

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte IO: Input/Output e File System

cap. IO2 – Il File System

IO2. IL FILE SYSTEM

1 Aspetti generali del FS

La funzione principale del FS consiste nel fornire un livello di astrazione, che chiameremo **Modello di Utente**, omogeneo e ragionevolmente facile da utilizzare, sopra il mondo complesso e variegato dei dispositivi periferici.

Per ottenere questo risultato il FS deve superare una serie di difficoltà:

- la varietà e l'evoluzione delle periferiche
- la varietà dei tipi di utilizzo delle periferiche e dei dati
- l'esigenza di ottimizzare il funzionamento delle periferiche e della memoria centrale

Il modello di utente si basa sulla nozione di **file**; **un file è costituito da una sequenza di byte**.

Questo unico modello permette di accedere sia ai normali file dati sia alle periferiche.

Device Drivers

In qualsiasi istante di tempo esistono molti tipi diversi di periferiche che il sistema deve gestire; inoltre, nel tempo tali tipi evolvono continuamente richiedendo un aggiustamento continuo del sistema. Per superare questo problema LINUX prevede la possibilità di aggiungere al SO nuovi moduli software, detti Device Drivers, ogni volta che si deve gestire una periferica di tipo nuovo; inoltre, dato che su un computer specifico sono presenti solo alcuni tipi di periferiche, LINUX permette di caricare nel sistema solo i Driver effettivamente necessari per gestire la configurazione data.

Per rendere possibile l'inserimento di nuovi driver è necessario:

1. avere un meccanismo generale per l'inserimento nel sistema di nuovo software; questo meccanismo è costituito dai cosiddetti "kernel_modules"
2. definire un modo di interfacciare i driver al sistema, in modo che per i livelli più alti del sistema tutti i driver si comportino in maniera omogenea

In sostanza un Driver è un modulo che opera su un particolare tipo di periferica rendendola visibile ai livelli superiori del sistema secondo un modello generale; LINUX definisce 2 tipi di modelli di driver:

1. **driver a carattere**: in questo modello il driver esegue le operazioni richieste dai livelli superiori (ad esempio read e write) al momento in cui gli vengono richieste
2. **driver a blocchi**: in questo modello il sistema pone le sue richieste in una coda dal quale il driver può prelevarle modificandone l'ordine in modo da ottimizzare l'accesso alla periferica; ovviamente un driver a blocchi non ha senso su periferiche strettamente sequenziali, come una stampante, dove l'ordine dei dati stampati deve essere esattamente quello dei comandi di stampa provenienti dai programmi applicativi, ma ha senso nella scrittura di blocchi su un disco, dove l'ordine di scrittura/lettura può essere ottimizzato.

Virtual Filesystem (VFS) e varietà dei Filesystem

LINUX è in grado di gestire molti filesystem diversi per i seguenti motivi:

- compatibilità col passato: i FS di default di LINUX sono gli extended file systems **ext2**, **ext3** e **ext4**; ext3 e ext4 sono versioni migliorate ed estese, ma compatibili, di ext2
- FS standardizzati per dispositivi particolari (ad esempio ISO 9660 per la gestione dei CD)
- FS per compatibilità con altri sistemi; il più importante è NTFS, il FS di Windows
- FS per ottenere particolari prestazioni, specialmente in ambiente server (ad esempio, un FS ottimizzato per gestire tantissimi piccoli file di messaggi oppure uno ottimizzato per gestire grandi file multimediali)
- Pseudo FS (o FS virtuali): sono FS che operano direttamente in memoria; i 3 più usati sono:
 - procfs - fornisce l'accesso ad informazioni interne di LINUX come se queste fossero memorizzate in una struttura a file
 - sysfs - svolge una funzione simile al precedente
 - tmpfs - permette di utilizzare dei file temporanei in memoria

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

LINUX permette a diversi FS di coesistere nello stesso sistema con il vincolo che ogni partizione (detta anche **Volume** o **Device**) può essere gestita da un unico FS. Ricordiamo che un volume è costituito da un insieme di blocchi identificati da un **LBA (Logical Block Address)**.

Il VFS definisce un modello, che chiameremo **modello del VFS**, basato su diverse strutture dati; tale modello costituisce la rappresentazione omogenea di qualsiasi volume in memoria centrale. I diversi FS possono organizzare i dati sul dispositivo fisico come vogliono, ma devono presentarli al sistema secondo il modello del VFS, caricandoli nelle strutture dati di tale modello.

Ottimizzazione del funzionamento delle periferiche e dell'uso della memoria centrale

Le periferiche sono spesso delle unità relativamente lente il cui uso deve essere ottimizzato. In particolare, l'accesso ai dati su disco costituisce un'operazione i cui tempi si misurano in millisecondi, mentre i tempi dell'esecuzione delle istruzioni da parte della CPU si misurano in nanosecondi. Per questo motivo, come abbiamo visto trattando della gestione della memoria e in particolare della Page Cache, **LINUX tenta di mantenere memoria i dati letti da disco il più a lungo possibile** per poterli riutilizzare se necessario senza doverli rileggere. Per raggiungere questo obiettivo il **VFS e i singoli FS devono collaborare strettamente con il gestore della memoria**.

Architettura complessiva

L'architettura complessiva del sistema per quanto riguarda l'IO è rappresentata in figura 1.1. In tale figura osserviamo che il VFS e i FS non invocano direttamente i gestori a blocchi, ma lo fanno attraverso il componente Page Cache del gestore della memoria (MM).

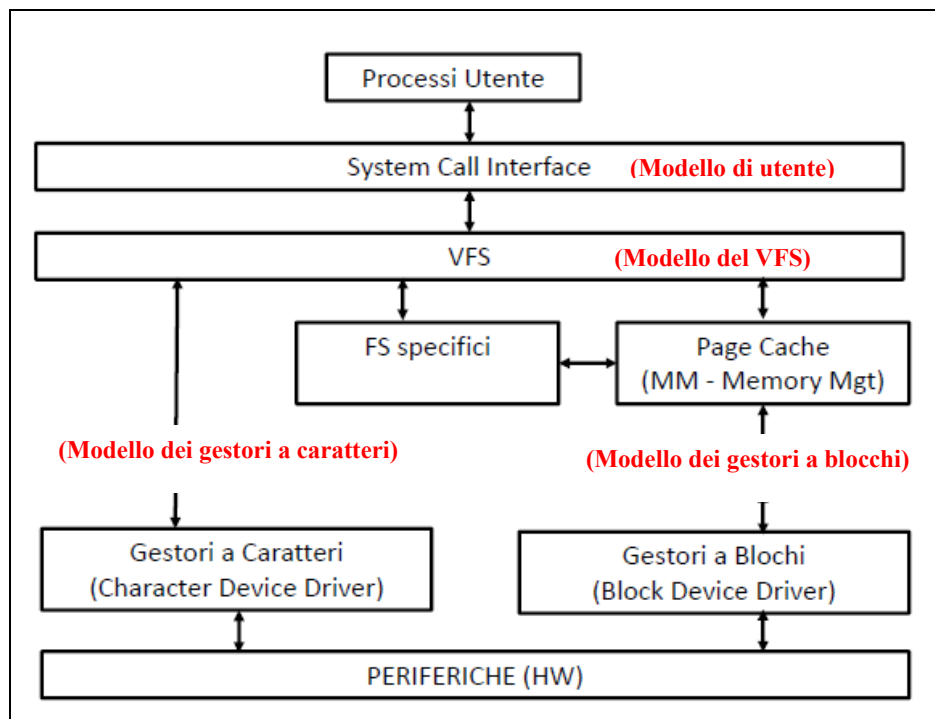


Figura 1.1

Infatti, quando il VFS o un FS ha bisogno di leggere un blocco su un Volume, esso lo richiede in realtà alla Page Cache:

- se il blocco è già in Page Cache, questa restituisce immediatamente l'indirizzo di memoria del blocco;
- in caso contrario la Page Cache alloca lo spazio necessario in una pagina e richiede al gestore a blocchi di leggere il blocco dal dispositivo nella pagina allocata

In ambedue i casi, quando la richiesta è completata il VFS o il FS sono in grado di operare sul blocco richiesto già trasferito in memoria.

2 Il modello d'utente

Possiamo suddividere il modello d'utente in 2 aspetti:

- **accesso al singolo file:** è costituito dalle funzioni utilizzate per scrivere e leggere informazioni sui singoli file – si tenga presente che anche i dispositivi periferici sono gestiti come dei file particolari, detti **file speciali**
- **organizzazione della struttura complessiva dei file:** è costituito dalle funzioni utilizzate per organizzare i file nei direttori (folder, cartelle) e volumi – alcune di queste funzioni richiedono i privilegi di amministratore (root)

2.1 Il modello per l'accesso al singolo file

L'accesso a un file può avvenire secondo due modalità:

1. la mappatura di una VMA sul file tramite la funzione `mmap()`, come descritto nel capitolo sulla memoria
2. le system calls “classiche” di accesso ai file, che in Linux sono molto simili alle funzioni della libreria C standard

Il VFS deve implementare ambedue le modalità; in particolare la mappatura di una VMA è utilizzata, come abbiamo visto, anche dal SO per implementare i meccanismi fondamentali di esecuzione dei programmi.

Le system calls classiche

L'insieme di API classico per la gestione dei files permette fondamentalmente di svolgere le seguenti operazioni:

- creare e cancellare i file
- leggere e scrivere i file
- creare e cancellare i direttori

Modello classico di un file in Linux

Un file è costituito da una sequenza di byte memorizzati su disco. Un programma non può lavorare direttamente su tale contenuto ma deve prima trasferirlo in proprie variabili. 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 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 glibc 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 (pathname) e l'intero “permessi” serve a gestire i permessi di accesso; nella `open` il parametro 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 accessibile tramite quel descrittore; questa funzione è ovviamente analoga alla funzione `fclose` della libreria del C.

Nelle specifiche generali di Linux si avverte che l'esecuzione di `close` non garantisce che i dati che sono stati scritti in memoria vengano trasferiti immediatamente su disco, ma molti filesystem eseguono una effettiva scrittura su disco al momento della `close`. Per essere sicuri che i dati siano stati scritti su disco è necessario invocare `fsync` (file synchronization).

```
int fsync(int fd)
```

Questa operazione scrive su disco tutti i dati del file che sono stati modificati in memoria – questi aspetti verranno ripresi più avanti trattando l'implementazione del VFS.

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 l'array di caratteri 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.

Le operazioni `read` e `write` sono sequenziali, nel senso che ogni operazione si svolge a partire dalla posizione corrente lasciata dalla operazione precedente. 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 2.1 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 2.1 – gestione della posizione corrente con `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 2.1 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);
}
```



```
pos= 0
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pos= 20
ABCDEFGHIJKLMNOPQRSTUVWXYZ

pos= 5
FGHIJ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Figura 2.1 – Il programma fileoperations e la sua esecuzione

2.2 L'organizzazione complessiva dei file

Per permettere la gestione dei file e dei volumi (partizioni dei diversi dischi) i file sono organizzati in una struttura ad albero i cui nodi sono detti **cataloghi** (**directory**, cartelle, folder, ecc...).

I cataloghi e i nomi dei file

Per semplificare la gestione dei file, i nomi dei file sono inseriti nei **cataloghi**. 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 o albero 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 2.2 è mostrata una struttura gerarchica di cataloghi e i pathname e il tipo dei file coinvolti.

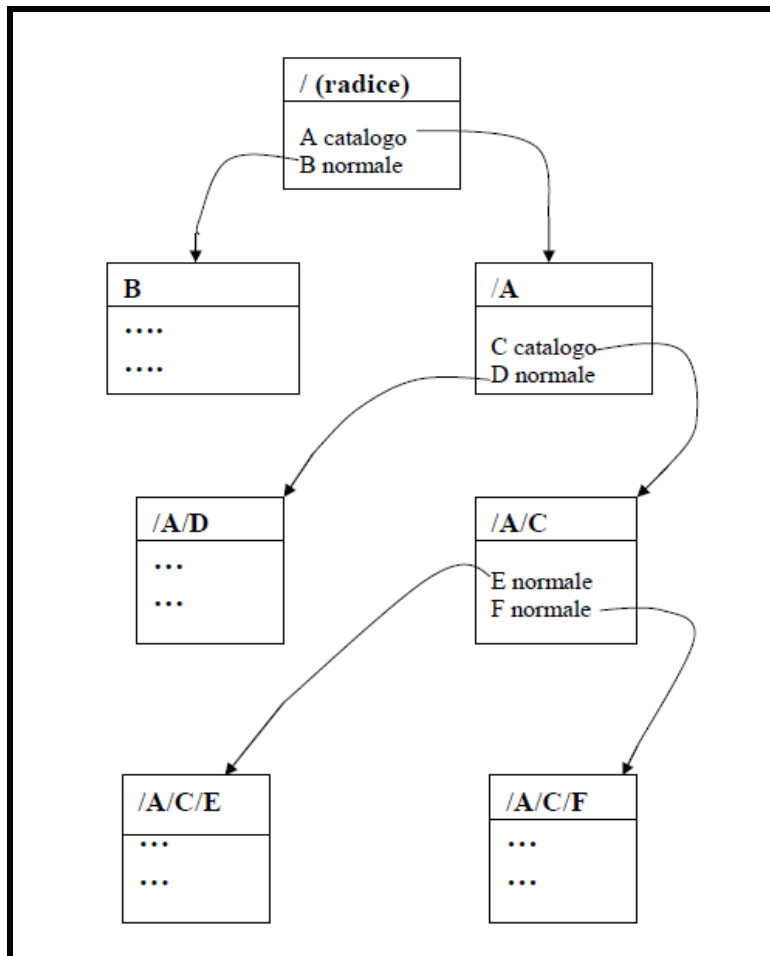


Figura 2.2 – Struttura dei cataloghi e dei pathname

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 riferimento al file, cancella il file stesso, rendendo disponibile lo spazio che occupava sul disco. 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.

La funzione `creat()` può creare solamente file normali; per creare dei cataloghi è necessario utilizzare una funzione diversa, `mkdir()`.

Periferiche e file speciali

In LINUX tutte le operazioni di ingresso/uscita sono svolte leggendo o scrivendo file, perché tutte le periferiche, compresi il video e la tastiera, sono considerati come file. In particolare, le periferiche sono accedute dal programma attraverso il filesystem 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 (video + tastiera) 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 nelle configurazioni più usuali 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, perché, 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. Molte funzioni di sistema utilizzano tali descrittori, ad esempio `printf()` scrive su standard output.

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 la funzione `dup` 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 questa funzione, *anche se un file è associato a più di un descrittore, il suo indicatore di posizione corrente è comunque unico*. Si veda il manuale per i dettagli della funzioni `dup`.

L'amministratore del sistema deve configurare tutte le periferiche e stabilire come organizzare i dati sui diversi dispositivi.

Le periferiche possono essere gestite da driver a carattere oppure a blocchi. Talvolta lo stesso dispositivo fisico, ad esempio un disco, può essere visto come una periferica a carattere e quindi scritto e letto direttamente dai programmi che lo utilizzano, oppure come periferica a blocchi, la cui gestione è fatta da un FS; in tal caso esisteranno 2 file speciali associati allo stesso dispositivo fisico.

Al momento i dispositivi a blocchi sono essenzialmente i vari tipi di dischi e gli SSD.

Creazione dei file speciali

Per accedere una periferica si devono utilizzare i servizi del file system con riferimento al file speciale della periferica. Pertanto, per ogni periferica installata nel sistema deve esistere un corrispondente file speciale, normalmente nel direttorio `/dev`.

Ogni periferica installata in un sistema è identificata da una coppia di numeri, detti **major** e **minor**; tutte le periferiche dello stesso tipo condividono lo stesso major, mentre il minor serve a distinguere le diverse periferiche appartenenti allo stesso tipo.

Se elenchiamo il contenuto del direttorio `/dev` otteniamo un risultato tipo il seguente:

```
syttty      4, 0
tty1        4, 1
tty2        4, 2
```

nel quale la prima colonna contiene il nome del file speciale e la colonna normalmente utilizzata per indicare le dimensioni (size) del file contiene una coppia di numeri che sono il major e il minor.

I file speciali non possono essere creati con la funzione `creat()`, ma devono essere creati con una funzione che può essere utilizzata solo dall'amministratore del sistema; tale funzione è `mknod` (sintassi semplificata eliminando una serie di opzioni)

```
mknod(pathname, type, major, minor),
```

dove `pathname` è il nome del file speciale, `type` indica che tipo di file si vuole creare (`c`=character special file, `b`=block special file) e `major` e `minor` sono i numeri che si vogliono assegnare alla periferica. Normalmente, la funzione `mknod` è utilizzata tramite un corrispondente comando di shell, ad esempio:

```
>mknod /dev/tty4 c 4 5
```

potrebbe essere utilizzato per creare un file speciale di tipo carattere di nome `tty4`, contenuto nel direttorio `/dev` e associato ai numeri `major=4` e `minor=5`.

I Volumi

A differenza di Windows, nel quale ogni albero di direttori parte da un dispositivo tipicamente individuato da una lettera (esempio `C:`, `D:`), in LINUX esiste un unico albero con un'unica radice (indicata da `/`); diversi Volumi, cioè partizioni dotate del loro FS, sono rappresentati da nodi dell'albero detti **mount_point**. Il sottoalbero la cui radice è un `mount_point` descrive la struttura interna del volume, mentre la posizione del `mount_point` nell'albero più generale lo rende raggiungibile a partire da `/` (root).

Per inserire un nuovo volume nella struttura è necessario compiere 2 operazioni:

- associare un FS al volume (device)
- montare il volume in un opportuno `mount_point`

Per associare un FS su una partizione si usa il comando `mkfs` con l'opzione `-t` per indicare un FS diverso dal default:

```
sintassi: mkfs -t type device
```

```
esempio: mkfs -t ext3 /dev/hda1 (associa il FS di tipo ext3 al volume /dev/hda1 )
```

L'inserimento di un volume nell'albero generale dei File avviene tramite il comando `mount`:

```
mount device mount_point
```

```
esempio: mount /dev/hda1 /home (inserisce il volume nel catalogo /home)
```

Si osservi che `/dev/hda1` è il nome di un file speciale che rappresenta il volume `hda1` all'interno del catalogo `dev`. Montando tale volume nel catalogo `home` si indica che il volume `hda1` contiene una sua struttura di file interna accessibile dal direttorio `home`. Pertanto i dati di un file `F` contenuto in tale volume potranno risultano accessibili in 2 modi:

- normalmente leggendo il file nel direttorio `/home/...`
- leggendo il file speciale `/dev/hda1`

L'amministratore del sistema ha una discreta libertà di scelta dei FS che vuole utilizzare, tuttavia esistono dei vincoli oggettivi: ad esempio non si può montare il Network Filesystem (NFS) su un disco.

3 Il Modello del VFS

3.1 Aspetti generali

Il modello del VFS deve rappresentare due tipi di informazioni:

- le informazioni relativamente statiche contenuti nei file e cataloghi memorizzati nei diversi volumi; queste informazioni sono presenti sui dischi e quindi permangono anche dopo lo spegnimento del sistema e vengono caricate in memoria in base alle esigenze
- le informazioni dinamiche associate ai file e cataloghi aperti durante il funzionamento del sistema, ad esempio la posizione corrente di un file

Il modello del VFS si basa su un certo numero di strutture dati; semplificandolo un po' noi ci limiteremo a considerare le 3 seguenti strutture principali:

- `struct dentry` (`dentry` sta per "directory entry", cioè una singola registrazione in una directory) – ogni istanza di questa struttura rappresenta un catalogo nel VFS
- `struct inode` (`inode` sta per "index node") – ogni istanza di questa struttura rappresenta uno e un solo file fisicamente esistente nei volumi, e contiene i cosiddetti metadati del file
- `struct file` – ogni istanza di questa struttura rappresenta un file aperto nel sistema; concettualmente questa struttura associa a un file fisico rappresentato dal suo `inode` le informazioni dinamiche, in particolare la posizione corrente

Queste strutture dati contengono, oltre ai campi utilizzati per rappresentare i contenuti specifici, dei puntatori che permettono di collegarle in modo da rappresentare in forma di strutture dati le informazioni necessarie a caratterizzare lo stato corrente dei file nel modello del VFS. Le principali tra tali strutture dati sono:

- la **struttura dei cataloghi**
- la **struttura di accesso** dai processi ai file aperti

3.2 La struttura dei cataloghi nel modello VFS

Indipendentemente da come ogni FS memorizza le proprie informazioni relative ai cataloghi, nel modello VFS tali informazioni devono essere rappresentate come nell'esempio di figura 3.1, che fa riferimento alla stessa struttura di cataloghi rappresentata nell'esempio di modello d'utente di figura 2.2.

Questa struttura è costituita esclusivamente da istanze di `struct dentry` che rappresentano i nodi dell'albero dei direttori.

L'albero è realizzato inserendo in ogni `dentry` un puntatore al primo delle sue subdirectory (`d_subdirs`) e un puntatore al prossimo `dentry` "fratello" (cioè figlio dello stesso padre) – questo puntatore è chiamato impropriamente `d_child`.

I campi principali di `struct dentry` sono i seguenti:

```
struct dentry {
    struct inode *d_inode;
    struct dentry *d_parent;
    struct qstr d_name; // nome del file
    struct list_head d_subdirs; //puntatore al primo dir figlio
    struct list_head d_child; //puntatore al fratello nell'albero
    ...
}
```

Oltre ai puntatori minimi necessari a rappresentare la struttura dell'albero dei cataloghi, questa struttura contiene il nome del file o catalogo rappresentato (`d_name`), un puntatore al padre (`d_parent`) per semplificare la navigazione verso l'alto nell'albero e un puntatore al `inode` (`d_inode`) del file fisico utilizzato nella struttura di accesso.

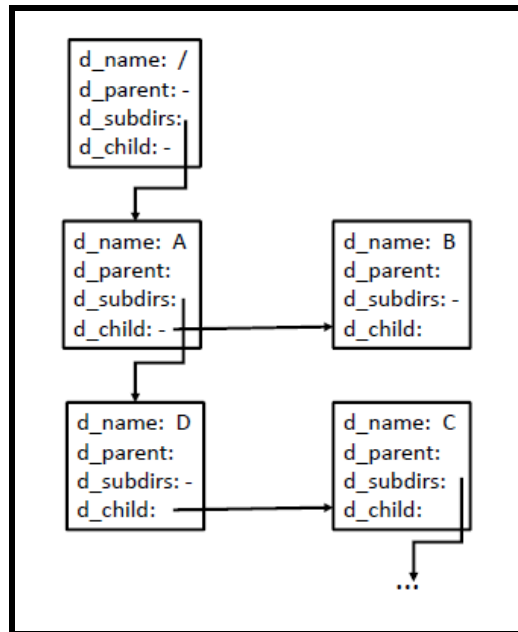


Figura 3.1

3.3 La struttura di accesso dai processi ai file aperti

Ogni descrittore di processo contiene 2 puntatori rilevanti ai fini dell'accesso ai file; riportiamo dal capitolo di introduzione al nucleo la parte rilevante di tale struttura:

```
struct task_struct {
    ...
    /* filesystem information */
    struct fs_struct *fs;
    /* open file information */
    struct files_struct *files;
    ...
}
```

Le istanze di `struct fs_struct` contengono i parametri che caratterizzano i singoli FS registrati nel sistema; il campo `fs` è un puntatore alla lista dei FS utilizzati dal processo e permette quindi di reperire, nei momenti in cui servono, le informazioni specifiche relative ai FS. Noi non tratteremo questo aspetto.

Il campo `files` è un puntatore a una struttura `files_struct`. Questa struttura contiene l'informazione relativa ai file aperti dal processo:

```
struct files_struct {
    ...
    int next_fd; //prossimo fd utilizzabile
    struct file * fd_array[NR_OPEN_DEFAULT]; //tabella file aperti
    ...
};
```

Il componente fondamentale di questa struttura è un array di dimensione fissa `fd_array` (che chiameremo anche “**tabella dei file aperti dal processo**”). Questo array contiene un elemento per ogni file aperto dal processo. Ogni elemento è un puntatore a un'istanza di `struct file` - si faccia attenzione a non confondere le due strutture:

- `struct files_struct` – è la struttura utilizzata nel descrittore del processo
- `struct file` – è la struttura utilizzata nel VFS per rappresentare i file aperti

Il **descrittore fd di un file F** è semplicemente un numero intero che identifica l'elemento all'interno della tabella dei file aperti del processo che si riferisce al file, cioè `fd_array[fd]` è utilizzato per accedere al file F.

La struttura `struct file` contiene tra l'altro i seguenti campi:

```
struct file {
    ...
    struct list_head f_list; //puntatore per la lista dei file aperti
    struct dentry *f_dentry; //riferimento al dentry usato in apertura
    loff_t f_pos; //posizione corrente
    int f_count //contatore dei riferimenti al file aperto
    ...
}
```

Il primo puntatore serve a collegare tutti i file aperti in una lista e non è utilizzato dalla struttura di accesso.

f_pos rappresenta la posizione corrente del file che viene aggiornata dinamicamente.

Il puntatore rilevante per costruire la struttura di accesso è `f_dentry`, che punta all'istanza di `dentry` che è stato utilizzato per aprire il file. Dato che, come visto sopra, un `dentry` punta al `inode` del file, possiamo definire la struttura di accesso di un processo P a un file aperto con descrittore FD come la catena di puntatori seguente:

`<descrittore di P> → files.fd_array[fd] → f_dentry → d_inode`

Quando un file viene aperto, viene allocata una nuova istanza di `struct file` e un puntatore a tale nuova istanza viene inserito nella prima posizione libera della tabella dei file aperti del processo; infine `open` restituisce l'indice di tale posizione (cioè il descrittore) al chiamante. Se non sono già presenti in memoria, vengono anche create le istanze delle strutture `dentry` e `inode`.

Numero di aperture contemporanee e contatore `f_count`

È importante osservare che diverse righe nell'ambito di uno stesso processo o di più processi possono puntare allo stesso file; in tal caso tutte le operazioni sui relativi descrittori **condividono la posizione corrente**.

Il contatore dei riferimenti `f_count` descrittori puntano contemporaneamente sul file.

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. Invece l'apertura indipendente da parte di un processo R di un file già aperto da parte di P crea una nuova istanza di `struct file`, con posizione corrente indipendente; tuttavia l'`i-node` è condiviso, perché si tratta dello stesso file fisico.

La struttura a valle dei descrittori, costituita da istanze di `dentry` e `inode`, è considerata eliminabile solo quando il contatore `f_count` diventa 0.

Esempio

In figura 3.2 è rappresentata la struttura di accesso costruita dalla seguente sequenza di operazioni (N.B. che altre sequenze potrebbero generare la stessa situazione):

1. il processo P ha aperto il file nominato `/dir1/file1` in un momento in cui il primo descrittore libero era nella riga 1; indichiamo il file fisico corrispondente con F1
2. il processo P ha duplicato il descrittore 1 in un momento in cui il primo descrittore libero era 3
3. il processo P ha letto 5 caratteri dal file con descrittore 1 (o 3)
4. il processo P ha eseguito una `fork` creando il processo Q
5. il processo R ha aperto il file nominato `/dir2/file2` ottenendo il descrittore 0; il file fisico corrispondente è lo stesso file F1; questo significa che in precedenza era stata usata il comando `link` per creare un secondo riferimento allo stesso file fisico
6. il processo R ha aperto un file denominato `/dir2/file3` corrispondente a un file fisico diverso da F1, ad esempio F2

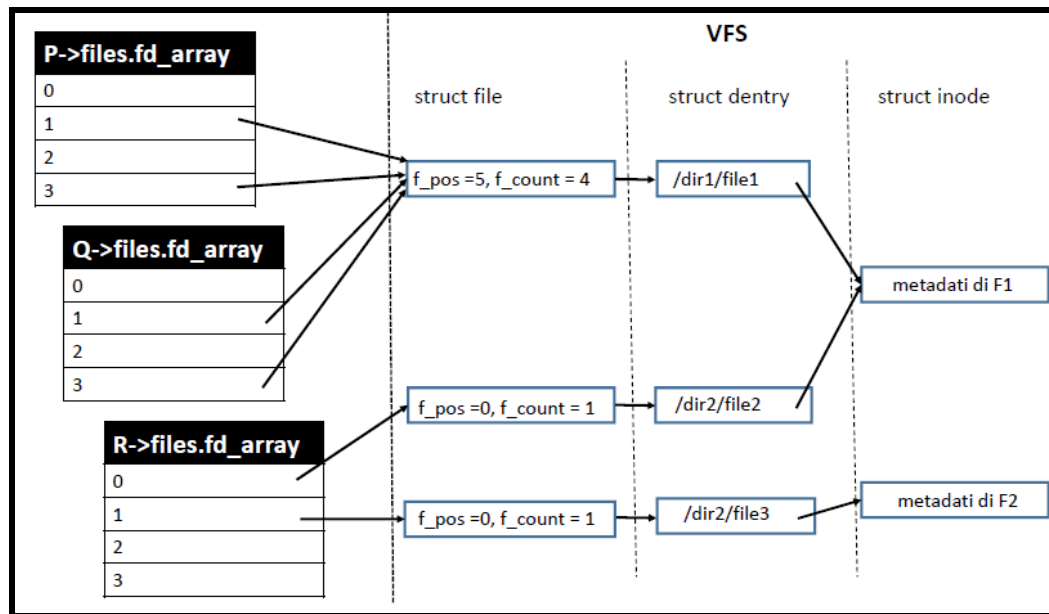


Figura 3.2

3.4 La struct inode

Un file è rappresentato nel VFS da un'istanza della `struct inode`. La corrispondenza tra file e (istanze di) inode è rigorosamente biunivoca.

Un inode contiene tutte le metainformazioni necessarie a caratterizzare il file; le principali sono riportate nella seguente definizione:

```
struct inode{
    loff_t i_size; //dimensione del file
    struct list_head i_dentry; //inizio della lista dei dentry del file
    struct super_block *i_sb; //superblocco del FS che lo gestisce
    struct inode_operations *i_op; //vedi sotto
    struct file_operations *i_fop;
    struct address_space *i_mapping; //struttura di mapping dei blocchi
    ...
}
```

Il significato di `i_size` è ovvio.

L'esistenza di una lista dei dentry (`i_dentry`) che fanno riferimento al file è dovuta al fatto che un file può avere più di un nome nella struttura dei cataloghi (vedi operazione `link` nel modello di utente).

Il superblocco di un FS è una struttura dati che contiene la definizione delle caratteristiche generali del FS stesso. Quindi il puntatore `i_sb` permette di risalire da un file al FS che lo gestisce.

Le strutture di tipo `..._operations i_op` e `i_fop` contengono i puntatori alle funzioni dello specifico FS che implementano le funzioni generali del VFS – sono quindi strutture simili a quelle delle classi di scheduling viste nella trattazione dello scheduler. Concettualmente queste 2 strutture dovrebbero risiedere solo nel superblocco, perché le funzioni di un FS sono condivise da tutti i file gestiti da tale FS, ma sono rese accessibili direttamente dal inode di ogni file per rendere più immediata la loro invocazione.

Le 2 strutture danno accesso rispettivamente alle funzioni del FS specifico per la gestione dell'organizzazione complessiva dei file e cataloghi e alle operazioni di accesso al singolo file:

- la `struct inode_operations` contiene puntatori alle implementazioni specifiche di operazioni quali `create`, `mknod`, `link`, `unlink`, `mkdir`, `rmdir`, ...

- la `struct file_operations` contiene puntatori alle implementazioni specifiche di operazioni quali `open`, `read`, `write`, `llseek`, ...

L'implementazione di queste funzioni appartiene allo specifico FS, ma deve ovviamente rispettare la struttura dei prototipi indicata in questa `struct`. Ogni FS è quindi libero di rappresentare un file e un catalogo sul dispositivo come vuole, ma deve essere in grado di realizzare le operazioni indicate nelle tabelle operando correttamente sulle strutture dati del VFS.

Il campo `struct address_space *i_mapping` contiene alcune informazioni fondamentali per realizzare le operazioni di trasferimento dei dati dal volume alla memoria.

3.5 Accesso ai dati di un file

Nel modello di utente l'accesso ai dati di un file avviene tramite le operazioni `read` e `write` di *lettura e scrittura di un certo numero di byte a partire dalla posizione corrente*, che entrano nel sistema come system calls `sys_read` e `sys_write`. La posizione corrente fa riferimento al file come una sequenza continua di byte. Questo riferimento deve essere trasformato in modo da essere utilizzabile per 2 scopi:

- trovare i dati sul Volume, che è organizzato in blocchi logici identificati da un LBA – per semplicità nel seguito supporremo che la dimensione dei blocchi sia di 1024 byte
- far transitare i dati dalla memoria, e in particolare dalla Page Cache, che è organizzata in pagine da 4096 byte

La lettura di un file è basata sulla pagina: il SO trasferisce sempre pagine intere di dati con ogni operazione. Se un processo esegue una system call `sys_read` anche di pochi byte il sistema esegue le seguenti operazioni:

1. determina la pagina del file alla quale i byte appartengono
2. verifica se la pagina è già contenuta nella Page Cache, in tal caso passa al punto 5
3. alloca una nuova pagina in Page Cache
4. riempie la pagina con la corrispondente porzione del file, caricando i blocchi necessari (4 se blocchi da 1K) dal volume
5. copia i dati richiesti nello spazio di utente all'indirizzo richiesto dalla system call

Trasformazione della posizione corrente in indirizzi sul volume (operazioni 1 e 4)

L'operazione 1 richiede la trasformazione della posizione corrente in un numero di pagina; l'operazione 4 richiede la trasformazione del numero di pagina negli LBA dei blocchi che costituiscono la pagina.

Un esempio di tali trasformazioni è fornito in figura 3.3, dove la posizione corrente è 5000. Le **pagine del file (FP)** sono numerate a partire da 0 e indicate con FP0, FP1, ecc...

La trasformazione della posizione corrente POS in un FPn può essere realizzata tramite una semplice divisione nel modo seguente, indicato utilizzando le convenzioni del linguaggio C:

$$FP = POS / 4096 \text{ (divisione all'intero)}$$

$$OFFSET = POS \% 4096 \text{ (l'operatore \% produce il resto della divisione)}$$

Nell'esempio di figura otteniamo quindi $FP = 1$, $offset = 904$

Una pagina può essere considerata costituita da N blocchi (4 nel nostro caso) i cui numeri, detti **FBA (File Block Address)** sono facilmente calcolabili; il primo è ottenuto moltiplicando il numero di pagina per N e i successivi sono in sequenza.

Nell'esempio di figura 3.3 i blocchi che costituiscono la pagina 1 sono FBA4, FBA5, FBA6 e FBA7.

I blocchi del volume che costituiscono la pagina sono quelli che memorizzano gli FBA della pagina. Tali blocchi non sono necessariamente consecutivi, perché non è detto che il FS abbia potuto o voluto memorizzare i dati di una pagina in blocchi consecutivi.

La corrispondenza tra FBA e LBA dipende dal modo in cui è organizzato il volume dal FS specifico; in figura tale corrispondenza è rappresentata come puntatori da FBA a LBA, ma in realtà i blocchi del file in generale contengono solo dati, non puntatori, e la struttura reale è quindi diversa e dipende da come il FS organizza il volume – vedremo un esempio nel capitolo relativo ai FS di tipo ext.

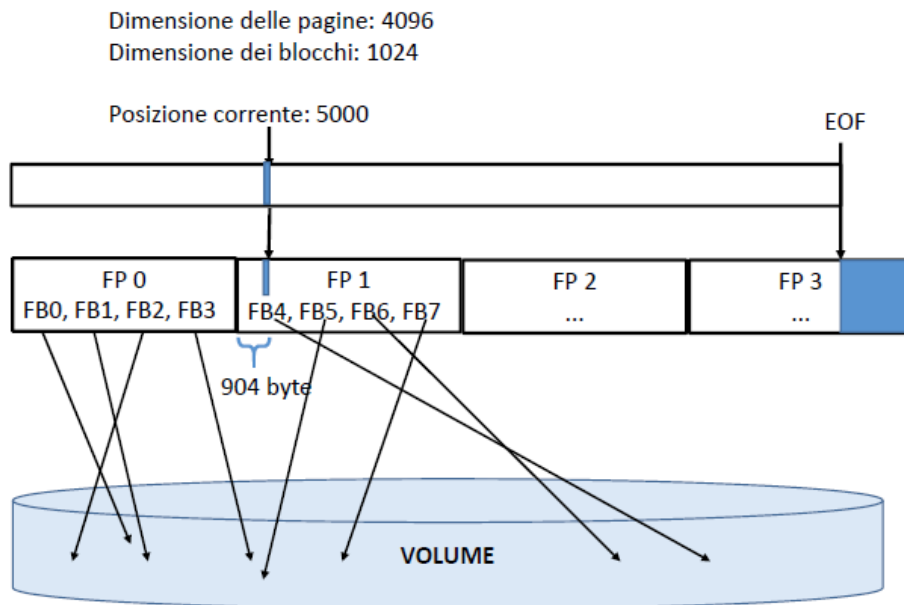


Figura 3.3

Operazioni delegate alla Page Cache (operazioni 2, 3, e 4)

Abbiamo visto trattando la gestione della memoria (capitolo M2) che la Page Cache è “un insieme di pagine fisiche utilizzate per rappresentare i file in memoria e un insieme di strutture dati ausiliarie e di funzioni che ne gestiscono il funzionamento. In particolare, le strutture ausiliarie contengono l’insieme dei descrittori delle pagine fisiche presenti nella cache; ogni *descrittore di pagina* contiene l’identificatore del file e l’offset sul quale è mappata, oltre ad altre informazioni, tra cui il *ref_count*. Inoltre, le strutture ausiliarie contengono un meccanismo efficiente (**Page_Cache_Index**) per la ricerca di una pagina in base al suo descrittore costituito dalla coppia <identificatore file, offset>.”

Possiamo ora precisare meglio il meccanismo utilizzato dalla Page Cache per determinare se una pagina di un file identificata dal suo numero FPn è già presente in memoria. La struttura dati utilizzata per questo scopo è la struttura `struct address_space` puntata dal campo `i_mapping` del inode del file.

I componenti principali della struttura sono i seguenti:

```
struct address_space {
    struct i_node host; //puntatore al inode che contiene questo mapping
    struct ... page_tree;
    struct ... a_ops //puntatori alle funzioni per operare sul mapping
    ...
}
```

L’elemento fondamentale di questa struttura dati è `page_tree`, una particolare struttura ad albero (radix tree) utilizzata per puntare a tutte le pagine della Page Cache relative a questo file. Dato un FP (numero di pagina nel file) questa struttura permette di determinare molto rapidamente se la pagina è già presente in Page Cache, e, in caso affermativo, di trovare il suo descrittore.

In sostanza, il meccanismo generico che avevamo chiamato **Page_Cache_Index** trattando della Page Cache è realizzato file per file tramite la struttura `address_space` puntata dal inode di ogni file: dato un numero di pagina relativo a un file FP la Page Cache opera nel modo seguente per verificare se tale pagina è già presente:

- accede al inode del file
- dal inode accede al page tree

- cerca nel page tree se esiste la pagina FP

Se la pagina non è presente è necessario procedere a caricarla. La struttura `a_ops` contiene operazioni specifiche del FS per accedere le pagine, in particolare `readpage` e `writepage`. Nella maggior parte dei casi queste funzioni invocano le corrispondenti funzioni del device driver che accede il dispositivo fisico, cioè il gestore a blocchi del volume. Ad esempio, la funzione che implementa `readpage` del inode di un file normale sa come localizzare le posizioni sui dispositivi fisici dei blocchi che corrispondono ad ogni pagina del file.

Si osservi a questo punto che il meccanismo descritto funziona in entrambe le situazioni principali di accesso ai file citate nel modello d'utente:

- supporta il meccanismo classico di lettura e scrittura dei file tramite `read` e `write`
- supporta il meccanismo delle aree virtuali

In ambedue i casi è infatti il numero di pagina del file ad alimentare l'individuazione della pagina di Page Cache che potrebbe contenere il dato utile; nel caso dell'accesso classico abbiamo appena visto che il numero di pagina nel file è derivato dalla posizione corrente, mentre nel caso delle VMA il numero di pagina nel file era la somma dell'offset della VMA rispetto al File sommato all'offset dell'indirizzo di pagina richiesto rispetto all'inizio della VMA stessa.

Gli aspetti che abbiamo trattato sono quelli di maggiore importanza nella interazione tra gestione della memoria e gestione dei file, perché i dati letti da un file vengono mantenuti nella Page Cache; altri aspetti da menzionare, che non approfondiremo, sono i seguenti:

- esistono anche due aree di memoria per memorizzare e conservare gli inode e i dentry, dette `inode_cache` e `dentry_cache`; queste aree occupano meno spazio rispetto alla Page Cache e la loro gestione viene ottimizzata separatamente (può infatti essere conveniente mantenere in memoria gli inode e i dentry anche quando la pagine dati di un file sono scaricate)
- esiste la possibilità di utilizzare pagine della Page Cache per memorizzare blocchi di un file che non corrispondono alle normali pagine dati del file – questa possibilità è sfruttata dai FS specifici per risolvere alcuni problemi che richiedono di memorizzare singoli blocchi)

3.6 Convenzioni per esercizi ed esercizi

Nella simulazione delle operazioni sui file gli eventi sono costituiti dall'esecuzione delle operazioni fondamentali sui file viste nel modello di utente, con parametri semplificati eliminando le numerose opzioni che non ci interessano, nel modo seguente

- `fd = Open(F)` - `fd` è il descrittore, `F` è il nome
- `Read(fd, num)` - `num` è il numero di caratteri da leggere
- `Write(fd, num)` - `num` è il numero di caratteri da scrivere
- `Lseek(fd, incr)` - `incr` è l'incremento da attribuire alla posizione corrente
- `Close(fd)` -

L'unica operazione per la quale modifichiamo anche la specifica di comportamento, anche in conformità a ciò che fanno molti FS, è la `close`, per la quale supporremo che ***i dati vengano scritti su disco se l'operazione di close riduce `f_count` a 0.***

Nelle simulazioni mostreremo principalmente lo stato della memoria secondo le convenzioni già viste trattando quell'argomento e il contenuto dei campi principali delle strutture dati mostrate in figura 3.2 (`f_pos` e `f_count`).

Inoltre vogliamo valutare il numero complessivo di accessi alle pagine del disco in lettura e scrittura.

Esercizio 1

Si consideri il seguente stato della memoria (è in esecuzione il processo P):

Parametri generali: MAX_FREE: 3 MIN_FREE: 2 MEM_SIZE: 8

===== Stato iniziale: =====

_____MEMORIA FISICA_____ (pagine libere: 5)

00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : ----
04 : ----	05 : ----
06 : ----	07 : ----

Indicare lo stato della memoria, la posizione corrente dei file e il numero totale di accessi a pagine del disco in lettura e scrittura dopo ognuno dei seguenti eventi:

1. `fd = open("F");`
2. `read(fd, 4100);`
3. `lseek(fd, -200);`
4. `write(fd, 4100);`
5. `read(fd, 4100);`
6. `close(fd);`

Soluzione

1)****processo P - Open *****

f_pos: 0 -- f_count: 1

2)****processo P - Read(fd,4100) *****

_____MEMORIA FISICA_____ (pagine libere: 3)

00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,0>
04 : <F,1>	05 : ----
06 : ----	07 : ----

f_pos: 4100 -- f_count: 1

Numero di accessi a pagine del DISCO: Lettura 2 Scrittura 0

3)****processo P - Lseek(fd,-200) *****

f_pos: 3900 -- f_count: 1

4)****processo P - Write(fd,4100) *****

_____MEMORIA FISICA_____ (pagine libere: 3)

00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,0> D
04 : <F,1> D	05 : ----
06 : ----	07 : ----

f_pos: 8000 -- f_count: 1

Numero di accessi a pagine del DISCO: Lettura 2 Scrittura 0

(le scritture sono avvenute sulle pagine in memoria, ma non sul disco – le pagine sono marcate D)

5)****processo P - Read(fd,4100) *****

_____MEMORIA FISICA_____ (pagine libere: 2)

00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,0> D
04 : <F,1> D	05 : <F,2>
06 : ----	07 : ----

f_pos: 12100 -- f_count: 1

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 0

6)**** processo P - Close(fd) *****

_____MEMORIA FISICA_____ (pagine libere: 2)

00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,0>
04 : <F,1>	05 : <F,2>
06 : ----	07 : ----

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 2

(in questo caso close causa la scrittura delle pagine su disco perché f_count diventa 0)

Esercizio 2

Come l'esercizio precedente, ma aggiungendo una ulteriore lettura prima della close:

```
1. fd = open("F");
2. read(fd, 4100);
3. lseek(fd, -200);
4. write(fd, 4100);
5. read(fd, 4100);
6. read(fd, 4100);
7. close(fd);
```

Soluzione

I primi 5 eventi come nell'esercizio precedente:

5)****processo P - Read(fd,4100) ****

MEMORIA FISICA (pagine libere: 2)			
00 : <ZP>	01 : Pc1/<X,1>		
02 : Pp0	03 : <F,0> D		
04 : <F,1> D	05 : <F,2>		
06 : ----	07 : ----		

f_pos: 12100 -- f_count: 1

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 0

6)****processo P - Read(fd,4100) ****

(interviene PFRA, che libera le pagine 3 e 4 scrivendole su disco perché dirty), poi viene caricata <F,3>)

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc1/<X,1>		
02 : Pp0	03 : <F,3>		
04 : ----	05 : <F,2>		
06 : ----	07 : ----		

f_pos: 16200 -- f_count: 1

Numero di accessi a pagine del DISCO: Lettura 4 Scrittura **2**

7)**** processo P - Close(fd) ****

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc1/<X,1>		
02 : Pp0	03 : <F,3>		
04 : ----	05 : <F,2>		
06 : ----	07 : ----		

Numero di accessi a pagine del DISCO: Lettura 4 Scrittura **2**

(close non ha bisogno di scrivere pagine, perché lo ha fatto PFRA)

Esercizio 3

Consideriamo 2 processi P (in esecuzione) e Q e il seguente stato iniziale della memoria

MAX_FREE: 3 MIN_FREE: 2 MEM_SIZE: 10

===== Stato iniziale: 2 processi in esecuzione =====

MEMORIA FISICA (pagine libere: 8)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : Pp0		
04 : ----	05 : ----		
...			

Eventi:

1. fd1 = open("F");
2. read(fd1, 4100);
3. lseek(fd1, -200);
4. write(fd1, 4100);
5. read(fd1, 4100);
6. contextSwitch("Q");
7. fd2 = open("F");
8. read(fd2, 4100);
9. close(fd2);
10. contextSwitch("P");
11. close(fd1);

Soluzione

I primi 5 eventi sono come negli esercizi precedenti:

5)****processo P - Read(F,4100) *****

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : Pp0		
04 : <F,0> D	05 : <F,1> D		
06 : <F,2>	07 : ----		
08 : ----	09 : ----		

File Aperto F in proc P f_pos: 12100 -- f_count: 1

Accessi a pagine del DISCO per file F: Lettura 3, Scrittura 0

6)*****ContextSwitch(Q)*****

7)****processo Q - Open *****

File Aperto F in proc Q f_pos: 0 -- f_count: 1

8)****processo Q - Read(F,4100) *****

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : Pp0 D		
04 : <F,0> D	05 : <F,1> D		
06 : <F,2>	07 : ----		
08 : ----	09 : ----		

File Aperto F in proc Q f_pos: 4100 -- f_count: 1

Accessi a pagine del DISCO per file F: Lettura 0, Scrittura 0

9)**** processo Q - Close(F) *****

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : Pp0 D		
04 : <F,0>	05 : <F,1>		
06 : <F,2>	07 : ----		
08 : ----	09 : ----		

Accessi a pagine del DISCO per file F: Lettura 0, Scrittura 2

**** processo P - Close(fd) *****

MEMORIA FISICA (pagine libere: 3)			
...	inalterata		

Accessi a pagine del DISCO per file F: Lettura 3, Scrittura 2

(la prima close ha già salvato le pagine del file)

Esercizio 4

Consideriamo un processo P in esecuzione e i primi 5 eventi come negli esempi precedenti, poi P esegue una fork.

Eventi

1. `fd1 = open("F");`
2. `read(fd1, 4100);`
3. `lseek(fd1, -200);`
4. `write(fd1, 4100);`
5. `read(fd1, 4100);`
6. `fork("Q"); + contextSwitch("Q");`
7. `read(fd1, 4100);`
8. `close(fd1);`
9. `contextSwitch("P");`
10. `close(fd1);`

Soluzione

Sono evidenziate le differenze rispetto al caso precedente.

5)****processo P - Read(fd,4100) *****

MEMORIA FISICA (pagine libere: 4)			
00 : <ZP>	01 : Pc1/<X,1>		
02 : Pp0	03 : <F,0> D		
04 : <F,1> D	05 : <F,2>		
06 : ----	07 : ----		
08 : ----	09 : ----		

File Aperto F in proc P f_pos: 12100 -- f_count: 1

Accessi a pagine del DISCO per file F: Lettura 3, Scrittura 0

6)***** fork e ContextSwitch(Q)*****

7)****processo Q - Read(fd,4100) *****

MEMORIA FISICA (pagine libere: 2)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : <F,0> D		
04 : <F,1> D	05 : <F,2>		
06 : Pp0 D	07 : <F,3>		
08 : ----	09 : ----		

File Aperto F in proc Q f_pos: 16200 -- f_count: 2

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura 0

(esiste una sola istanza della struct file che rappresenta F, con riferimenti dai 2 processi: la posizione corrente è condivisa tra i processi P e Q)

8)**** processo Q - Close(fd) *****

MEMORIA FISICA (pagine libere: 2)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0	03 : <F,0> D		
04 : <F,1> D	05 : <F,2>		
06 : Pp0 D	07 : <F,3>		
08 : ----	09 : ----		

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura **0**

(close non scrive le pagine D, perché f_count dopo la close vale 1)

9)***** ContextSwitch(P)*****

10)**** processo P - Close(fd) *****

MEMORIA FISICA (pagine libere: 2)			
00 : <ZP>	01 : Pc1/Qc1/<X,1>		
02 : Qp0 D	03 : <F,0>		
04 : <F,1>	05 : <F,2>		
06 : Pp0 D	07 : <F,3>		
08 : ----	09 : ----		

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura **2**

(questa close scrive le pagine D, perché f_count dopo la close vale 0)

4 Gli extended file systems ext2, ext3 e ext4

4.1 Introduzione

Gli extended file systems ext2, ext3 e ext4 sono i FS di default di LINUX. Tipicamente sui PC viene installato uno di questi.

ext3 e ext4 sono versioni migliorate ed estese di ext2 con elevata compatibilità, cioè un file scritto con ext2 può essere letto con uno dei successivi.

Alcune delle principali caratteristiche sono le seguenti

	max dim. file	max dim. partizione	principali differenze	anno di creazione
ext2	2 Tb	32 Tb		1993
ext3	2 Tb	32 Tb	ext2 + journalling	2001
ext4	16 Tb	1 Eb	ext3 + extent	2008

Il journalling consiste in un meccanismo transazionale che evita la creazione di file corrotti nel caso di chiusura anomala. Infatti, se il sistema viene chiuso in maniera anomala quando non tutte le strutture dati di un file sono state salvate su disco, tali strutture possono trovarsi in uno stato inconsistente. Noi non tratteremo questo aspetto ulteriormente.

Il meccanismo degli extent introdotto da ext4 costituisce una struttura di memorizzazione dei file diversa da quella di base di ext2 che è particolarmente indicata per file di grandi dimensioni utilizzati in modalità sequenziale, come i file di dati multimediali.

Nel mondo dei server vengono utilizzati anche altri FS con l'obiettivo di ottenere particolari prestazioni: Reiser, XFS, JFS

4.2. Il filesystem ext2

Il FS ext2 è nato nel contesto di LINUX e quindi le strutture dati che esso crea e mantiene sul Volume hanno una forte relazione con le strutture dati del VFS.

Un Volume gestito da ext2 è suddiviso in un certo numero di sottoinsiemi di blocchi detti **Block Groups**; per capire l'organizzazione complessiva conviene però iniziare trascurando questa nozione e immaginare che l'intero volume sia un unico Block Group. Successivamente introdurremo le modifiche necessarie.

4.2.1 Organizzazione di un Volume ext2 (senza Block Group)

Le principali strutture dati costruite da ext2 sul Volume sono le seguenti:

- **superblock**: contiene informazioni globali sul volume ed è eventualmente utilizzato in fase di boot del SO
- **tabella degli inode**: contiene tutti gli inode
- **inode**: contiene l'informazione relativa a un singolo file
- **directories**: i cataloghi presenti sul volume
- **blocchi dati**: contengono i dati che costituiscono i file dati

Il superblock è posizionato a un indirizzo prefissato ed è quindi raggiungibile dal FS; a partire dal superblock sono raggiungibili la tabella degli inode e la radice del (sotto)albero dei cataloghi contenuto nel volume - il volume può essere quello di primo livello, la cui radice coincide con la radice complessiva del sistema, oppure essere di un livello inferiore, la cui radice coincide con un mount-point interno all'albero complessivo).

La struttura dei cataloghi è realizzata creando dei file dati particolari che rappresentano le informazioni specifiche del catalogo, che devono essere sufficienti a riempire la struttura a dentry del VFS.

Nella Tabella degli inode gli inode sono memorizzati in sequenza e quindi sono reperibili in base al loro numero e alla loro dimensione (`inode_size`).

I blocchi dati sono inizialmente organizzati in una specie di albero, la lista libera, dal quale vengono prelevati per essere inseriti nei file; a un dato istante quindi ogni blocco dati appartiene a un file oppure alla lista libera. Quando un file viene eliminato oppure decresce liberando blocchi, i blocchi liberati tornano nella lista libera.

Tutti questi aspetti sono abbastanza ovvi e non vengono approfonditi qui; ci limitiamo ad approfondire come un inode rappresenta un file e permette di trovare i suoi blocchi sul volume e di leggerli.

Contenuto di un i-node

Tutta l'informazione generale relativa a un file è contenuta nel suo inode. Un file esiste infatti nel sistema quando esiste il suo inode. Il riferimento fisico a un file è costituito dal suo numero di inode; ad esempio, nei cataloghi sono contenute coppie <nome file, numero di inode 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) e che supportano l'accesso al file

Accesso ai Blocchi del File

Il FS deve memorizzare i blocchi dei file FBA in corrispondenti blocchi del volume LBA e deve possedere un meccanismo per trasformare un FBA in un LBA.

La dimensione dei blocchi può essere compresa tra 1Kb e 64Kb, ma è soggetta anche al limite costituito dalla dimensione delle pagine nell'architettura HW considerata, perché un blocco deve poter essere contenuto in una pagina; nell'architettura x64 sono quindi possibili solo 3 valori: 1, 2 o 4 Kb.

La struttura di accesso ai blocchi dei file nel FS ext2 è basata su 15 puntatori (Figura 4.1): 12 puntatori diretti, 1 puntatore indiretto semplice, un puntatore indiretto doppio e un puntatore indiretto triplo. Il significato di questi puntatori è reso evidente dalla figura.

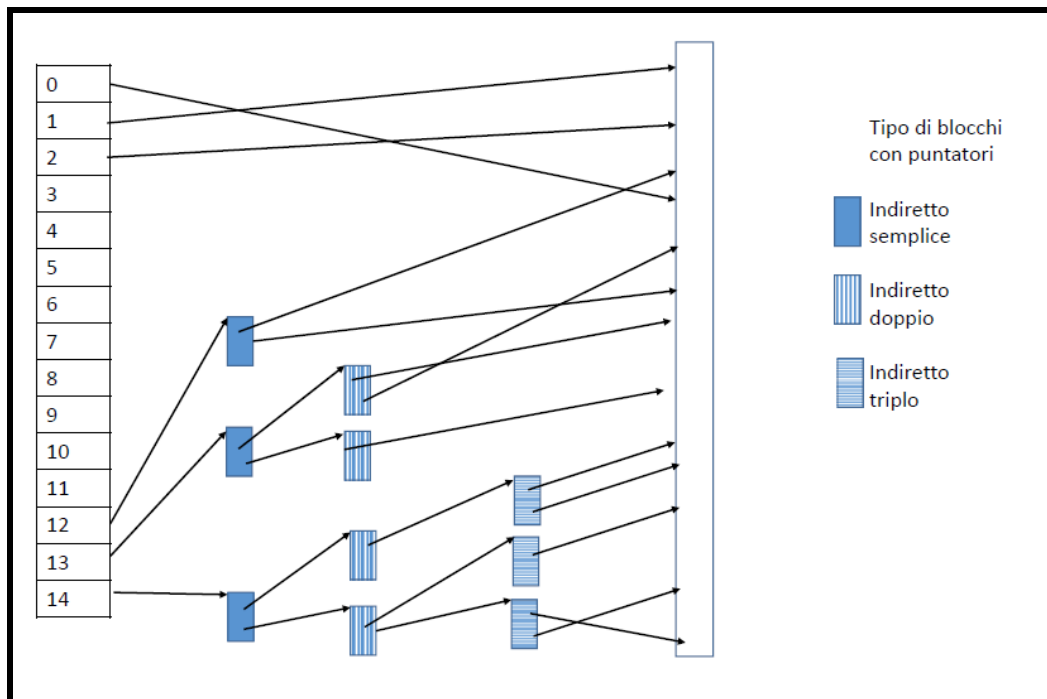


Figura 4.1

La dimensione dei puntatori ai blocchi è di 32 bit, quindi 4 byte.

La dimensione massima di un file dipende dalla dimensione dei blocchi: se b è la dimensione di un blocco (in byte) allora la massima dimensione di un file è determinata dalla seguente formula:

$$((b/4)^3 + (b/4)^2 + b/4 + 12) * b$$

dove $b/4$ è il numero di puntatori ai blocchi contenuti in un blocco di indizione.

L'obiettivo di questo meccanismo è di rendere efficiente l'accesso ai file piccoli (fino a 12 blocchi) rendendo i loro blocchi immediatamente accessibili e di incrementare l'onere dell'accesso progressivamente passando a file sempre più grandi.

4.2.2 La suddivisione del Volume in Block Group

Consideriamo adesso la suddivisione del volume in Block Group – un block group è una porzione di blocchi caratterizzata da un intervallo continuo di LBA. Se consideriamo che la numerazione dei blocchi rappresentata dagli LBA rifletta in una certa misura la “vicinanza” tra i blocchi, nel senso che LBA numericamente più vicini lo siano anche in termini di tempi di accesso, allora un block group costituisce un insieme di blocchi che si possono accedere insieme più rapidamente di un insieme casuale di blocchi.

E' compito del dispositivo e del suo driver fare in modo che questa ipotesi sia verificata.

La suddivisione in Block Group mira a supportare la memorizzazione di informazioni correlate tra loro in blocchi contigui in base all'ipotesi che informazioni correlate vengano accedute insieme.

La dimensione dei block group è determinata dalla possibilità di mantenere una mappa di bit per l'intero gruppo all'interno di un unico blocco, ovvero dal numero di bit di un blocco; ad esempio, se $b=1\text{Kb}$, la dimensione del gruppo è $8*1\text{K} = 8192$ blocchi, se $b=4\text{k}$ la dimensione è 32768.

I gruppi sono numerati partendo da 0 e sono consecutivi, senza buchi tra di loro.

Il superblock del volume è logicamente unico, ma può essere replicato in molti block group per ragioni di affidabilità.

La Tabella degli inode è logicamente unica, ed è suddivisa in parti uguali nei diversi block group. Nel superblocco è memorizzato il parametro `inodes_per_group` che indica il numero di inode assegnato ad ogni porzione di tabella degli inode. Ovviamente il numero complessivo di inode del volume, cioè il numero massimo di file memorizzabile, è dato dal prodotto di `inodes_per_group` per il numero di block group, e quindi dipende dalla dimensione dei blocchi.

Dato che ogni gruppo contiene nella propria porzione di tabella degli inode un numero prefissato di inode, è possibile risalire da un numero di inode (`inode_num`) al gruppo bg nella cui tabella è definito, tramite la seguente trasformazione (N.B. gli inode sono numerati da 1, non da 0):

$$\text{bg} = (\text{inode_num} - 1) / \text{inodes_per_group}$$

Lo specifico inode si troverà nella porzione di tabella degli inode del gruppo nella posizione indicata dallo spiazzamento seguente

$$\text{offset} = [(\text{inode_num} - 1) \% \text{inodes_per_group}] * \text{inode_size}$$

Al fine di memorizzare informazioni correlate tra loro in blocchi contigui in base all'ipotesi che informazioni correlate vengano accedute insieme, il FS tenta di allocare tutti i blocchi di un file nello stesso block group del catalogo che lo riferisce.

Ad esempio, con blocchi da 1Kb e quindi gruppi da 8196 blocchi, i file minori di 8Mb possono essere allocati in un unico gruppo, invece di essere dispersi sul disco.

Riassumendo, ogni gruppo contiene l'informazione necessaria alla gestione dei propri blocchi:

- copia del superblock (opzionale, per affidabilità)
 - BGDT- block group descriptor table (globale, ridondante)
 - block usage bitmap: ogni bit indica se il corrispondente blocco è libero oppure utilizzato (deve occupare un solo blocco, e spiega il dimensionamento indicato sopra)
 - i-node usage map: ogni bit indica se il corrispondente i-node è libero o utilizzato (deve utilizzare un solo blocco, quindi il numero di i-node ha la stessa limitazione)
 - porzione della tabella degli i-node, contenente `inodes_per_group` inode
 - blocchi dati (il meccanismo di allocazione dei blocchi tenta di allocare i blocchi dati di un file nello stesso gruppo che contiene il suo inode)
-

4.4 Il meccanismo degli extent in ext4

Un extent è un *insieme di blocchi logicamente contigui all'interno del file e tenuti contigui anche sul dispositivo fisico*.

La rappresentazione di un extent richiede tre parametri:

- il blocco del file (FBA) di inizio del extent
- la dimensione del extent
- il blocco del volume (LBA) di inizio del extent

Ad esempio, in figura 4.2 è rappresentata la struttura di un file di 1700 blocchi tramite 2 extent:

- il primo extent fa riferimento ai primi 700 blocchi del file (FBA di inizio = 0, dimensione = 700); tali FBA sono memorizzati in altrettanti blocchi del volume a partire da LBA = 1500
- il secondo extent fa riferimento ai successivi 1000 blocchi del file (FBA di inizio = 700, dimensione = 1000); tali FBA sono memorizzati in altrettanti blocchi del volume a partire da LBA = 4000

In questo modo non è più necessario mantenere un puntatore per ogni blocco. Nell'esempio appena considerato si sono risparmiati più di 1700 puntatori ai singoli blocchi che sarebbero stati necessari con la struttura normale.

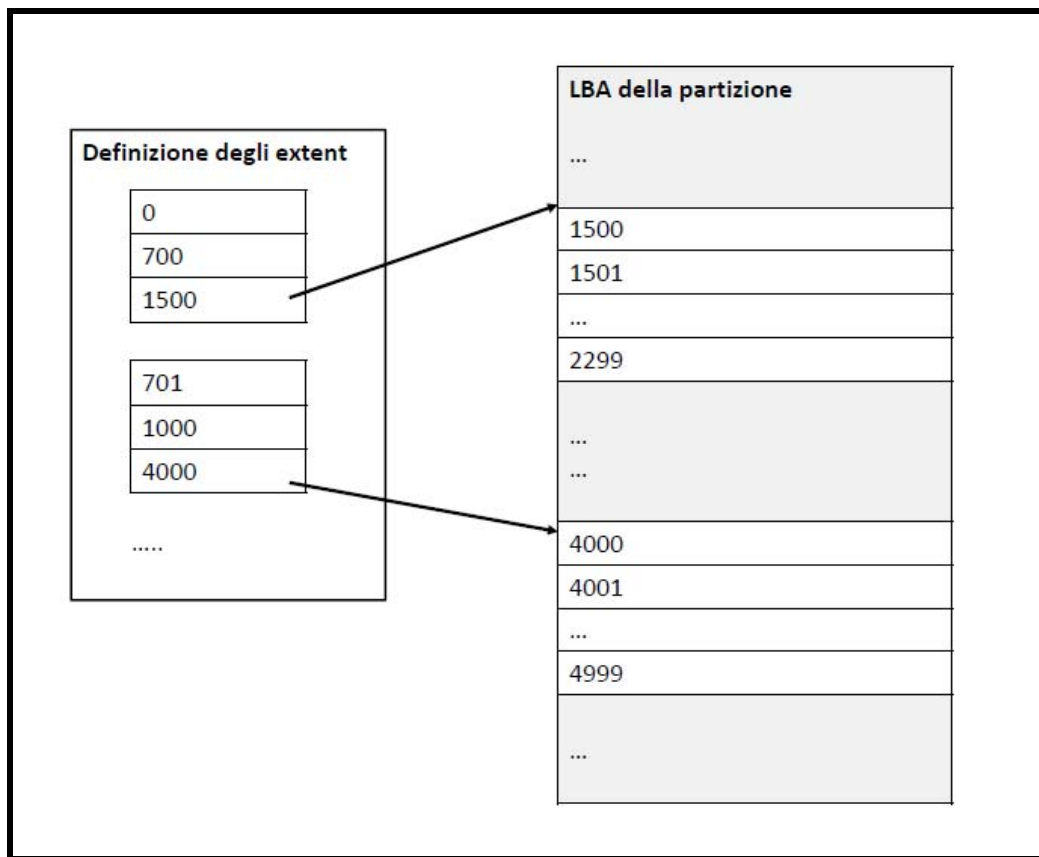


Figura 4.2

Questo meccanismo ha i seguenti vantaggi, significativi quando si memorizzano file molto grandi:

- riduce il numero di puntatori necessari (in realtà, questo risparmio di spazio non è molto significativo, perché lo spazio occupato dai puntatori è comunque una piccola frazione di quello occupato dai dati)
- migliora le prestazioni, perché non richiede la gestione della struttura dei puntatori anche indiretti

- tende a supportare l'allocazione contigua dei file, che è molto utile specialmente per file utilizzati sequenzialmente
- aumenta la dimensione massima dei file fisicamente mappabili

5 Driver a Carattere

5.1 Introduzione

I gestori di periferiche (**device drivers**) sono dei componenti il cui funzionamento, a differenza di quello del filesystem, è fortemente intrecciato con quello del nucleo del sistema. Come abbiamo visto, il modo in cui è scritto un gestore di periferiche mette in luce e motiva molti aspetti di organizzazione del nucleo del sistema.

Tra tutti i componenti di un sistema operativo i gestori di periferiche sono quello che richiede più spesso di essere modificato o esteso per trattare nuovi tipi di periferiche. Pertanto, si tratta dell'area dei sistemi operativi sulla quale capita più spesso di dover lavorare. D'altra parte, la comprensione del funzionamento del nucleo è fondamentale per scrivere un buon gestore di periferica. Infatti, il gestore adatta una specifica periferica a funzionare nel contesto del sistema operativo.

Ogni volta che un nuovo tipo di periferica deve essere gestito da un sistema operativo è necessario che venga realizzato il relativo gestore. Un gestore di periferica è quindi un componente software relativo ad un specifica coppia <sistema operativo/ tipo di periferica>. Chi deve realizzare un gestore di periferica deve quindi conoscere sia le caratteristiche Hardware della periferica da gestire, sia il modo in cui il gestore deve interfacciarsi con il resto del sistema operativo considerato. In genere, per ogni sistema operativo esiste un manuale dal tipo "guida alla scrittura di un gestore di periferica", che spiega come affrontare quest'ultimo aspetto.

Nel seguito ci occuperemo di questo aspetto relativamente al sistema operativo LINUX, cioè delle caratteristiche che deve possedere un generico gestore di periferica per LINUX.

In LINUX deve esistere uno specifico gestore per ogni tipo di periferica installata: un gestore del disco fisso, un gestore di USB, un gestore dei CD, un gestore dei terminali, ecc...

I gestori sono suddivisi in gestori a carattere (**character device driver**) e gestori a blocchi (**block device driver**).

5.2 File speciali e identificazione dei gestori

Per accedere una periferica si devono utilizzare i servizi del file system con riferimento al file speciale della periferica. Pertanto, per ogni periferica installata nel sistema deve esistere un corrispondente file speciale, normalmente nel direttorio /dev.

Ogni periferica installata in un sistema è identificata da una coppia di numeri, detti **major** e **minor**; tutte le periferiche dello stesso tipo, cioè gestite dallo stesso gestore, condividono lo stesso major, mentre il minor serve a distinguere le diverse periferiche appartenenti allo stesso tipo.

Se elenchiamo il contenuto del direttorio /dev otteniamo un risultato tipo il seguente:

```
sys tty      4, 0
tty1         4, 1
tty2         4, 2
```

nel quale la prima colonna contiene il nome del file speciale e la colonna normalmente utilizzata per indicare le dimensioni (size) del file contiene una coppia di numeri che sono il major e il minor.

I file speciali non possono essere creati con la funzione `creat()`, ma devono essere creati con una funzione che può essere utilizzata solo dall'amministratore del sistema (root); tale funzione è

```
mknod(pathname, type, major, minor),
```

dove `pathname` è il nome del file speciale, `type` indica che tipo di file si vuole creare (`c`=character special file, `b`=block special file) e `major` e `minor` sono i numeri che si vogliono assegnare alla periferica. Normalmente, la funzione `mknod` è utilizzata tramite un corrispondente comando di shell, ad esempio:

```
>mknod /dev/tty4 c 4 5
```

potrebbe essere utilizzato da root per creare un file speciale di tipo carattere di nome `tty4`, contenuto nel direttorio /dev e associato ai numeri `major=4` e `minor=5`.

I valori `major` e `minor` sono posti nel i-node del file speciale, che ovviamente non contiene puntatori ai dati.

5.3 Le routine del gestore di periferica

Le principali funzioni svolte da un gestore di periferica in LINUX sono le seguenti:

- inizializzare la periferica alla partenza del sistema operativo
- porre la periferica in servizio o fuori servizio
- ricevere dati dalla periferica
- inviare dati alla periferica
- scoprire e gestire gli errori
- gestire gli interrupt della periferica

Il gestore è sostanzialmente costituito da un insieme di routine che vengono attivate dal nucleo al momento opportuno, e dalla routine di interrupt che viene attivata dagli interrupt della periferica.

In ogni gestore esiste una tabella, che chiameremo **tabella delle operazioni**, che contiene puntatori alle funzioni del gestore stesso; il tipo di tale tabella è `struct file_operations`. Tale tipo, la cui definizione è mostrata parzialmente in figura 5.1, contiene molti elementi, dei quali solo alcuni sono utilizzati dalla maggior parte dei gestori. Alcune funzioni saranno discusse più avanti.

```
struct file_operations{
    int (*lseek) ( );
    int (*read) ( );
    int (*write) ( );
    ...
    int (*ioctl) ( );
    ...
    int (*open) ( );
    void (*release) ( );
    ...
}
```

Figura 5.1 – La struttura `file_operations` (parziale) di un gestore a carattere

All'avviamento del sistema operativo viene attivata una funzione di inizializzazione per ogni gestore di periferica installato. Tale funzione di inizializzazione, dopo aver svolto le operazioni necessarie di inizializzazione, restituisce al nucleo un puntatore alla propria tabella delle operazioni.

Il nucleo possiede al proprio interno due tabelle, che chiameremo **tabella dei gestori a carattere** e **tabella dei gestori a blocchi**.

Nella Tabella dei gestori opportuna il nucleo inserisce l'indirizzo della tabella delle operazioni di un gestore in posizione corrispondente al major del tipo di periferica gestito dal driver. In questo modo, dopo l'inizializzazione, il nucleo possiede una struttura dati come quella mostrata in figura 5.2, che permette, dato un major, di trovare l'indirizzo di una qualsiasi funzione del corrispondente driver.

Il meccanismo per l'utilizzazione dei gestori a blocchi è più complesso, perché passa attraverso la Page Cache e mira a ottimizzare l'ordine delle operazioni di accesso al dispositivo.

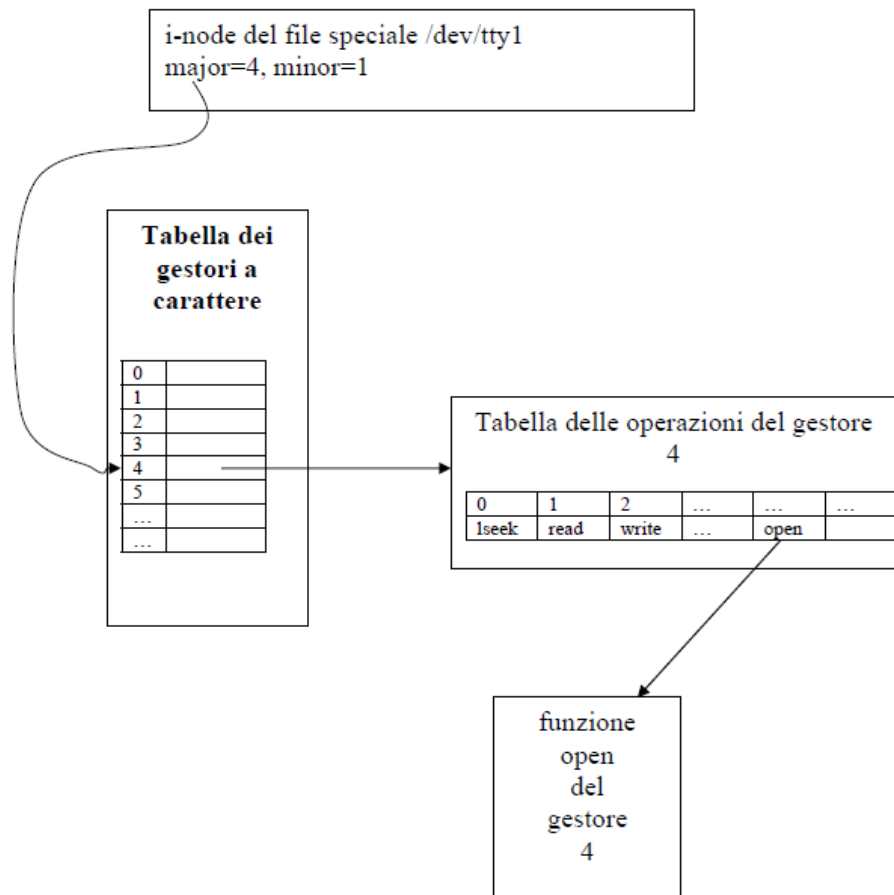


Figura 5.2 – Tabelle utilizzate dal sistema per indirizzare una funzione di un gestore

Come esempio di uso del meccanismo consideriamo un processo che invoca la funzione `open(/dev/tty1)`.

Il filesystem accede al file `/dev/tty1` e scopre che si tratta di un file speciale a carattere contenente i numeri `major=4` e `minor=1`. Il filesystem allora accede alla tabella dei gestori a carattere in posizione 4 e vi trova l'indirizzo della tabella delle operazioni di quel gestore. A questo punto il filesystem può attivare la funzione `open` di quel gestore, passandole come parametro il `minor` che aveva precedentemente trovato, in modo che la funzione `open` sappia quale è il terminale da aprire.

5.4 Principi di funzionamento di un gestore di periferica

Abbiamo visto come il sistema operativo possa indirizzare le funzioni di un gestore. Vediamo ora alcuni aspetti fondamentali del funzionamento di un gestore sotto LINUX.

Normalmente, dopo l'inizializzazione del sistema, un gestore di periferica viene attivato solamente quando un processo richiede qualche servizio relativo alla periferica. In questo caso la funzione del gestore che viene attivata opera nel contesto del processo che ne ha richiesto il servizio, e quindi l'eventuale trasferimento di dati tra il gestore e il processo non pone particolari problemi. Tuttavia, dato che le periferiche come abbiamo visto operano normalmente tramite interrupt, quando un processo P richiede un servizio a una periferica X, se X non è pronta il processo P viene posto in stato di attesa, e un diverso processo Q viene posto in esecuzione. Sarà l'interrupt della periferica a segnalare il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto. Quindi, *quando si verificherà l'interrupt della periferica X il processo in esecuzione sarà Q (o un altro processo subentrato a Q), ma non P*, cioè non il processo in attesa dell'interrupt stesso.

Questo aspetto è fondamentale nella scrittura di un gestore di periferica, perché implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere non possono essere trasferiti al processo interessato, che non è quello corrente, ma devono essere conservati nel gestore stesso.

L'esecuzione dell'operazione è descritta nel seguito con riferimento allo schema di figura 5.3. Il buffer che compare in tale figura è una zona di memoria appartenente al gestore stesso, non è un buffer generale di sistema. Supponiamo che tale buffer abbia una dimensione di B caratteri. Le operazioni svolte saranno le seguenti:

1. Il processo richiede il servizio tramite una funzione *write()* e il sistema attiva la routine *X.write()* del gestore;
2. la routine *X.write* del gestore copia dallo spazio del processo nel buffer del gestore un certo numero C di caratteri; C sarà il minimo tra la dimensione B del buffer e il numero N di caratteri dei quali è richiesta la scrittura;
3. la routine *X.write* manda il primo carattere alla periferica con un'istruzione opportuna;
4. a questo punto la routine *X.write* non può proseguire, ma deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il prossimo; per non bloccare tutto il sistema, *X.write* pone il processo in attesa creando una *waitqueue* WQ e invocando la funzione *wait_event_interruptible(WQ, buffer_vuoto)* – dove *buffer_vuoto* è una variabile booleana messa a true ogni volta che il buffer è vuoto
5. quando la periferica ha terminato l'operazione, genera un interrupt;
6. la routine di interrupt del gestore viene automaticamente messa in esecuzione; se nel buffer esistono altri caratteri da stampare, la routine di interrupt esegue un'istruzione opportuna per inviare il prossimo carattere alla stampante e termina (IRET); altrimenti, il buffer è vuoto e si passa al prossimo punto;
7. la routine di interrupt, dato che il buffer è vuoto, risveglia il processo invocando la funzione *wake_up(WQ)*, cioè passandole lo stesso identificatore di *waitqueue* che era stato passato precedentemente alla *wait_event_interruptible*;
8. *wake_up* ha risvegliato il processo ponendolo in stato di pronto; prima o poi il processo verrà posto in esecuzione e riprenderà dalla routine *X.write* che si era sospesa tramite *wait_event_interruptible*; se esistono altri caratteri che devono essere scritti, cioè se N è maggiore del numero di caratteri già trasferiti nel buffer, *X.write* copia nuovi caratteri e torna al passo 2, ponendosi in attesa, altrimenti procede al passo 9;
9. prima o poi si deve arrivare a questo passo, cioè i caratteri già trasferiti raggiungono il valore N e quindi il servizio richiesto è stato completamente eseguito; la routine *X.write* del gestore esegue il ritorno al processo che la aveva invocata, cioè al modo U

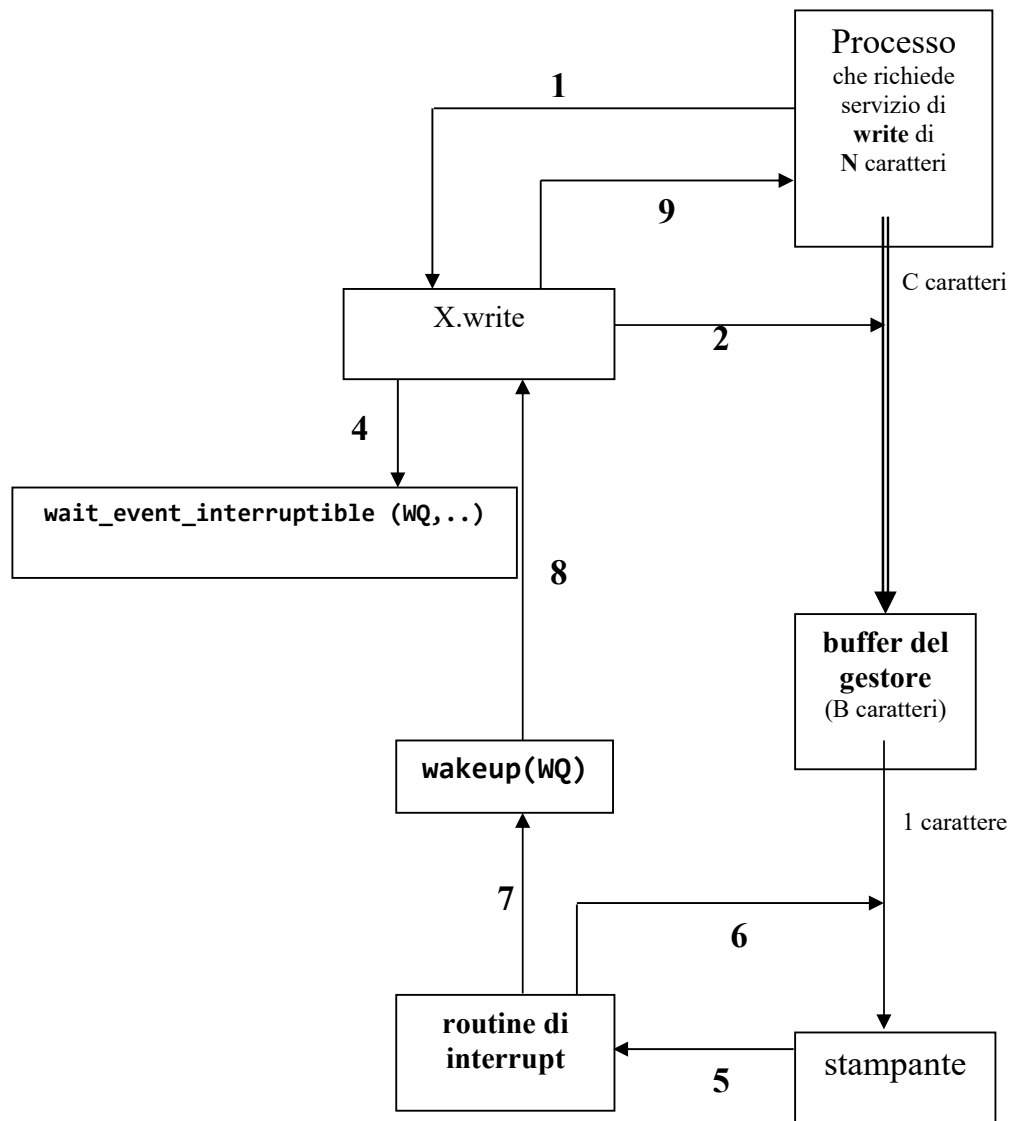


Figura 5.3 – Esecuzione di un servizio di scrittura

5.5 Le funzioni di un gestore a carattere

Le principali funzioni di un gestore a carattere sono `open()`, `release()`, `read()`, `write()` e `ioctl()`. Vediamo brevemente cosa fanno.

open

Le tipiche funzioni di questa routine del gestore, che viene ovviamente invocata quando un processo richiede un servizio `open` relativo ad un file speciale, sono le seguenti:

verificare che la periferica esiste ed è attiva (on line);

se la periferica deve essere utilizzata da un solo processo alla volta, `open` dovrebbe controllare una apposita variabile del gestore: se la variabile indica “periferica aperta”, `open` restituirà un opportuno codice di errore, altrimenti `open` porrà la variabile al valore “periferica aperta”;

release

Se la periferica deve essere utilizzata da un solo processo alla volta, la funzione `release` viene chiamata quando questo esegue una `close` del file speciale, altrimenti solamente quando l’ultimo processo che ha aperto la periferica esegue la `close`. La principale funzione di questa routine è la verifica della terminazione effettiva delle operazioni, cioè che ad esempio non ci siano ancora caratteri da stampare in un buffer; in tal caso tipicamente la `release` pone il processo in attesa di tale terminazione.

read

La routine `read`, invocata in conseguenza della richiesta di un servizio `read` sul file speciale, che deve essere già aperto, opera in maniera simmetrica a quanto è stato descritto relativamente alla `write` nell’esempio trattato sopra: `read` deve leggere un carattere dalla periferica se disponibile e poi porre il processo in attesa su un evento opportuno; al risveglio `read` trasferirà i caratteri al processo ed eventualmente, se i caratteri trasferiti non sono sufficienti, tornerà in attesa.

Esercizio: disegnare uno schema simile a quello di figura 5.3 per la funzione `read` di N caratteri dalla tastiera.

write

Il funzionamento della routine `write` è stato già analizzato sopra.

ioctl

Questa routine deve permettere di svolgere operazioni speciali su una periferica; le operazioni speciali sono operazioni che hanno senso solamente per quel particolare tipo di periferica, e non appartengono, a differenza delle operazioni viste sopra, alle operazioni standard svolte su tutte le periferiche (ad esempio, cambiare la velocità di trasmissione di una scheda di rete, oppure riavvolgere un nastro magnetico, ecc...)

La routine `ioctl` di un gestore viene attivata quando un processo invoca il servizio `ioctl()` relativamente al suo file speciale. Il servizio `ioctl` non è stato trattato nella programmazione di sistema, perché molto specificamente rivolto alla gestione delle periferiche. Il suo formato generale è

```
int ioctl(int fd, int comando, int param)
```

Per quanto riguarda la specifica dei comandi e parametri esistono molte varianti, che dipendono sia dagli standard, sia dalle periferiche. Si rimanda quindi al manuale per i dettagli.

La routine `ioctl` di un gestore riceve dal corrispondente servizio il comando e i parametri. In pratica, è il progettista del gestore a indicare la specifica dei servizi ottenibili tramite il servizio `ioctl`; questo servizio è quindi uno strumento per permettere ad un processo di invocare, attraverso il file system, una generica routine del gestore passandole dei parametri.
