Memoria

Gabr1313

December 17, 2023

Contents

Chapter 1

Il File System

Fornisce un livello di astrazione (**Modello d'utente**) per consentire l'accesso alle periferiche (file speciali).

1.1 Drivers

Le tipologie di periferiche sono molteplici, e queste evolvono nel tempo. Linux permette quindi di aggiungere dei moduli software al SO: i **device drivers**.

Questi vengono caricati solo quando necessari. Il meccanismo di inserimento nel sistema di nuovo software è chiamato `kernel_modules`. A un livello astratto i driver devono avere un comportamento omogeneo.

I driver si dividono in due categorie:

- **Driver a carattere**: le operazioni sono eseguite, ordinatamente, direttamente dai programmi che le richiedono.
- **Driver a blocchi**: il sistema pone le sue richieste in una coda, gestita da un FS. Il driver la può modificare così da ottimizzare le prestazioni.

1.2 Virtual File System e VFS

Linux è in grado di gestire una moltitudine di file system (compatibilità, ottimizzazioni ...). Il **Virtual File System Switch** è lo strato che premette ai programmi applicativi di essere indipendenti dall'implementazione del file system: ridirigono le richieste di servizi alle corrette routine. L'interfaccia unica è chiamata **modello del VFS**.

Più FS possono coesistere nello stesso sistema, con il vincolo che ogni partizione (**Device**) sia essere gestita da un solo FS. Ogni Device è costituito da un insieme di blocchi identificati da un **LBA** (Logical Block Address).

1.3 Ottimizzazioni con la memoria

L'accesso a periferiche come un disco può richiedere diversi millisecondi: perciò Linux tenta di mantenere in memoria i dati letti da disco il più lungo possibile. Il VFS deve quindi collaborare strettamente col gestore della memoria. Quando il VFS o un FS ha bisogno di accedere a un Device, la richiesta passa per la **Page Cache** che si occupa dell'eventuale trasferimento in memoria del blocco.

Chapter 2

Modello d'utente

2.1 Modello per l'accesso al singolo file

L'accesso a un file può avvenire secondo 2 modalità:

- mappatura di una VMA sul file (`mmap()`)
- system calls, le cui API permettono di creare, cancellare, leggere o scrivere su file, oppure di creare o cancellare i direttori.

2.1.1 Modello di un file

Un file è costituito da una sequenza di byte memorizzati su disco. Un programma non lavora mai direttamente su tale contenuto, ma su una sua copia in memoria.

Esiste un'**indicatore della posizione corrente** che punta al byte in cui avviene l'operazione di lettura o scrittura.

Al momento dell'apertura un file (identificato dal nome) restituirci un intero non negativo: il **descrittore del file** (un puntatore).

Funzioni della libreria di Linux

- `int open(char *file_name, int type, int permission)`: `pathname` è il nome completo; `type` indica scrittura, lettura o entrambi.
- `int creat(char *file_name, int permission)`
- `int close(int fd)`: elimina il legame tra descrittore e file. Non è garantita la scrittura su disco.
- `int fsync(int fd)`: forza la scrittura su disco.
- `int read(int fd, void *buffer, int size)`: lettura
- `int write(int fd, void *buffer, int size)`: scrittura
- `long lseek(int fd, long offset, int whence)`: modifica la posizione corrente (permette di non operare in maniera sequenziale su un file) e restituisce il valore della nuova posizione. `whence` può essere 0, 1 o 2: inizio del file, posizione corrente, fine del file.

2.2 Organizzazione complessiva dei file

2.2.1 Directory e filename

I file e le partizioni sono organizzati in una struttura ad albero i cui nodi sono detti **directory** (folder...). Le directory sono dei file (di **tipo directory**) che contengono i nomi e le informazioni di accesso ad altri file (sia di **tipo normale** che di tipo directory). Per scrivere o leggere su questi file si utilizzano servizi speciali.

Il **Pathname** o nome completo è costituito dal nome delle directory separate da uno "/". La **root** è indicata da "/".

- `link()`: permette di creare un nuovo nome per un file già esistente
- `unlink(char *file_name)`: rimuove il riferimento a un file. In Linux non esiste un servizio per la cancellazione di un file, ma semplicemente se un file rimane senza riferimenti, quello spazio diventa libero.
- `mkdir()`: è l'equivalente di `creat()` per le directory.

2.2.2 Periferiche e file speciali

Tutte le periferiche sono considerate come **tipo speciale**. I file speciali sono trattati come normali file, con la differenza che alcune delle operazioni perdono di significato. Spesso i file speciali risiedono nella cartella `/dev`.

In questa cartella sono presenti anche i terminali virtuali (creati dalla gui), di solito indicati con `/dev/pts/‘n‘`. A differenza dei dispositivi fisici, la creazione di terminali virtuali avviene su comando. Per conoscere il nome del file speciale associato ad un terminale si usa il comando `tty`.

L'interprete dei comandi (shell) fa in modo che un programma in esecuzione abbia i descrittori 0, 1 e 2: **stdin**, **stdout** e **stderr** (file speciali). Questi descrittori possono essere rediretti su file normali: quando ad esempio si usa la funzione `close(0)`, la stdin viene rediretta sul file successivamente aperto.

La funzione `dup()` duplica un descrittore, seppur l'indicatore della posizione rimane univoco.

Creazione di file speciali

Ogni periferica è identificata da una coppia di numeri: **major** e **minor**. Il primo identifica il driver, il secondo il dispositivo. L'equivalente di `creat()` per i file speciali è `mknod(pathname, type, major, minor)`, la quale può essere utilizzata solo dall'amministratore del sistema (type = character / block).

2.3 I Device

A differenza di Windows, in cui i device sono identificati da una lettera (C:, D:) che india la radice dell'albero, in Linux i device sono rappresentati da nodi detti **mount point**. Il sottoalbero ne descrive la struttura interna.

Per inserire un nuovo device nel sistema si deve:

- associare un FS al device (con `mkfs -t type device`)
- montare il volume in un mount point (con `mount device mount_point`)

A questo punto i file possono essere acceduti sia dal mount point che dal device.

Chapter 3

Il Modello del VFS

Le informazioni contenute nel VFS sono di 2 tipi: quelle statiche relative al contenuto dei file e quelle dinamiche associate ai file aperti.

Le strutture dati su cui si base il VFS sono:

- `struct dentry` (directory entry): ogni istanza rappresenta una directory nel VFS.
- `struct inode`: ogni istanza rappresenta un file fisicamente presente nei device.
- `struct file`: ogni istanza rappresenta un file aperto.

Queste strutture dati sono accedute tramite puntatori contenuti in:

- **struttura delle directory**
- **struttura di accesso dai processi ai file aperti**

3.1 Struttura delle directory

Questa struttura dati ad albero è costituita da `struct dentry`.

```
struct dentry {
    struct inode *d_inode;
    struct dentry *d_parent;
    struct qstr d_name;
    struct list_head d_subdirs;
    struct list_head d_child;
    ...
}
```

- `d_inode` è un puntatore alla struttura che rappresenta il file fisico.
- `d_name` è il nome del file o della directory.
- `d_subdirs` è un puntatore al primo dei figli.
- `d_child` è un puntatore al prossimo fratello.

3.2 Struttura di accesso dai processi ai file aperti

Nel descrittore dei processi sono presenti 2 puntatori rilevanti ai fini dell'accesso ai file:

```
struct task_struct {
    struct fs_struct *fs;
    struct files_struct *files;
    ...
}
```

- `fs` è un puntatore alla lista dei file system utilizzati dal processo.

```
struct files_struct {
    int next_fd;
    struct file **fd_array[NR_OPEN_DEFAULT];
    ...
}
```

- Ogni elemento di `fd_array` è un puntatore a `struct file`, una struttura dati utilizzata per rappresentare i file aperti.
- `fd`, descrittore del file F, è l'indice di `fd_array`.

```
struct file {
    struct list_head f_list;
    struct dentry *f_dentry;
    loff_t f_pos;
    int f_count;
    ...
}
```

- `f_list` serve a collegar tutti i file aperti in una lista.
- `f_dentry` è un puntatore alla directory entry utilizzata per aprire il file.
- `f_pos` è l'indicatore della posizione corrente.
- `f_count` è il numero di processi che hanno aperto il file.

Il percorso di un file aperto è quindi: `file_descriptor->files.fd_array[fd]->f_dentry->d_inode`.

Quando un nuovo file viene aperto, viene creata una nuova istanda di `struct file` e viene inserita nella prima posizione libera della tabella dei file aperti. Se non sono già presenti in memoria vengono create anche le struct `dentry` e `inode` (eliminate quando `f_count` diventa 0).

L'operazione di `fork()` duplica la tabella dei file aperti usando la funzione `dup()`, quindi il riferimento a `struct file` di tutti i file aperti è lo stesso.

Se lo stesso file fisico viene aperto da un'altra posizione (link), invece, viene creata una nuova istanza di `struct file`, il quale avrà una `f_pos` differente, che punta a una `f_dentry`, la quale però punta allo stesso `inode`.

3.3 Struct inode

La corrispondenza tra `struct inode` e file fisico è 1 a 1.

```
struct inode {
loff_t i_size;
    struct list_head i_dentry;
    struct super_block *i_sb;
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct address_space *i_mapping;
    ...
}
```

- `i_size` è la dimensione del file.
- `i_dentry` è l'inizio della lista dei dentry del file (un file può avere più nomi: link).
- `i_sb` è un puntatore al superblocco che lo gestisce: una struttura dati che contiene le caratteristiche del FS.
- `i_op` e `i_fop` contengono i puntatori alle funzioni dello specifico FS (non risiedono nel blocco per rendere le operazioni più immediate).
  - `i_op` contiene le operazioni per la gestione complessiva dei file e delle directory: `create`, `mknod`, `link`, `unlink`, `mkdir`, `rmdir` ...
  - `i_fop` contiene le operazioni di accesso ai file: `read`, `write`, `open`, `close`, `llseek` ...
- `i_mapping` contiene informazioni per il trasferimento dei dati da device a memoria.

3.4 Accesso ai dati

`read()` e `write()` chiamano delle system calls. Il disco è suddiviso in LBA da 1024 byte e la page cache in pagine da 4096 byte. La lettura di un file è basata sulla pagina: è l'unità minima trasferita dal SO. Una pagina è quindi costituita da 4 blocchi, ognuno identificato dal **File Block Address** (**FBA**).

- determina la pagina del file alla quale i byte appartengono: `OFFSET = POS % 4096`.
- verifica se la pagina è contenuta nella page cache.
- altrimenti alloca una nuova pagina nella page cache e carica 4 LBA: `FP = POS / 4096`.
- copia i dati richiesti all'indirizzo richiesto dalla system call.

Blocchi di FBA consecutivi possono essere allocati in LBA non consecutivi (dipende dalla disponibilità di risorse e dal FS utilizzato).

3.4.1 Page cache

La page cache contiene pagine fisiche, strutture dati e funzioni. Le strutture dati contengono i descrittori delle pagine fisiche, i quali a loro volta contengono l'identificatore del file, l'offset, il ref_count .... Una pagina è identificata dalla coppia `<file, offset>`.

Il campo `i_mapping` di `struct inode` punta a

```
struct address_space {
struct inode host;
struct radix_tree_root page_tree;
struct ... a_ops;
...
}
```

Questa struttura serve a determinare se una pagina di un file è presente nella page cache.

- `page_tree` è una struttura dati che punta a tutte le pagine nella Page Cache relative al file. Se la pagina non viene trovata, si procede a caricarla.
- `a_ops` contiene operazioni specifiche del FS per accedere alle pagine:
  `readpage`, `writepage`, ...
  Queste solitamente chiamano funzioni del device driver.

Questo processo supporta anche la mappatura di una VMA su un file.

Altro

- Esistono anche la `inode_cache` e la `dentry_cache`, separate dalla page cache.
- esiste la possibilità di caricare sulla page cache un numero di blocchi che non è multiplo della dimensione della pagina.

Chapter 4

Gli extended FS ext2, ext3 e ext4

4.1 Introduzione

I FS default di windows sono **ext2** e **ext3** e **ext4** (le ultime sono delle versioni aggiornate e compatibili con la prima).

Il journaling è una tecnica che evita la creazione di file corrotti in caso di chiusura anomala.

4.2 Il FS ext2

Ogni device è suddiviso in **Block Groups**

4.2.1 ext2 senza Block Groups

Le principali strutture dati utilizzate sono:

- **superblock**: contiene le informazioni globali e viene eventualmente utilizzato durante il boot.
- **tabella degli inode**: array di inode
- **inode**: informazioni relative al singolo file
- **directories**
- **blocchi dati**: il contenuto effettivo dei file

Il superblock ha un indirizzo prefissato: da esso sono raggiungibili la tabella degli inode e la root directory. La root directory può coincidere con la root del sistema o con un mount point.

A ogni istante un blocco dati appartiene a un file oppure alla **lista libera**.

Contenuto di un inode

Contiene le informazioni generali relative a un file. Nei cataloghi i file sono individuati dalla coppia `<nome, numero di inode>`. Le informazioni più importanti sono:

- **tipo**: normale, directory o speciale
- **numero di riferimenti** (link) al file
- **dimensione**
- **puntatori ai blocchi dati** a eccetto dei file speciali.

Accesso ai Blocchi dati

La dimensione degli FBA (1KB - 64KB) deve essere minore o uguale a quelli di una pagina (4KB su x64).

In ext2 l'accesso ai blocchi è basato su 15 puntatori. I primi 12 puntano direttamente ai blocchi dati, gli ultimi 3 sono puntatori indiretti (uno singolo, uno doppio e uno triplo). Questa struttura serve a rendere più efficiente l'accesso ai file di piccole dimensioni.

Se b è la dimensione di un blocco, allora la massima dimensione del file è

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

$b/4$ è la dimensione di un blocco diviso per la dimensione di un puntatore.

4.2.2 ext2 con Block Groups

Un Block Group è un insieme di LBA contigui. L'accesso consecutivo a LBA consecutivi è più veloce di un accesso casuale: è compito del driver disporre vicini blocchi con informazioni correlate.

La dimensione di un Block Group è `8 * block_size`: in un blocco è memorizzata una bitmap che indica l'utilizzo o meno degli altri blocchi.

Il superblocco del volume è unico, ma può essere replicato in più block group.

La tabella degli inode è logicamente unica, suddivisa in parti uguali nei diversi block group. Nel superblocco è memorizzato il parametro `inodes_per_group`.

```
    bg = (inode_num - 1) / inodes_per_group
offset = ((inode_num - 1) % inodes_per_group) * inode_size
```

In generale, le strutture dati contenute in un Block Group sono:

- **Superblock**: contiene le informazioni globali (opzionale)
- **Block group descriptor table**
- **Block usage bitmap**
- **Inode usage bitmap**
- **Porzione della tabella degli inode**: contenente soli `inodes_per_group` inode
- **blocchi dati**: il contenuto effettivo dei file

4.3 Extent in ext4

Un **extent** è un insieme di blocchi contigui sia all'interno del file che sul dispositivo fisico. Un extent è rappresentato da `<FBA iniziale, LBA iniziale, dimensione>`: non sono più necessari i puntatori ai singoli blocchi.

Chapter 5

Driver a carattere

5.1 Introduzione

I **device driver** (gestori delle periferiche) sono dei moduli software che adattano una l'hardware di una specifica periferica a funzionare nel contesto del sistema operativo. I gestori sono suddivisi in **character devie driver** e **block device driver**.

5.2 File speciali e identificazione dei gestori

Per ogni periferica installata nel sistema deve esistere un corrispondente file speciale (di solito nella cartella `/dev`).

Ogni periferica è identificata da una coppia di numeri: **major** e **minor**. Il primo identifica il driver, il secondo il dispositivo. Questi valori sono posti nell'inode del file speciale.

5.3 Routine di un gestore di periferica

Le funzioni che un gestore di periferica svolge sono:

- inizializzazione all'avvio del SO
- porre la periferica in servizio o fuori servizio
- ricevere e inviare dati
- scoprire e gestire gli errori
- gestire gli interrupt

Il gestore è quindi costituito da routine attivate al momento opportuno e dalla routine di interrupt. In ogni gestore esiste una **tabella delle operazioni** così composta:

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

- `open()` verifica che la periferica sia attiva, e nel caso questa si utilizzabile da un solo processo alla volta viene controllata un'apposita variabile (e nel caso viene restituito un errore).
- `release()` viene chiamata dalla `close()` di un file speciale: controlla che il buffer sia svuotato e modifica la variabile di utilizzo.
- `write()`: vedi 5.4
- `read()` simmetrica a `write()`.
- `ioctl(int fd, int command, int parameter)` permette di svolgere le operazioni specifiche della periferica.

All'avvio del SO viene attivata una funzione di inizializzazione per ogni gestore di periferica installato che restituisce un puntatore alla propria tabella delle operazioni. Il kernel si salva il puntatore nella **tabella dei gestori a carattere** o nella **tabella dei gestori a blocchi**.

Ad esempio, nel caso di una periferica a carattere (che non utilizza la page cache), il kernel accede al file `/dev/tty1`, dove legge il numero di major `4` , per accedere alla tabella delle operazioni e chiamare la funzione `open()`.

5.4 Principi di funzionamene di un gestore di periferica

Essendo che un processo in attesa di un servizio della periferica non è in esecuzione, questo non è in grado di rispondere prontamente alla richiesta di interrupt e di trasferire i dati. Perciò i dati da scrivere o leggere vengono conservati dal gestore stesso in un buffer.

- il processo chiama X.write()
- la routine di X.write() copia dallo spazio del processo nel buffer del gestore $C = min(buffer.size(), char_to_write)$ caratteri.
- X.write() manda il primo carattere alla periferica.
- X.write() deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il successivo. `wait_event_iterruptible(WQ, empty_buffer)`.
- interrupt della periferica
- la routine di interrupt viene subito eseguita e se disponibile il carattere successivo viene inviato, altrimenti...
- `wake_up(WQ)` risveglia il processo e lo mette in stato di pronto.
- quando il processo riprende l'esecuzione X.write() copia i successivi caratteri nel buffer del gestore, altrimenti...
- il processo ritorna al modo U.