartella di input (viene utilizzata la cartella corrente)
- file di output (opzione -f)
  il file system deve essere all'esterno della cartella corrente
- dimensione dei blocchi in byte (opzione -b)
  questo valore deve essere una potenza del 2 fra 2^{10} (1024B) e 2^{25} (32MB).
  E` consigliato impostare la dimensione dei blocchi a un valore basso peché la funzione kmalloc() può alloccare solo 128KB di memoria. Inoltre, questo valore può scendere ulteriormente se la memoria in kernel space è troppo frammentata. Nel caso in cui l'allocazione di un blocco dovesse fallire verrà mostrato all'utente un input output error.
- tipo di compressione (opzione -c)
  Gli algoritmi di compressione supportati da zstfs sono:
  - obsolete
    Algoritmo creato al solo scopo di testare il file system. Non comprime i dati, ma aggiunte un header e un footer col solo scopo di separare i vari blocchi.
  - zlib
    Algoritmo di compressione utilizzato nei gzip (è l'opzione di default anche se zstfs non lo supporta)
  - bz2
    Da implementare
  - bwt
    Burrows-Wheeler Transform
    Da testare (anzi, reimplementare)
  - huffman
    È basato su un'implementazione open source di terzi.

3.4.1 Requisiti software

- Python 2.5 (con un po' di fortuna anche Python 2.4)
- Genromfs

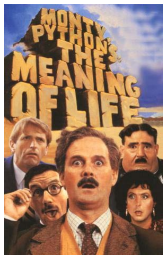
3.4.2 Installazione

make install

3.4.3 Come funziona?

Esso è scritto in Python e fa uso di genromfs. Prima chiama uno script (**compresstree.py**) che crea una cartella nascosta .zst/files che contiene tutti i file della cartella sulla quale viene invocato genzstfs compressi con l'algoritmo di compressione indicato, ognuno dei quali corredato di indice e dimensione del file decompresso. Poi richiama **genromfs** sulla cartella .zst/files che crea un filesystem romfs, e infine chiama un altro script (**fixoutput.py**), che trasforma il filesystem romfs in un filesystem zstfs.

3.4.4 Perché Python?



La scelta di questo linguaggio è duvuta alla necessità di sviluppare genzstfs in un breve periodo di tempo. Esso privilegia la semplicità di sviluppo e la leggibilità del codice a discapito della velocità di esecuzione. Bisogna considerare anche che gli algoritmi di compressione possono essere sviluppati in altri linguaggi, incrementando significamente la performance (ad esempio zlib o huffman sono scritti in C).