8. IL FILE SYSTEM

In tutti i sistemi operativi il File System costituisce una parte molto rilevante del codice complessivo, ma noi lo tratteremo abbastanza brevemente, perchè dal punto di vista dei meccanismi fondamentali del sistema operativo il file system non presenta molti aspetti significativi.

In LINUX sono presenti molti file system diversi, alcuni per ragioni storiche, altri per ragioni di compatibilità con altri sistemi (ad esempio msdos) e per la gestione di particolari tipi di periferiche (ad esempio ISO 9660 per la gestione dei CD).

Nel seguito descriveremo il file system “System V”, che è supportato da altre versioni di UNIX e in LINUX esiste principalmente per compatibilità.

I programmi applicativi sono resi indipendenti dall’esistenza dei diversi file system grazie ad uno strato di interfaccia detto VFS (Virtual Filesystem Switch), che permette a LINUX di far coesistere i diversi file system redirigendo le richieste di servizi alle routine del file system corretto.

1. I servizi per la gestione dei file

Un file contiene una sequenza di byte. Il processore non può lavorare direttamente su tale contenuto ma deve prima trasferirlo in memoria. I servizi fondamentali di accesso a un file permettono perciò di trasferire byte dal file in memoria (**lettura** del file) oppure di trasferire byte dalla memoria al file (**scrittura** del file). Tutte le operazioni di lettura e scrittura operano su sequenze di byte a partire da un byte indicato dall’indicatore di **posizione corrente**; tali operazioni inoltre spostano la posizione corrente in modo che un’eventuale successiva operazione inizi esattamente dove la precedente è terminata.

Prima di operare su un file è necessario aprirlo e, se necessario, crearlo o cancellarne il contenuto. Nell’operazione di apertura il file viene identificato in base al nome. Al momento dell’apertura il sistema esegue dei controlli e restituisce un numero intero non negativo (unsigned short integer) detto **descrittore del file**. Per qualsiasi operazione successiva all’apertura il file viene identificato tramite il descrittore e non più in base al nome. Il descrittore utilizzato dalle funzioni della

libreria LINUX svolge quindi una funzione analoga al puntatore al file (file pointer) della libreria del linguaggio C.

Al momento dell'apertura viene anche inizializzata la posizione corrente del file; la normale apertura pone la posizione corrente all'inizio del file (byte 0).

A differenza della libreria del linguaggio C, che fornisce solamente una funzione (*fopen*) per l'apertura dei file, la libreria di LINUX fornisce due diverse funzioni: *open* per aprire file già esistenti e *creat* per aprire, creandoli, file nuovi:

*int open(char * nomefile, int tipo, int permessi)*

*int creat(char * nomefile, int permessi)*

In ambedue "nomefile" deve essere un valido nome completo e l'intero "permessi" serve a gestire i permessi di accesso; nella open l'intero "tipo" serve ad indicare se si vuole un'apertura in lettura e scrittura oppure in sola lettura o sola scrittura. Si rimanda al manuale per i valori da attribuire agli interi tipo e permessi.

La funzione *close(int fd)* elimina il legame tra il descrittore e il file; dopo la close il valore del descrittore è libero e può essere associato ad un altro file, mentre il file chiuso non è più utilizzabile; questa funzione è ovviamente analoga alla funzione *fclose* della libreria del C.

Le operazioni fondamentali su file sono la lettura e scrittura, realizzate tramite le due funzioni seguenti:

int letti = read(int fd, char buffer[ ], int numero)

int scritti = write(int fd, char buffer[ ], int numero)

In ambedue *fd* è il descrittore del file sul quale operare, *buffer* è il vettore del programma dal quale leggere o sul quale scrivere, e *numero* è il numero di byte da trasferire. I valori di ritorno indicano il numero di byte effettivamente letti o scritti; il valore –1 indica che si è verificato un errore.

In lettura il valore di ritorno 0 indica che è stata raggiunta la fine del file.

Operazioni non sequenziali

Le operazioni read e write sono sequenziali, nel senso che ogni operazione si svolge a partire dalla posizione corrente lasciata dalla operazione precedente. In LINUX esiste una funzione, *lseek( )*, che permette di operare in maniera non sequenziale su un file. La funzione lseek ha il prototipo

long lseek(int fd, long offset, int riferimento).

Essa non esegue alcuna lettura o scrittura, ma modifica il puntatore alla

posizione corrente del file secondo la regola illustrata in tabella 1 e restituisce il nuovo valore della posizione corrente.

riferimento	**nuova posizione corrente**
0	inizio del file + offset
1	vecchia posizione corrente + offset
2	fine del file + offset

Tabella 1 – gestione della posizione corrente in lseek

Si noti che questa funzione è molto flessibile, ad esempio

- lseek(fd, 0L, 1) restituisce l'attuale posizione corrente senza modificarla,
- lseek(fd, 0L, 0) posiziona all'inizio del file,
- lseek(fd, 0L, 2) posiziona alla fine del file.

(la costante 0L indica uno 0 di tipo long integer),

In figura 1 è mostrato un programma che utilizza alcune delle operazioni citate e il risultato della sua esecuzione.

```
/* esempio di uso delle operazioni LINUX sui file .*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
void main()
{       int fd1;
        int i,pos;
        char c;
                /* apertura del file */
        fd1=open("fileprova", O_RDWR);
                /* visualizza posizione corrente */
        printf("\npos= %ld \n", lseek(fd1,0,1));
                /* scrittura del  file */
        for (i=0; i<20;i++)
        {       c=i + 65; /*caratteri ASCII a partire da A*/
                write(fd1, &c, 1);
                printf("%c",c);}
                /* visualizza posizione corrente; riportala a 0 */
        printf("\npos= %ld \n", lseek(fd1,0,1));
        lseek(fd1,0,0);
                /* lettura e visualizzazione del  file */
        for (i=0; i<20;i++)
        {       read(fd1, &c, 1);
                printf("%c",c);}
        printf("\n");
                /* posizionamento al byte 5 del file e stampa posiz.*/
        printf("\npos= %ld\n", lseek(fd1,5,0));
                /*lettura di 5 caratteri */
        for (i=0; i<5;i++)
        {       read(fd1, &c, 1);
                printf("%c",c);}
        printf("\n");
                /*scrittura di 5 caratteri x*/
        c= 'x';
        for (i=0; i<5;i++)
        {       write(fd1, &c, 1);}
                /* lettura del  file dall'inizio */
        lseek(fd1,0,0);
        for (i=0; i<20;i++)
        {       read(fd1, &c, 1);
                printf("%c",c);}
        printf("\n");
                /* chiusura file */
        close(fd1);
}
```

```
[pelagatt@pelagatti1 temilab]$ ./fileoperations

pos= 0
ABCDEFGHIJKLMNOPQRST
pos= 20
ABCDEFGHIJKLMNOPQRST

pos= 5
FGHIJ
ABCDEFGHIJxxxxxPQRST
```

Figura 1 – Il programma fileoperations e la sua esecuzione

2. I cataloghi e i nomi dei file

Per semplificare la gestione dei file, i nomi dei file sono inseriti in **cataloghi** (**directory**). Un catalogo non è altro che un file dedicato a contenere i nomi di altri file e le informazioni necessarie al sistema per accedere tali file. I file dedicati a servire come catalogo sono detti di **tipo catalogo**, mentre i file che contengono normali informazioni sono detti di **tipo normale**. I programmi applicativi non possono leggere e scrivere i cataloghi tramite read e write come se fossero file normali ma devono utilizzare dei servizi speciali per questo scopo.

Dato che un unico grande catalogo sarebbe troppo scomodo per gestire numeri elevati di file, specialmente da parte di molti utenti diversi, i cataloghi seguono la nota struttura gerarchica. Tale struttura si basa sull'esistenza di un unico catalogo principale, detto **radice** (**root**), che il sistema è in grado di identificare e accedere autonomamente, e che può contenere riferimenti sia a file normali sia a file catalogo, i quali a loro volta possono contenere riferimenti sia a file normali sia ad altri file catalogo, costituendo in questo modo la struttura gerarchica dei cataloghi.

Il **nome completo** (**pathname**) di un file di qualsiasi tipo è costituito dal concatenamento dei nomi di tutti i cataloghi sul percorso che porta dalla radice al file stesso, separati dal simbolo "/". La radice è indicata convenzionalmente col simbolo "/" iniziale. In figura 2 è mostrata una struttura gerarchica di cataloghi e i pathname e il tipo dei file coinvolti.

In LINUX non esiste un servizio di cancellazione dei file, ma solamente la funzione *unlink(char * name)*, che elimina un nome di file da un catalogo e, se questo era l'unico nome del file, cancella il file stesso. Esiste anche un servizio *link( )*, che permette di creare un nuovo nome per un file già esistente, in modo che un file possa avere più nomi (e questo fatto giustifica il particolare funzionamento del servizio unlink). Per i dettagli di link si rimanda al manuale.

Per quanto riguarda la programmazione elementare, possiamo limitarci a considerare il seguente modello semplificato: normalmente un file viene creato con la funzione *creat*, che gli attribuisce un nome, ed eliminato con la funzione *unlink*, che elimina il nome e cancella il file.

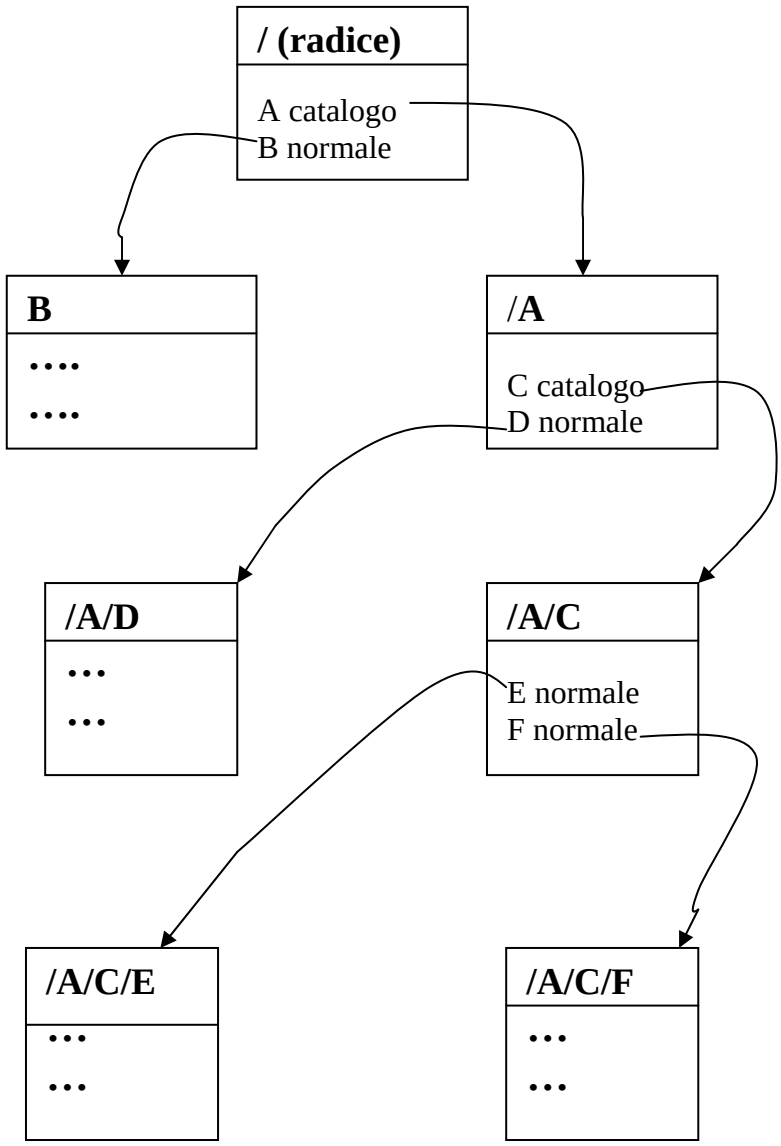

Figura 2 – Struttura dei cataloghi e dei pathname

La funzione creat( ) può creare solamente file normali; per creare dei cataloghi è necessario utilizzare una funzione diversa, *mknod( ),* che verrà spiegata al prossimo paragrafo.

3. Periferiche e file speciali

In LINUX tutte le operazioni di ingresso/uscita sono svolte leggendo o

scrivendo file, perchè tutte le periferiche, compresi il video e la tastiera, sono considerati come file. In particolare, le periferiche sono "viste" dal programma attraverso il file system tramite la nozione di **file speciale**. A questo punto abbiamo incontrato tutti i 3 tipi di file possibili in LINUX: normali, cataloghi e speciali.

Un file speciale è simile a un file normale per il programma, che può eseguire su di esso le operazioni tipiche citate sopra, ad esempio *open*, *read*, ecc…; tuttavia fisicamente esso non corrisponde ad un normale file su disco ma ad una periferica. In realtà, un programma non può eseguire tutti i tipi di operazioni su un file speciale: sono escluse *creat* e le operazioni che non hanno senso per quello specifico tipo di periferica (ad esempio *write* su una tastiera non ha senso). Spesso i file speciali sono posti sotto il direttorio */dev* nelle normali configurazioni di LINUX; pertanto, ad un terminale potrebbe corrispondere un file speciale con il nome completo */dev/tty10*. Un programma potrebbe aprire tale file con un comando tipo *fd=open("/dev/tty10")* e poi scrivere sul corrispondente terminale tramite *write(fd,buffer,numerocaratteri)*, esattamente come se fosse un file normale.

Anche i terminali virtuali creati dall'interfaccia grafica sono associati a file speciali. Il loro nome è */dev/pts/n*, dove n è un numero che varia da un terminale all'altro. I terminali virtuali si comportano come terminali normali, ma la loro creazione è molto più dinamica, perchè, invece di essere definita in configurazione, come per i dispositivi fisici, avviene su comando. Pertanto i nomi dei relativi file speciali sono più aleatori. Per conoscere il nome del file speciale associato ad un certo terminale virtuale è sufficiente dare allo shell di quel terminale il comando *tty*.

L'interprete comandi di LINUX fa in modo che un programma venga posto in esecuzione avendo già i 3 descrittori 0, 1, e 2 aperti, detti standard input **(stdin),** standard output **(stdout**) e standard error **(stderr)**. I file associati a tali descrittori sono i file speciali che corrispondono alla tastiera e al video del terminale, quindi al momento iniziale di esecuzione di un programma la situazione è quella rappresentata in figura 3 e rimane tale se il programma non svolge operazioni particolari per modificarla. Molte funzioni di sistema utilizzano tali descrittori, ad esempio *printf( )* scrive su standard output.

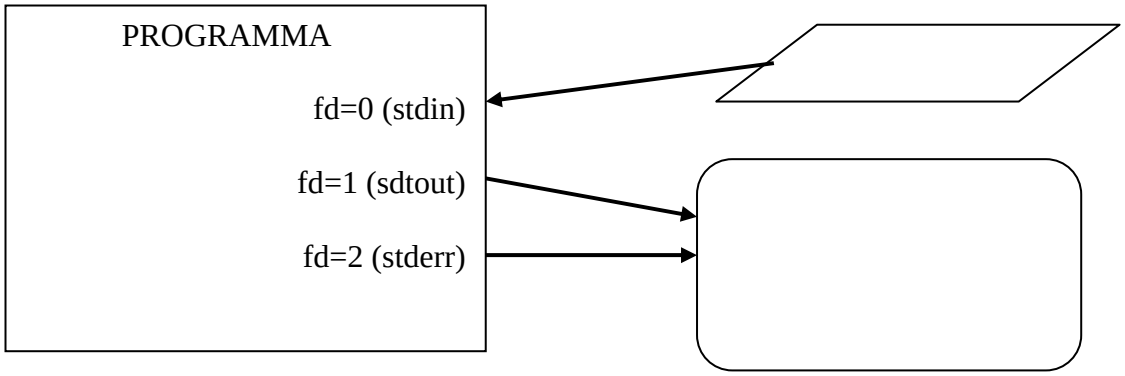

Figura 3

E’ possibile redirigere i 3 descrittori standard, ad esempio associando lo standard output a un file normale, in modo che tutte le stampe prodotte vadano su tale file. Un modo per eseguire tale redirezione consiste nel chiudere il file da redirigere, liberando il descrittore, e poi aprire il file su cui si vuole redirigere: la funzione open infatti riutilizza il primo descrittore libero. Ad esempio, dopo l’esecuzione delle seguenti istruzioni

```
close(0); /* chiude stdin */
fd = open("./inputfile", O_RDONLY);   /* apre il file sul fd=0 */
```

un programma che legge logicamente da stdin, cioè dal descrittore 0, concretamente legge dal file “inputfile” invece che dal terminale.

Esiste la possibilità di duplicare un descrittore associato ad un file già aperto, tramite le funzioni *dup e dup2*, e il risultato è simile ad una doppia apertura.

```
int fd1, fd2;
fd1=open("fileprova", O_RDONLY); /* apertura del file */
fd2=dup(fd1); /* duplicazione del file descriptor */
```

A questo punto esistono due descrittori che si riferiscono allo stesso file. Si tenga presente che, utilizzando le funzioni di LINUX, *anche se un file è aperto più volte e quindi associato a più di un descrittore, il suo indicatore di posizione corrente è*

comunque unico. Si veda il manuale per i dettagli delle funzioni dup e dup2.

3. Il concetto di volume

Il File System non gestisce direttamente i dischi sui quali si appoggia, ma opera attraverso due componenti che forniscono un modello semplificato dei dischi che chiameremo **volume**. Un volume è un vettore di blocchi, e un blocco è una sequenza di byte che vengono trasferiti con una sola operazione tra il disco e la memoria centrale (fisicamente quindi un blocco è costituito da un numero intero di settori del disco). La dimensione dei blocchi è un parametro configurabile del sistema, ma usualmente essa è di 512 o 1024 byte; nel seguito noi faremo riferimento a blocchi da 512 byte. Un blocco può essere letto o scritto in una zona di memoria di pari dimensione detta **buffer**. In figura 4 è mostrato il modello del volume sul quale opera il file system.

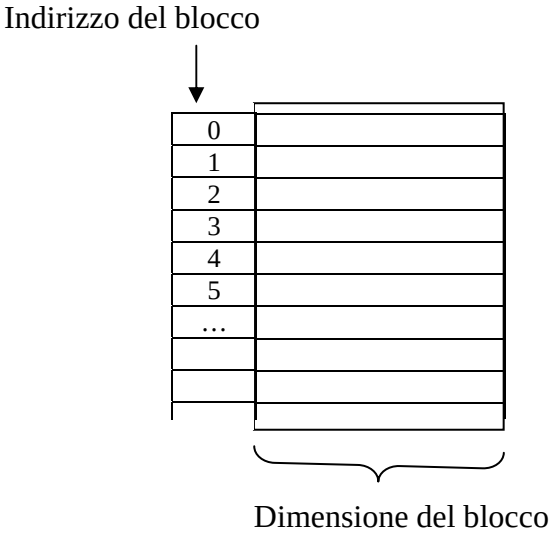

Figura 4 –modello del volume

Abbiamo già visto nel capitolo sulla memoria che esiste un componente, detto **gestore dei buffer** (**buffer cache**), il quale ha il compito di ottimizzare l'impiego della memoria disponibile per mantenere il più possibile in memoria i dati letti dal disco, in modo da evitare letture multiple.

Quando il file system ha bisogno di leggere un blocco, esso non richiede direttamente la lettura al gestore del disco, ma richiede il blocco al gestore dei buffer.

Quest'ultimo verifica se il blocco è già in memoria e in caso contrario richiede al gestore del disco di leggere il blocco dal disco e trasferirlo in un buffer di memoria. L'interazione tra il gestore dei buffer e il gestore del disco sarà descritta nel capitolo relativo ai gestori di periferiche. Dal punto di vista del file system questi due componenti servono a fornire un modello semplice sul quale realizzare la nozione di file, senza dover tenere conto delle particolarità strutturali dei diversi tipi di dischi e dei problemi di ottimizzazione dell'uso delle risorse.

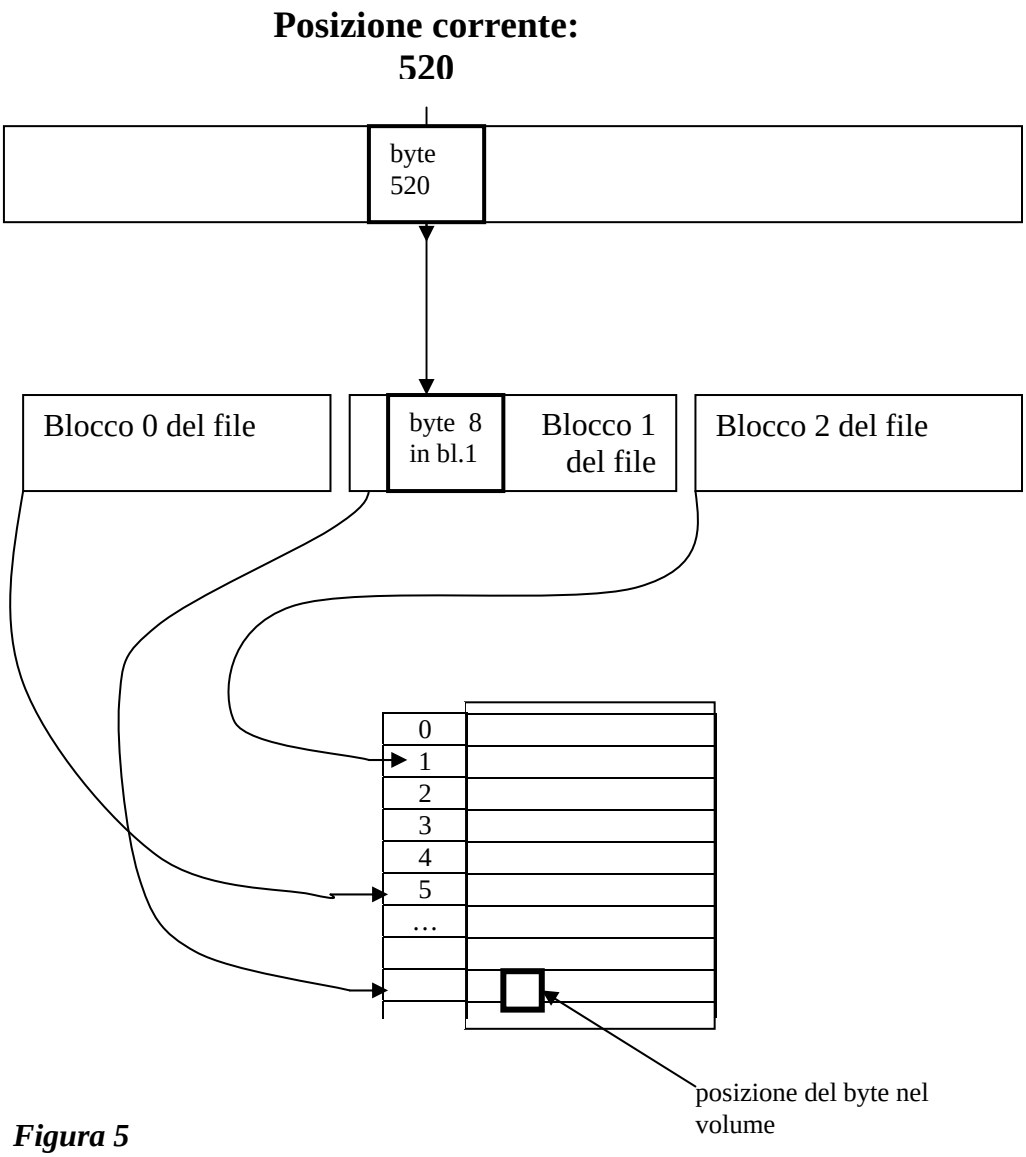

Figura 5

Basandosi sulla nozione di volume è facile interpretare la funzione fondamentale che il file system deve svolgere, cioè la realizzazione dei servizi di

lettura e scrittura di un certo numero di byte a partire da una posizione corrente. Si consideri infatti un file come una sequenza di blocchi numerati logicamente all'interno del file (blocco 0, blocco 1, ecc…); la trasformazione della posizione corrente N in una coppia <numero di blocco logico B, posizione interna al blocco I> può essere realizzata (vedi figura 5) tramite la semplice divisione nel modo seguente, indicato utilizzando le convenzioni del linguaggio C, nell'ipotesi che 512 sia la lunghezza del blocco:

B = N / 512

I = N % 512 (l'operatore % produce il resto della divisione)

La trasformazione del numero di blocco logico all'interno del file in un numero di blocco sul volume è invece supportata dalle strutture dati realizzate dal file system come descritto nel prossimo paragrafo; in figura 5 è mostrato che alla base di tali strutture devono esistere dei puntatori ai blocchi.

4. Organizzazione del volume

Il volume è organizzato dal file system secondo la struttura seguente:

- Il blocco 0 è riservato per contenere il programma di avvio (bootstrap) del sistema operativo, se questo è un volume di boot, altrimenti è allocato ma non utilizzato;
- Il blocco 1, detto **superblock**, contiene informazioni globali relative all'intero volume;
- Un certo numero di blocchi successivi (tale numero è un parametro di configurazione) contiene la **tabella degli i-node**; ogni elemento di tale tabella è detto **i-node**, e descrive un file contenuto nel volume;
- Tutti i blocchi rimanenti costituiscono l'area dati del volume, cioè sono utilizzati per contenere l'informazione contenuta nei file.

Quando un volume viene inizializzato dal file system la tabella degli i-node viene dimensionata, e quindi il numero massimo di i-node (e quindi di file) che possono essere creati sul volume risulta fissato.

Contenuto di un i-node

Tutta l'informazione generale relativa a un file è contenuta nel suo i-node. Un file esiste infatti nel sistema quando esiste il suo i-node. Il riferimento fisico a un file

è costituito dal suo numero di i-node; ad esempio, nei cataloghi sono contenute coppie <nome file, numero di i-node del file>.

Le informazioni più importanti relative ad un file sono le seguenti:

- il tipo del file, che può essere normale, catalogo o speciale
- il numero di riferimenti dai cataloghi al file stesso, cioè il numero di nomi che sono stati assegnati al file (tale numero normalmente è 1, ma può essere superiore);
- le dimensioni del file;
- i puntatori ai blocchi dati che costituiscono il file (eccetto per i file speciali, che non possiedono blocchi dati).

La memorizzazione all'interno di un i-node dei puntatori ai blocchi dati che lo costituiscono segue una struttura orientata a creare un compromesso tra due esigenze: permettere di accedere file grandi senza penalizzare l'accesso ai file piccoli, che sono la maggior parte. Tale struttura è rappresentata in figura 6, e consiste nel riservare all'interno del i-node lo spazio per un numero limitato di indirizzi di blocchi (13 in questo caso), adottando la seguente convenzione per il loro impiego:

- se un file contiene 10 blocchi dati o meno, il loro indirizzo è memorizzato nei primi 10 puntatori e quindi l'accesso al file è immediato a partire dal contenuto dell'i-node;
- se il file supera i 10 blocchi di dimensione, allora l'undicesimo indirizzo individua un blocco che non è utilizzato per contenere dati ma per contenere indirizzi; supponendo che il numero di indirizzi contenuti in un blocco sia 128, allora i blocchi dall'undicesimo al 138esimo sono accessbili tramite la lettura di tale blocco ausiliario;
- se il file supera i 138 blocchi, allora il dodicesimo indirizzo dell'i-node viene utilizzato per realizzare un accesso indiretto a due livelli, permettendo di indirizzare ulteriori 128*128=16.384 blocchi al prezzo della lettura di 2 blocchi ausiliari;
- infine, se il file supera anche la dimensione massima ottenibile al punto precedente, il tredicesimo indirizzo dell'i-node viene utilizzato per un accesso indiretto a 3 livelli, che permette di raggiungere la dimensione massima prevista per i file, che è $10+128+128^2 + 128^3 = 2.113.674$ blocchi

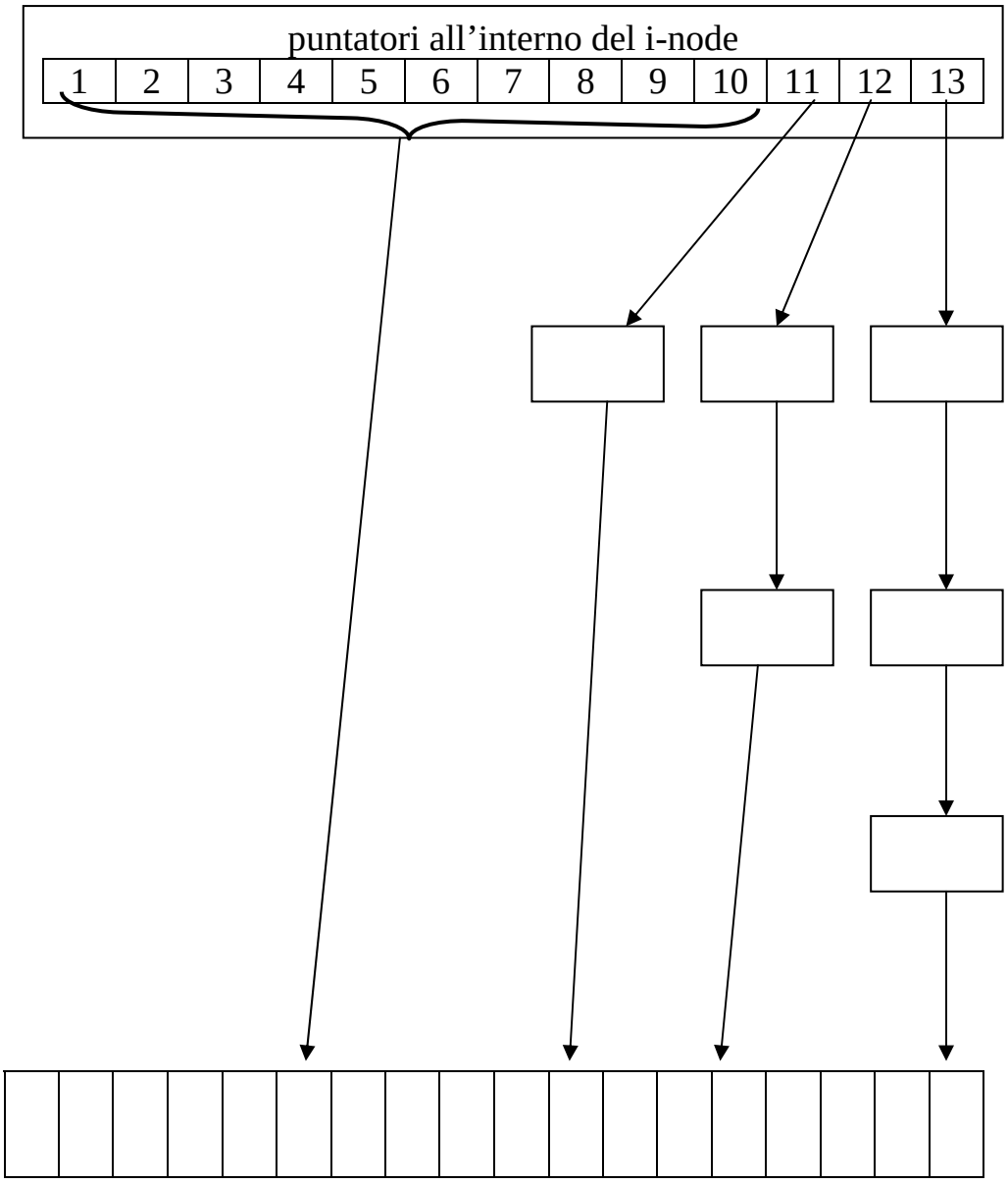

Figura 6

La lista dei blocchi liberi

Il sistema deve poter determinare in certi momenti quali siano i blocchi dati liberi, cioè non appartenenti a nessun file, in modo da poterli utilizzare, tipicamente per allocarli a un nuovo file oppure a un file che cresce di dimensione.

Non è possibile sapere se un blocco dati è libero o no osservandone il contenuto, perchè tale contenuto può essere una qualsiasi sequenza di byte, e quindi non è possibile riservare una particolare sequenza per indicare che il blocco è libero. D'altra parte, cercare di determinare se un blocco è libero andando a vedere se il suo indirizzo non è per caso già contenuto in un i-node di un file è un'operazione impraticabile, perchè richiederebbe un tempo eccessivo.

Per risolvere il problema il sistema mantiene una struttura dati in forma di lista che elenca gli indirizzi dei blocchi liberi. Naturalmente, questa **lista dei blocchi liberi** deve essere contenuta nel volume stesso. Una soluzione potrebbe consistere nel suo inserimento nel superblock, tuttavia, quando il volume viene inizializzato tale lista è molto grande, perchè contiene gli indirizzi di tutti i blocchi dati del volume, e quindi non può essere completamente contenuta nel superblock. Per questo motivo solo la parte iniziale di tale lista è inserita nel superblock, mentre la parte rimanente è inserita in blocchi dati che in tal modo non risultano disponibili per essere allocati ai file. La struttura risultante, mostrata in figura 7, è una lista di blocchi, iniziante nel superblock, il cui contenuto è costituito dagli indirizzi dei blocchi liberi.

Contenuto del superblock

Le informazioni principali contenute nel superblock sono le seguenti:

- la dimensione del volume;
- il numero di blocchi liberi sul volume
- la parte iniziale della lista dei blocchi liberi
- il numero di elementi della tabella degli i-node
- il numero di i-node liberi
- una lista di un certo numero di i-node liberi, utilizzata per ottimizzare la ricerca di un i-node libero quando serve; questa lista viene ricaricata tramite una scansione completa della tabella degli i-node ogni volta che gli i-node indicati sono stati consumati;
- altre informazioni ausiliarie che non interessano qui.

In sostanza, il sistema trova nel superblock le informazioni che gli permettono di eseguire le operazioni più fondamentali, che richiedono di creare o eliminare file oppure di aggiungere o togliere blocchi dati a un file esistente.

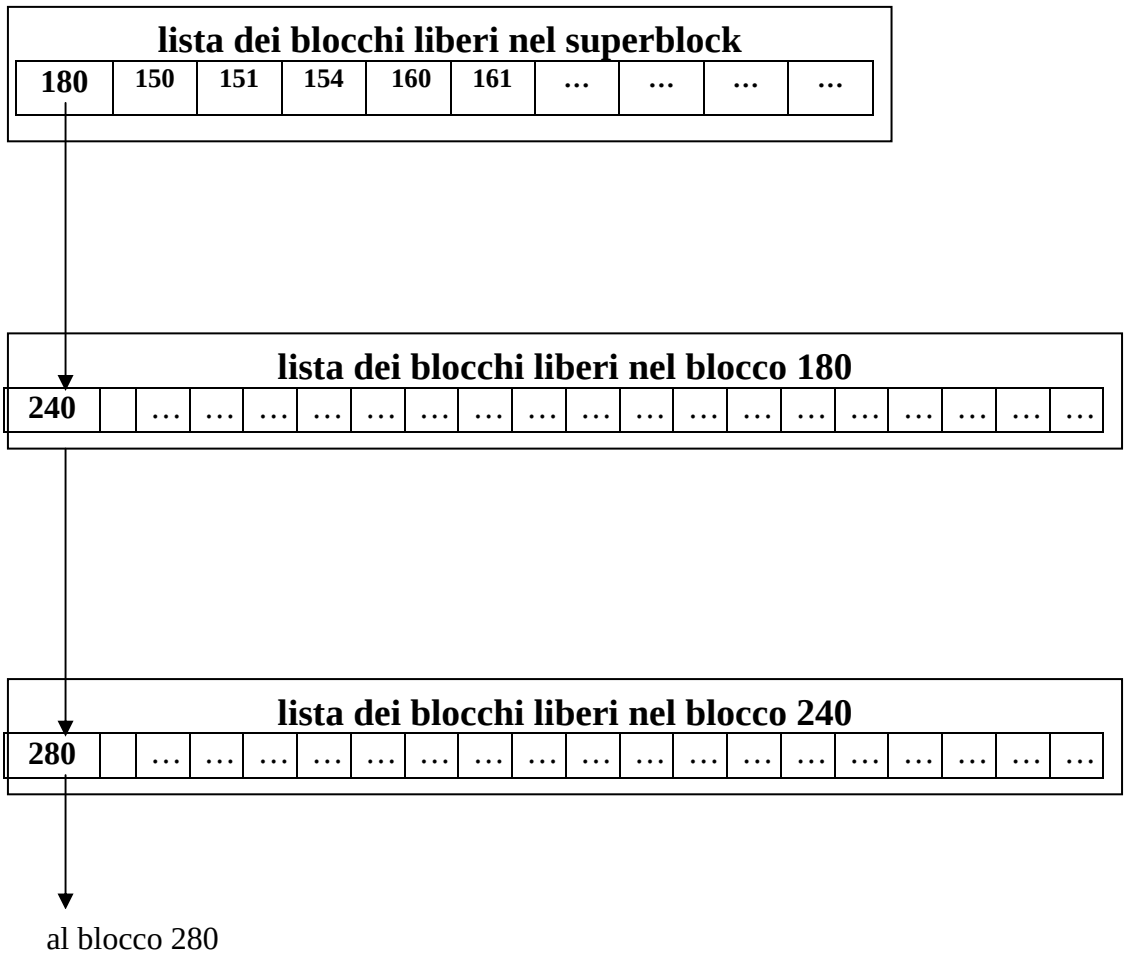

Figura 7 –lista libera di un volume

Esercizi di approfondimento

1) Si descrivano le operazioni sul volume che il file system deve fare quando svolge le seguenti funzioni; scegliere i parametri che devono caratterizzare tali operazioni:

- inizializzazione del volume
- creazione di un file (al momento della creazione di un file si supponga che il sistema gli allochi un certo numero di blocchi possibilmente contigui)
- cancellazione di un file
- crescita di un file
- diminuzione delle dimensioni di un file

2) Un volume si dice frammentato se i suoi file sono allocati in blocchi molto dispersi

invece che in blocchi contigui tra loro. Spiegare perchè si verifica la frammentazione di un volume anche se alla creazione di un file vengono allocati N blocchi contigui.

5. Strutture dati e funzionamento del file system

Il file system deve eseguire i servizi che gli vengono richiesti dai processi. Per far questo esso mantiene durante il funzionamento alcune strutture dati, rappresentate in figura 8. Tali strutture sono:

- **Tabella degli i-node**: è un vettore i cui elementi contengono la copia in memoria degli i-node del volume e alcune informazioni aggiuntive;
- **Tabella (globale) dei file aperti**: è un vettore che contiene un elemento per ogni file aperto nel sistema; tale elemento contiene a sua volta un puntatore all'i-node del file e l'indicatore di posizione corrente del file;
- **Tabella dei file aperti del processo**: è un vettore contenuto nella struttura procrec di ogni processo, i cui elementi puntano all'interno della tabella globale dei file aperti.

Come si vede in figura 8, il descrittore di un file non è altro che un indice all'interno della tabella dei file del processo. E' evidente che ad un dato istante il file system, anche se esistono molte tabelle di file dei diversi processi, riceve richieste di servizi solamente per i file aperti del processo in esecuzione.

E' importante osservare che diverse righe nell'ambito di uno stesso processo o di più processi possono puntare allo stesso file nella tabella globale; in tal caso tutte le oeprazioni sui relativi descrittori condividono la posizione corrente. Il modo più comune per generare due descrittori che puntano allo stesso file nell'ambito di un unico processo è costituito dall'uso del servizio dup. L'operazione fork, creando un processo figlio identico al padre, duplica in particolare la tabella dei file aperti del processo e quindi determina l'esistenza di descrittori in diversi processi che puntano allo stesso file (processi P e Q in figura 7). Invece l'apertura indipendente da parte di un processo R di un file già aperto da parte di P crea una nuova riga nella tabella dei file aperti, con posizione corrente indipendente; tuttavia l'i-node è condiviso, perchè si tratta dello stesso file.

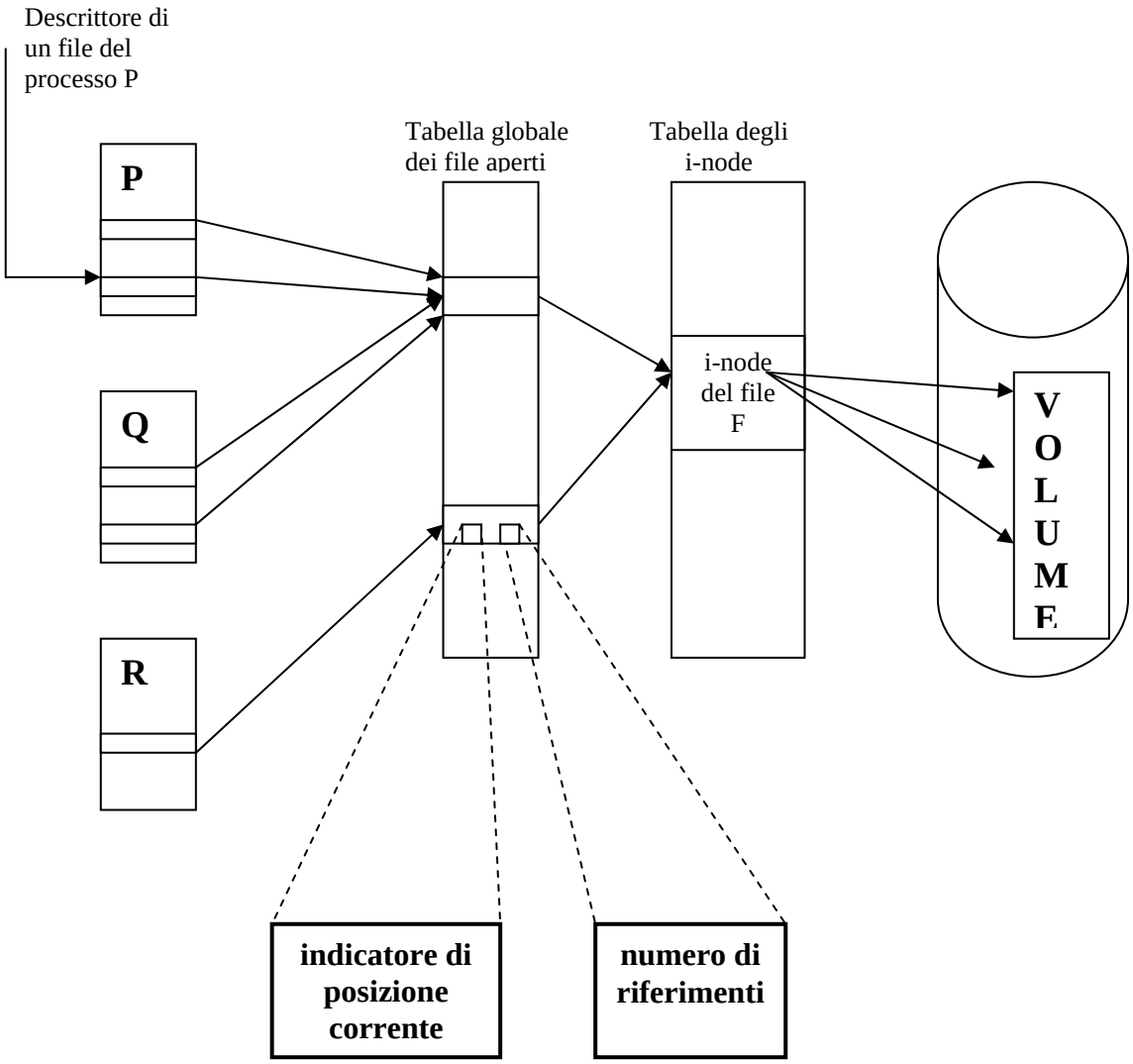

Figura 8 – Strutture dati utilizzate dal file system. La figura è costruita ipotizzando che il processo P abbia duplicato un descrittore a un file F e poi abbia generato un figlio Q; inoltre un processo R ha aperto lo stesso file F.

Gli elementi della tabella globale dei file aperti contengono anche un contatore del numero di riferimenti da parte dei processi al file stesso; quando si chiude un file tale contatore viene decrementato, ma l'elemento viene eliminato solamente se il contatore scende a zero.

Esercizi

1) Descrivere come opera il sistema nell'esecuzione di un servizio di open.
2) Descrivere, con riferimento ai soli file, cosa avviene al momento dell'esecuzione di una fork
3) Scrivere un programma che apre un file, duplica il descrittore, visualizza i valori dei due descrittori, scrive sul file e visualizza nuovamente il valore dei due descrittori; dopo ogni visualizzazione il programma si ferma (ad esempio, aspetta un carattere tramite getchar( )). Lanciare tale programma da due terminali diversi variando il momento in cui si lancia la seconda volta (ad esempio, dopo la prima visualizzazione del primo programma o dopo la seconda) e osservare il comportamento dei descrittori. Spiegare i risultati in base al funzionamento del file system.