filesystem e *device driver*

aspetti generali

- fornire un livello di astrazione (**modello di utente**) omogeneo e ragionevolmente facile da utilizzare sopra il mondo complesso e variegato dei dispositivi periferici
- 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
- ***Virtual Filesystem*** (**VFS**): un'interfaccia unica sopra diversi *filesystem* (FS) specifici
  - per compatibilità col passato: esempio ***ext2***, ***ext3*** ed ***ext4***
  - per compatibilità con standard: esempio **ISO 9660** per Compact Disk
  - per compatibilità con altri sistemi: esempio **NTFS** di Windows
  - per ottenere particolari prestazioni: esempio **FS specializzato** per gestire file molto grandi o molto piccoli, ma molto numerosi, ecc
- ciascun dispositivo (volume o partizione) può essere gestito da un solo FS

gestore di periferica (*device driver*)

- **gestore di periferica (*device driver*)**: adattamento alla varietà di periferiche
- per rendere possibile l'inserimento di nuovi *driver* è necessario
  - avere un meccanismo generale per l'inserimento nel sistema di nuovo software, costituito dai cosiddetti ***kernel_module***
  - 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
- il sistema operativo LINUX definisce due tipi di modello di *driver*
- ***driver* a carattere**: il *driver* esegue le operazioni richieste dai livelli superiori (per esempio *read* e *write*) quando esse vengono richieste
- ***driver* a blocchi**
  - il sistema mette in una coda le operazioni richieste
  - il *driver* le può prelevare della coda e servirle, eventualmente modificandone l'ordine allo scopo di ottimizzare l'accesso alla periferica
  - ovviamente un *driver* di tipo a blocchi non ha senso per una periferica strettamente sequenziale, come per esempio una stampante

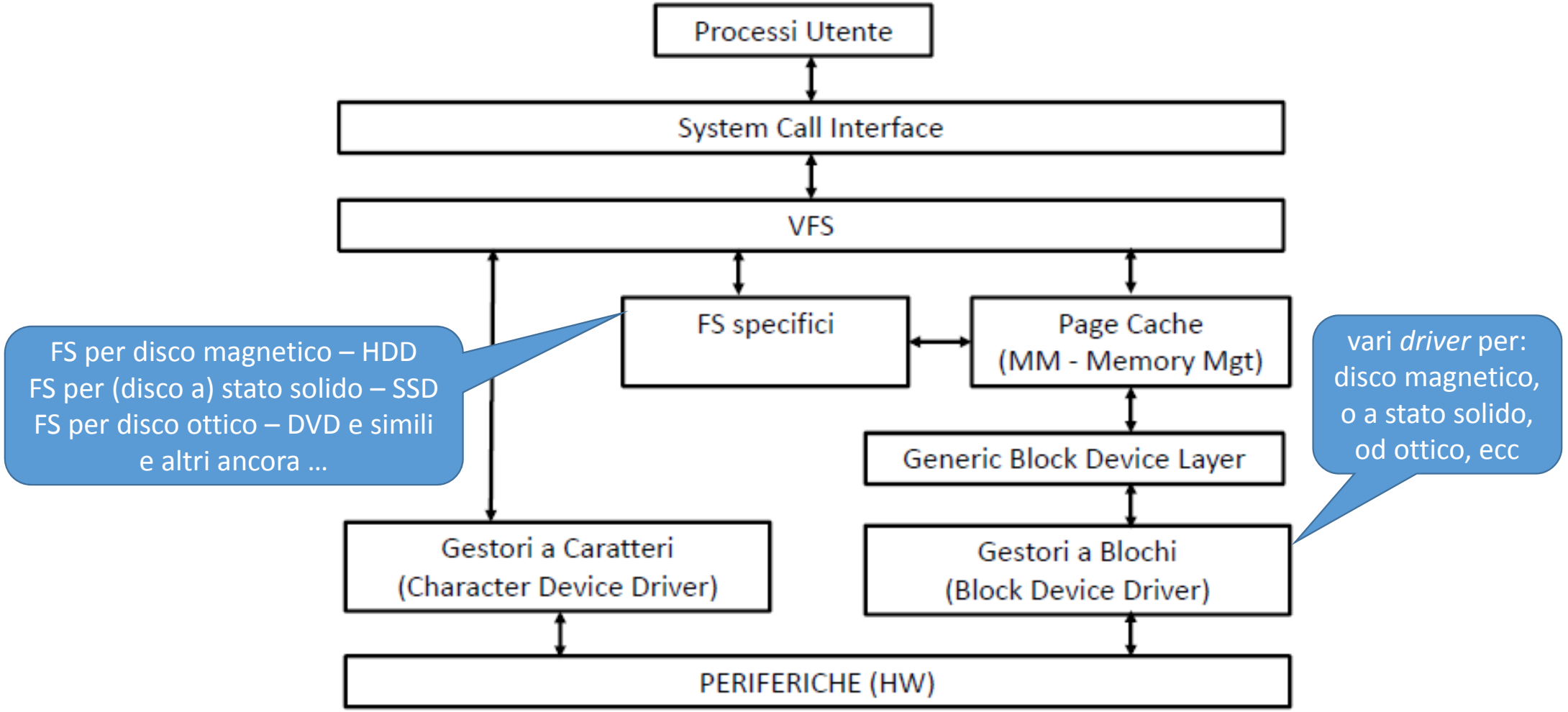
driver dei dispositivi a blocchi

- l'operazione richiesta (*read* o *write*) contiene i parametri necessari per eseguirla
  - indirizzo del blocco iniziale su disco
  - numero di byte da trasferire
  - indirizzo del *buffer* di sistema da utilizzare
  - verso del trasferimento (lettura o scrittura)
- quasi sempre l'operazione viene eseguita dal *canale di DMA* al momento opportuno
- l'interrupt di fine DMA segnala la conclusione dell'operazione di trasferimento di un blocco di dati ed effettua il risveglio del processo in attesa
- tale meccanismo permette al gestore di periferica di ***riorganizzare la coda di operazioni richieste in modo da ottimizzare la sequenza di accessi al disco***
- ecco un possibile algoritmo di ottimizzazione (qui tratteggiato solo a titolo di esempio)
  - simile al cosiddetto *algoritmo dell'ascensore*, cioè
    - ✓ sali o scendi mantenendo il verso (in su o in giù), ma fermati a ogni piano dove c'è una richiesta, indipendentemente dall'ordine di arrivo delle richieste (pressione dei pulsanti di chiamata)
    - ✓ cambia verso solo quando (nel verso corrente) non ci sono più richieste pendenti
  - nel caso di un disco: sposta la testina di lettura / scrittura su tracce / settori contigui mantenendo il verso di spostamento, leggendo / scrivendo dove richiesto e riordinando la coda se è opportuno, e inverti il verso solo quando non ci sono più operazioni pendenti nel verso corrente

interazione con la memoria

- accedere a disco ha tempo di esecuzione dell'ordine di grandezza **millisecondo** (10^{-3} s), ma eseguire un'istruzione macchina ha tempo dell'ordine di grandezza **nanosecondo** (10^{-9} s) – rapporto pari a **6** ordini di grandezza !
- ***LINUX cerca di tenere in memoria il più a lungo possibile i dati letti da disco***
- FS e gestione della memoria (in particolare la *Page Cache*) devono collaborare
- quando il VFS o un FS ha bisogno di leggere un blocco su un volume (*device*), in realtà lo richiede alla *Page Cache*
  - se il blocco è già in *Page Cache*, questa restituisce subito l'indirizzo di memoria del blocco
  - altrimenti la *Page Cache* alloca in memoria lo spazio necessario per una pagina e richiede al gestore a blocchi di leggere il blocco dal dispositivo (volume, disco) e di caricarlo nella pagina
- in entrambi i casi, quando la richiesta è completata il VFS o il FS sono in grado di operare sul blocco richiesto già trasferito in memoria

struttura complessiva del sistema (VFS + vari FS)



modello di utente – accesso al singolo file

- il modello di utente può essere suddiviso in due aspetti
  - accesso al singolo file
  - organizzazione complessiva dei file
- l'accesso a un file può avvenire secondo due modalità
  - ➢ mappando una VMA sul file tramite la funzione ***mmap***
  - ➢ o con una ***system call classica*** di accesso a file (***read*** o ***write***)
    (in Linux sono simili alle funzioni della libreria C standard)
- il VFS di Linux deve realizzare entrambe le modalità
- la mappatura di VMA è utilizzata dal SO anche per realizzare i meccanismi fondamentali di esecuzione dei programmi

system call classica per file – 1

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

- *open* e *creat* rendono il file disponibile all'uso da parte del processo
  (*open* apre un file supposto già esistente e *creat* lo crea ex novo, all'inizio vuoto)
- restituiscono un descrittore (un intero), che è sempre il primo libero
- inizializzano la posizione corrente nel file (in genere sul primo byte)

```
int letti = read (int fd, char * buffer, int numero)
int scritti = write (int fd, char * buffer, int numero)
```

- *read* e *write* lavorano a partire dalla posizione corrente nel file, muovendosi in avanti
- restituiscono quanti caratteri hanno effettivamente letti o scritti (–1 indica errore)
- scrivere oltre la fine del file estende il file quanto serve per completare la scrittura

system call classica per file – 2

`close (int fd)`

- *close* elimina il legame tra descrittore e file, e rende il descrittore riusabile
- non garantisce che i dati scritti in memoria vengano trasferiti subito su disco, ma molti *filesystem* li scaricano effettivamente su disco al momento di *close*

`int fsync (int fd)`

- *fsync* garantisce che i dati scritti in memoria vengano scaricati subito su disco, ossia allinea i dati in memoria e quelli su disco, ma non chiude il file

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

- *lseek* modifica solo la posizione corrente: ***pos. corr.*** = ***riferimento*** + ***offset***
- il riferimento può specificare inizio file, precedente pos. corr. o fine file
- *lseek* restituisce la nuova posizione corrente

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

modello di utente – organizzazione complessiva

- ecco i tre tipi di file di Linux
  - file ***normale***
    - ✓ di tipo ***dato***: contiene dati generici (testo, dati con formato, o eseguibile)
    - ✓ di tipo ***catalogo***: è un file dati, ma è usato per rappresentare una lista di nomi di file (e consente solo operazioni sensate per un catalogo)
  - file ***speciale***: rappresenta una periferica, e leggere o scrivere un file speciale significa trasferire dati con la periferica, in lettura o scrittura
- ecco alcune caratteristiche del catalogo (anche chiamato *directory*, cartella o *folder*)
  - i cataloghi sono organizzati ad albero, con radice *root* rappresentabile con «/»
  - il *nome articolato* (detto *pathname*) di un file descrive il cammino da *root* al file, attraverso i cataloghi lungo il percorso
  - p.es.: «/A/dir1/C/file», dove «/» (ossia *root*), «A», «dir1» e «C» sono cataloghi, e «file» è il nome del file all'interno del catalogo «C»
  - un nome di file senza percorso si riferisce al catalogo corrente, rappresentabile anche con «./», p. es. «file» è equivalente a «./file»
- per creare un catalogo si usa la funzione di libreria o comando Shell ***mkdir***
- per creare un file speciale si usa la funzione di libreria o comando Shell ***mknod***

modello di utente – creare (o eliminare) un riferimento a file

`link` `(`**`char`** `* old_name,` **`char`** `* new_name)`

- *link* aggiunge in un catalogo (usualmente nel catalogo corrente) un nuovo riferimento, ossia un nuovo nome, a un file esistente
- ma non può creare un file ex novo (solo *creat* lo può fare) !
- due cataloghi possono contenere due riferimenti uguali allo stesso file
- un catalogo può contenere due riferimenti diversi allo stesso file
- teoricamente il numero totale di riferimenti a un file è illimitato
- il riferimento iniziale al file (il nome assegnatogli tramite *creat*) non è in alcun modo privilegiato rispetto a qualunque altro riferimento aggiuntogli in seguito tramite *link*

`unlink` `(`**`char`** `* name)`

- *unlink* elimina da un catalogo (usualmente dal catalogo corrente) un riferimento a file
- se il file non ha più riferimenti (nomi), viene fisicamente cancellato sul volume !
- in Linux non esiste la cancellazione esplicita di un file, il file viene automaticamente cancellato non appena perde il suo ultimo riferimento (nome)

file speciale

- per il programma, un file speciale è molto simile a un file normale contenente dati
  - si possono eseguire le operazioni tipiche citate prima, per esempio *open*, *read*, *write*, ecc
  - tuttavia fisicamente esso non corrisponde a un normale file su disco, ma a una periferica
- nelle configurazioni di LINUX più comuni, spesso i file speciali sono posti sotto il catalogo «/dev», come per esempio
  - a un terminale, cioè all'insieme di video-audio-camera + tastiera + puntatore gestiti come dispositivo integrato, può corrispondere un file speciale con nome articolato «/dev/tty10»
  - un programma può aprire tale file tramite un comando come

    ```
    fd = open ("/dev/tty10")
    ```

    e poi scrivere sul terminale corrispondente tramite il comando

    ```
    write (fd, buffer, numerocaratteri)
    ```

    esattamente come se fosse un file normale
- comunque un file speciale può anche stare in un catalogo (o più di uno) qualunque

descrittore di file standard

- un programma va in esecuzione con i tre descrittori 0, 1, e 2 aperti, chiamati

stdin	standard input	$fd = 0$
stdout	standard output	$fd = 1$
stderr	standard error	$fd = 2$

- i file associati a tali descrittori sono speciali e corrispondono alla tastiera (*stdin*) e al video (*stdout* e *stderr* rispettivamente) del terminale
- numerose funzioni di sistema utilizzano tali descrittori, p. es. *printf* scrive su *stdout*, *scanf* legge da *stdin*, e altre ancora
- si possono ridirigere i tre descrittori standard su dispositivi o file diversi, così

```
// chiude stdin finora diretto su terminale
close (0)
// riapre stdin ridirigendolo su ./inputfile
fd = open ("./inputfile", O_RDONLY)
```

creazione di file speciale

- ciascuna periferica installata in un sistema è identificata da una coppia di numeri, detti **ma*jor*** e ***minor***, memorizzati nel file speciale associato alla periferica
- tutte le periferiche dello stesso tipo hanno un identico numero *major*, mentre il numero *minor* serve a distinguere le diverse periferiche di uno stesso tipo
- ecco un esempio tratto dal catalogo (di sistema) «/dev», che ospita numerosi file speciali

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

- ecco come creare un file speciale (versione semplificata) tramite funzione di libreria

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

dove type indica il tipo di *driver*, a carattere oppure a blocchi

- ecco invece un esempio di creazione tramite comando Shell (comando *mknod*)

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

crea un file speciale di tipo carattere «tty4», in «/dev» e associato a *major* = 4 e *minor* = 5

volume (*device* e partizione)

- in LINUX esiste un solo albero di cataloghi con una sola radice denotata da «/»
- diversi volumi sono rappresentati da altrettanti nodi dell'albero, detti ***mount_point***
  - il sottoalbero la cui radice è un *mount_point* descrive la struttura interna del volume (*device*)
  - la posizione del *mount_point* nell'albero generale lo rende raggiungibile a partire da «/» (*root*)
  - in Windows ci possono essere più alberi di cataloghi e ciascun albero parte da un dispositivo di solito individuato tramite una lettera, per esempio C: o D:
- per inserire un nuovo volume nella struttura generale occorrono due operazioni
  - associare un *filesystem* al volume (device) – comando Shell ***mkfs*** (vedi dispensa)
  - montare il volume (device) in un opportuno *mount_point* dell'albero complessivo

```
mount device mount_point
```

(questo è un comando Shell, ma esiste anche una funzione di libreria equivalente)

modello di VFS

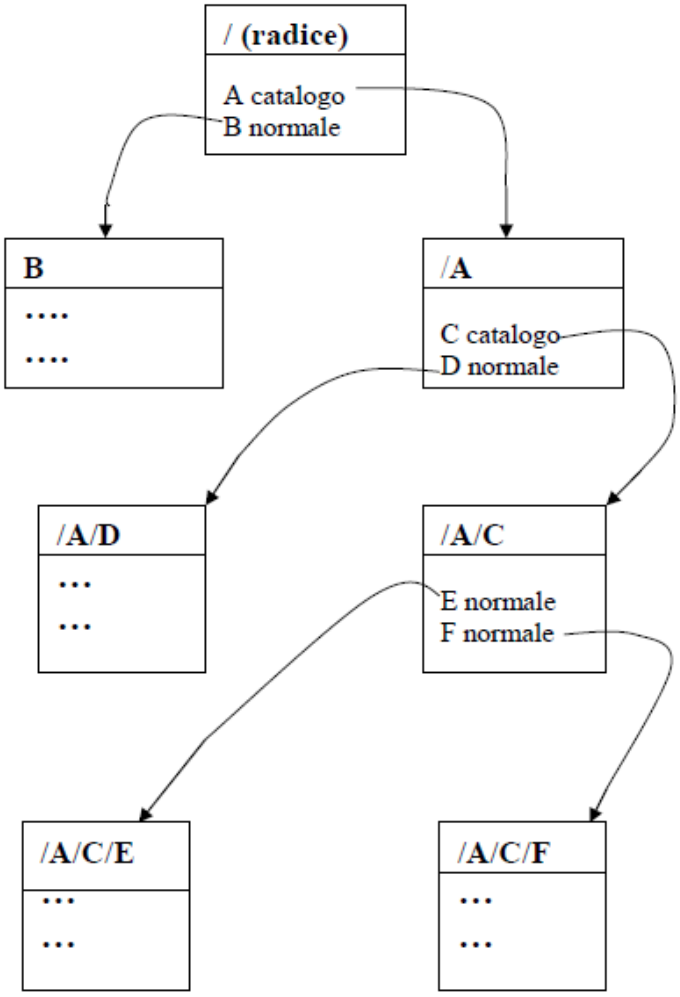
- il modello di VFS deve rappresentare due insiemi di informazione
  - quella relativamente statica contenuta nei file e nei cataloghi memorizzati sui diversi volumi
  - quella dinamica associata ai file e ai cataloghi aperti durante il funzionamento del sistema, per esempio la posizione corrente in un file
- il modello di VFS si basa su varie strutture dati C, ecco le tre principali

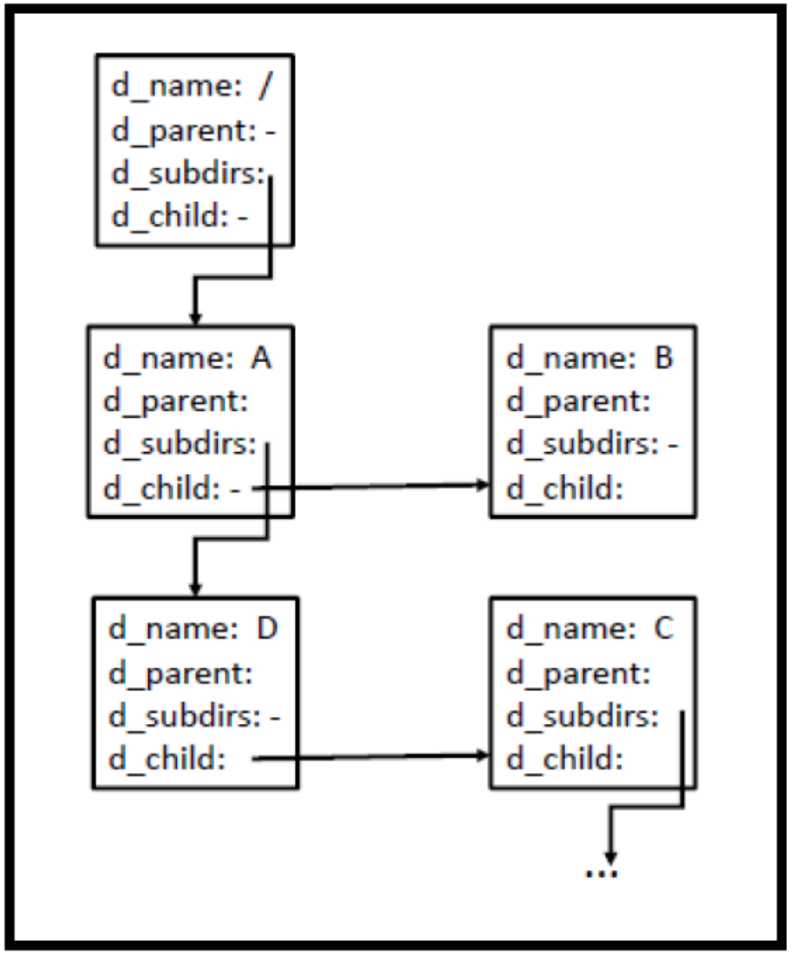
  `struct dentry` ciascuna istanza rappresenta una riga di catalogo nel VFS

  `struct inode` ciascuna istanza rappresenta un file fisico su un volume

  `struct file` ciascuna istanza rappresenta un file aperto nel sistema
- esse sono utilizzate in combinazione per realizzare strutture dati dinamiche
- ecco le due strutture dati dinamiche principali create e gestite dal sistema
  - la **struttura dei cataloghi** che costituisce l'albero complessivo del FS
  - la **struttura di accesso** ai file aperti da parte di un singolo processo

cataloghi nel modello di utente e nel VFS

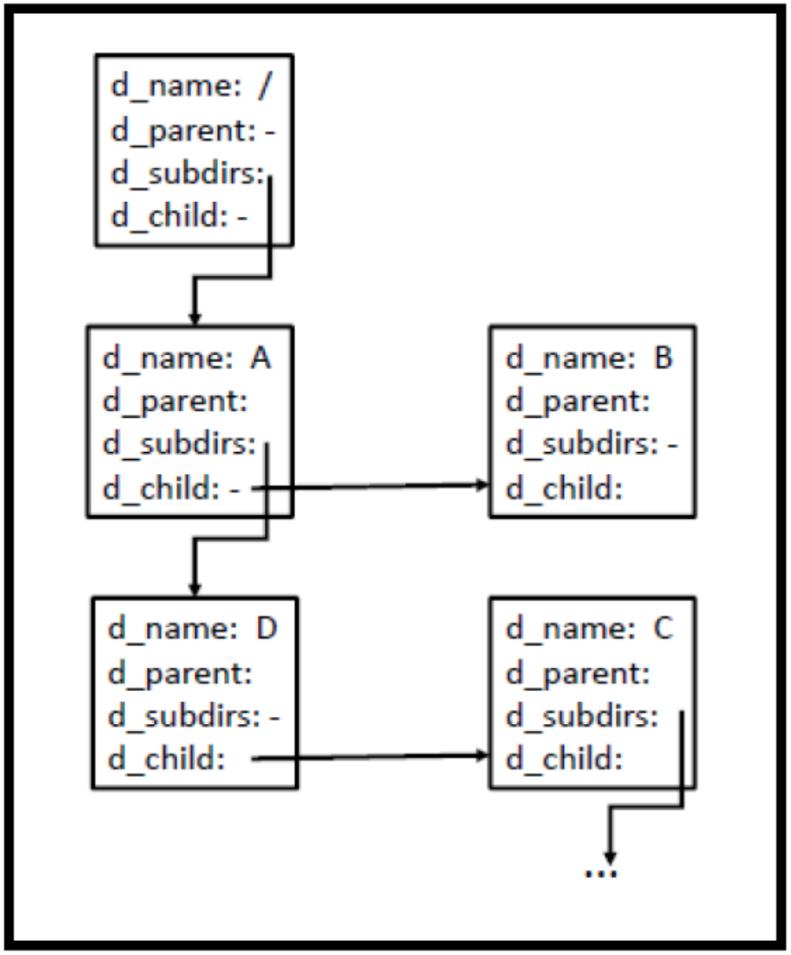




struttura del catalogo – *struct dentry*

```
struct dentry {
   // puntatore allo i-node del file
    struct inode * d_inode
    struct dentry * d_parent
    // nome del file (stringa di char)
    struct qstr d_name
    // puntatore al primo dir figlio
    struct list_head d_subdirs
    // puntatore al fratello nell'albero
    struct list_head d_child
    ...
} / * dentry */

// d_inode è utilizzato per puntare al
// file dati nella struttura di accesso
```



struttura di accesso ai file aperti – 1

- dal descrittore del processo alla sua tabella dei file aperti *fd_array*

```
struct task_struct {
    ...
    struct files_struct * files
    ...
} /* task_struct */

struct files_struct {
   ...
   struct file * fd_array [NR_OPEN_DEFAULT]
   ...
} /* files_struct */
```

- *fd_array* contiene un elemento per ciascun file aperto dal processo
- e costituisce la **tabella dei file aperti dal processo**
- ciascun elemento è un puntatore a un'istanza di *struct file*

struttura di accesso ai file aperti – 2

- da ciascuna istanza di *struct file* allo *i-node del file*

```
struct file {
    ...
    // riferimento al dentry usato in apertura
    struct dentry * f_dentry
    // posizione corrente nel file aperto
    loff_t f_pos    // è un intero da 64 bit
    // contatore dei riferimenti al file aperto
    int    f_count
    ...
} /* file */
```

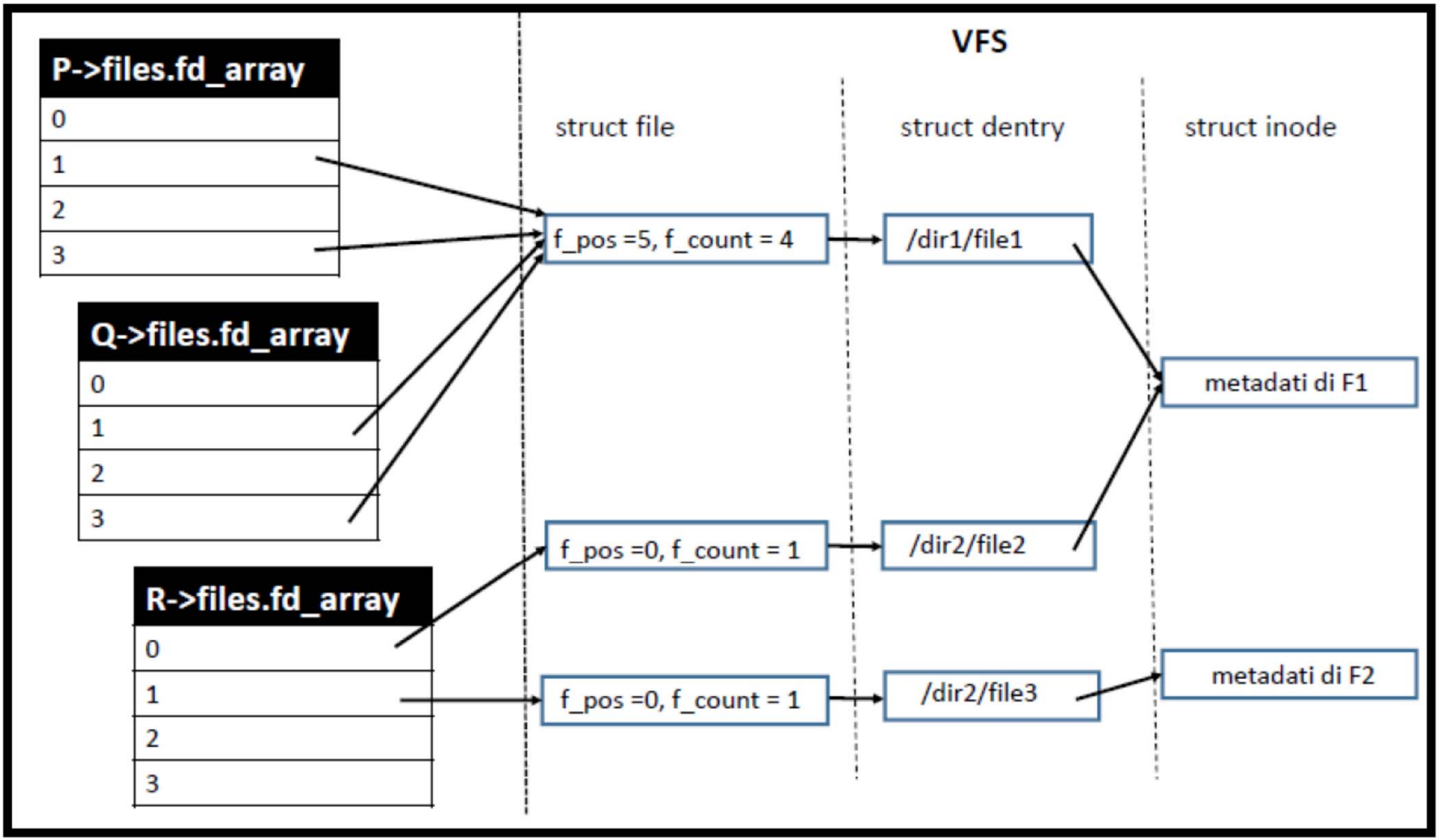
- *f_dentry* punta allo *i-node* del file (vedi le slide successive per lo *i-node*)
- un processo P apre un file con descrittore *fd*, e così si raggiunge lo *i-node*

`⟨task_struct di P⟩→files.fd_array[fd]→f_dentry→d_inode`

esempio

- il processo P ha aperto il file denominato «/dir1/file1» quando il primo descrittore libero era nella riga 1 di *fd_array*; indichiamo il file fisico corrispondente con F1
- P ha duplicato (tramite *dup*) il descrittore 1 quando il primo descrittore libero era 3
  - la chiamata di sistema *dup* duplica un descrittore di file aperto, usando il primo descrittore libero
  - il descrittore così duplicato è distinto da quello di partenza, ma è del tutto equivalente ad esso
  - la funzione *dup* è spesso usata in certe operazioni di ridirezione di file (standard e di altro genere)
- P ha letti cinque caratteri dal file con descrittore 1 (oppure 3)
- P ha eseguita una *fork* e così ha creato il processo figlio Q
- il processo R ha aperto il file denominato «/dir2/file2» ottenendo il descrittore 0; il file fisico corrispondente è lo stesso file F1; significa che in precedenza era stato usato il comando *link* per creare un secondo riferimento allo stesso file fisico
- R ha aperto il file denominato «/dir2/file3» corrispondente a un file fisico diverso da F1, per esempio F2
- la figura seguente mostra lo stato finale raggiunto dopo questa sequenza di eventi

esempio


VFS
struct file
struct dentry
struct inode
P->files.fd_array
0
1
2
3
Q->files.fd_array
0
1
2
3
R->files.fd_array
0
1
2
3
f_pos =5, f_count = 4
/dir1/file1
metadati di F1
f_pos =0, f_count = 1
/dir2/file2
f_pos =0, f_count = 1
/dir2/file3
metadati di F2

accesso ai dati

- ecco come svolgere le operazioni *read* e *write* di *lettura e scrittura di un certo numero di byte a partire dalla posizione corrente (muovendo in avanti)*
  - prima bisogna localizzare i dati sul volume, che è organizzato in blocchi logici, ciascuno dei quali e identificato (indicizzato) da un LBA (Logical Block Address)
    - ✓ di solito un blocco è costituito da un numero prefissato di settori consecutivi sul disco
    - ✓ per semplicità si suppone valga: dimensione del blocco = 1024 byte
  - e poi bisogna fare transitare i dati dalla memoria, e in particolare dalla *Page Cache*
    - ✓ la Page Cache è organizzata in pagine da 4 K byte (4096 byte)
- la lettura di un file è basata sulla pagina e il SO trasferisce sempre pagine intere di dati per ogni operazione
  1. determina la pagina del file alla quale i byte appartengono
  2. **if** (la pagina NON è contenuta nella *Page Cache*) **then**
     3. alloca una nuova pagina di memoria e registrala nella *Page Cache*
     4. riempi la pagina allocata con la corrispondente porzione del file, caricando i blocchi necessari (p. es. quattro se i blocchi fossero da 1 K byte) dal volume (*device*)

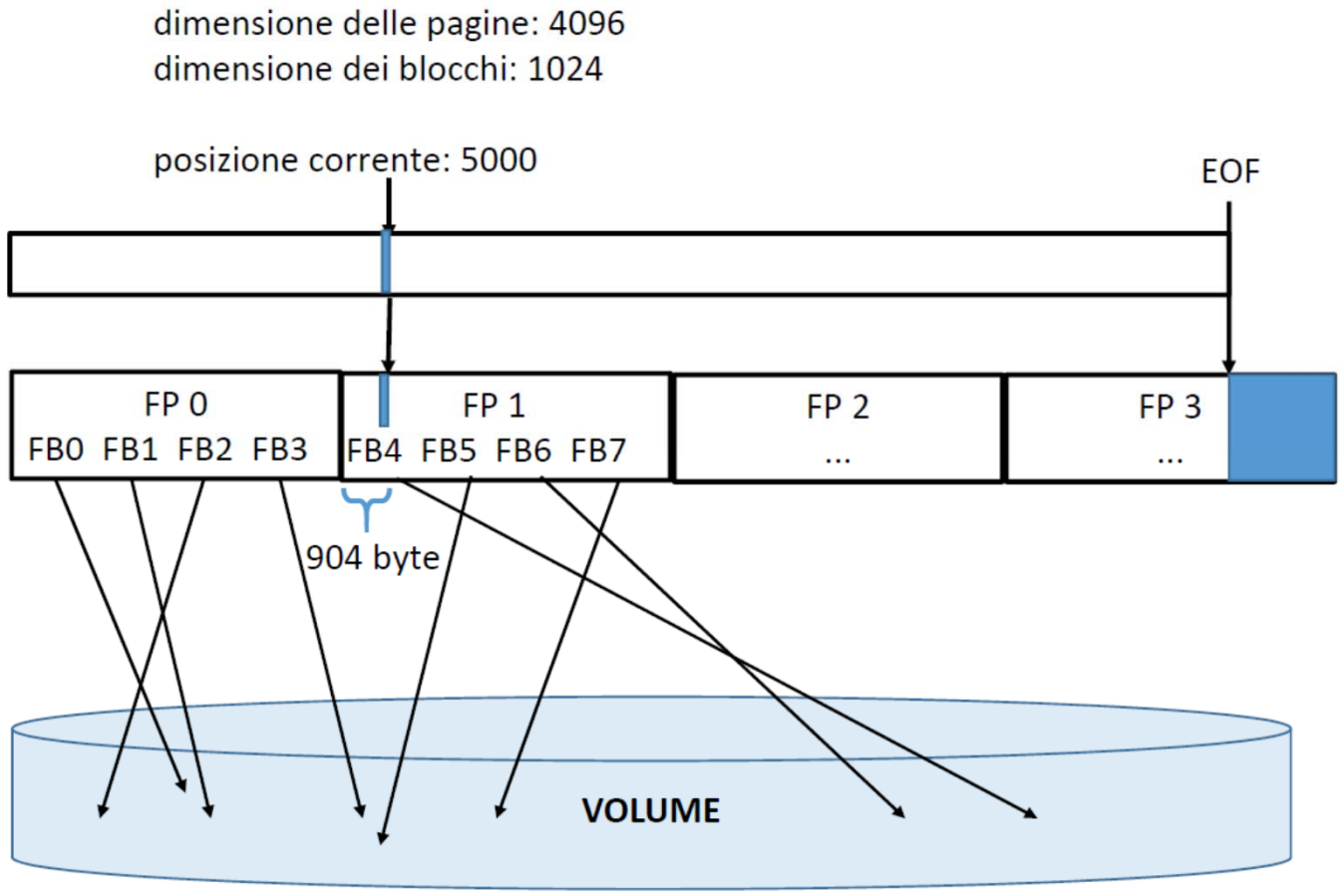
     end if
  5. copia i dati richiesti nello spazio di utente all'indirizzo richiesto dalla *system call*

trasformazione della posizione corrente in indirizzo sul volume

FP*n* = File Page *n*

n = POS / 4096
OFFSET = POS % 4096

la corrispondenza tra FBA (File Block Address) nel file fisico e LBA (Logical Block Address) sul volume dipende da come il volume stesso è stato strutturato da parte del FS specifico del volume



operazioni delegate alla *Page Cache*

- ecco il meccanismo di *Page Cache* per determinare se una pagina FP*n* di un file sia già in memoria

```
struct inode {
   ...
   struct address_space * i_mapping
   ...
} /* inode */
struct address_space {
   ...
   struct ... page_tree
   struct ... a_ops
   ...
} /* address_space */
```

- *page_tree* è una struttura ad albero (*radix tree*) che punta a tutte le pagine di *Page Cache* relative al file
- dato un numero di pagina FP*n* relativo a un file, la *Page Cache* verifica se tale pagina sia già in memoria
  - accede allo *i-node* del file
  - dallo *i-node* accede a *page_tree*
  - cerca la pagina FP*n* in *page_tree*
- se la pagina FP*n* non è in memoria, allora è necessario caricarla (**operazione su disco**)
- *a_ops* contiene operazioni specifiche del FS per accedere le pagine (***readpage*** e ***writepage***)

convenzioni per esercizi

- ecco i possibili eventi di *filesystem*, che impattano sulla memoria fisica

fd = *Open* (F)	*fd* è il descrittore di file e F è il nome del file
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 dare alla posizione corrente
Close (*fd*)	*fd* è il descrittore del file da chiudere (non cancellare !)

  - tutte queste operazioni hanno il comportamento normale, tranne *close*
  - si suppone che ***i dati vengano scritti su disco se close riduce f_count a 0***
- ecco i risultati principali da esibire
  - stato della memoria fisica (descrittori di pagina fisica)
  - campi principali delle strutture dati: *f_pos* e *f_count* (vedi *struct file*)
  - numero complessivo di accessi alle pagine del volume, in lettura e scrittura

esercizio 1 – 1

si consideri il seguente stato della memoria, mentre è in esecuzione il processo P

```
Parametri generali: MAX_FREE: 3  MIN_FREE: 2  MEM_SIZE: 8
======= Stato iniziale (descrittori di pagine fisiche): ======
____MEMORIA FISICA____(pagine libere: 5)_____________________________
      00 : <ZP>                  ||  01 : Pc1 / <X,1>          ||
      02 : Pp0                   ||  03 : ----                 ||
      04 : ----                  ||  05 : ----                 ||
      06 : ----                  ||  07 : ----                 ||
```

indicare: stato della memoria, posizione corrente (*f_pos*) e numero di riferimenti (*f_count*) per il file, e numero totale di accessi a pagine del disco in lettura e scrittura, dopo ciascuno 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*)

esercizio 1 – 2

1)**** processo P - **Open** (F) **********

```
____MEMORIA FISICA____(pagine libere: 5)_____________________________
      00 : <ZP>                    ||  01 : Pc1 / <X,1>          ||
      02 : Pp0                     ||  03 : ----                 ||
      04 : ----                    ||  05 : ----                 ||
      06 : ----                    ||  07 : ----                 ||
```

f_pos: **0** -- f_count: **1**

Numero di accessi a pagine del DISCO: nessun accesso (a pagine dati di F)

lo i-node del file è stato acquisito, da disco o era già in memoria – non lo contiamo

per ora in memoria fisica non è cambiato niente rispetto allo stato iniziale

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

esercizio 1 – 3

```
3)**** processo P – Lseek (fd,–200) **********************
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                    ||  01 : Pc1 / <X,1>          ||
      02 : Pp0                     ||  03 : <F,0>                ||
      04 : <F,1>                   ||  05 : ----                 ||
      06 : ----                    ||  07 : ----                 ||
f_pos: 3900 -- f_count: 1
```

in memoria fisica non è cambiato niente rispetto allo stato raggiunto dopo l'evento 2

```
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

esercizio 1 – 4

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

contiamo le letture cumulativamente: 2 letture di prima + 1 ora = 3 (idem scritture)

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

altri esercizi su *filesystem* in capitolo dispensa

extended filesystem – ext2 ext3 ed *ext4*

nome modello	**max dim. file**	**max dim. di partizione**	**differenze principali**	**anno di rilascio**
ext2	2 T byte	32 T byte		1993
ext3	2 T byte	32 T byte	ext2 + journaling	2001
ext4	16 T byte	1 E byte ($= 10^6$ T byte)	ext3 + extent	2008

organizzazione del volume *ext2* (inizialmente semplificato)

- ***superblock***: contiene informazioni globali sul volume (è eventualmente utilizzato in fase di *boot*)
  - sta sul volume in una posizione prefissata (di solito all'inizio del volume) e quindi nota al FS
  - a partire dal *superblock* sono raggiungibili: la tabella degli *i-node*, la radice del (sotto)albero dei cataloghi contenuto nel volume, e la lista libera
- ***tabella degli i-node* (*i-list*)**
  - contiene tutti gli *i-node* dei file esistenti (e anche tutti gli *i-node* liberi)
  - gli *i-node* vi sono memorizzati in sequenza e quindi sono reperibili in base al loro numero e alla loro dimensione (*i-node_size*)
- ***i-node***: contiene l'informazione relativa a un singolo file, in particolare i puntatori ai blocchi dati
- ***directory***: i cataloghi presenti sul volume – sono file normali opportunamente strutturati
- ***blocchi dati***
  - contengono i dati che costituiscono i file dati
  - sono inizialmente organizzati in un albero, la **lista libera**
  - vengono prelevati dall'albero per essere inseriti nei file
  - pertanto in un dato istante ogni blocco dati appartiene a un file oppure alla lista libera
  - quando un file viene eliminato o decresce liberando blocchi, questi tornano in lista libera

i-node (in *ext2*)

- un file esiste nel sistema quando esiste il suo *i-node*
- tutti gli *i-node* di un FS sono memorizzati sul volume in una tabella, detta *i-list*
- il riferimento fisico a un file è costituito dal suo numero di *i-node*
  - il numero (o indice) dello *i-node* è anche chiamato *i-number* (1, 2, 3, ecc)
  - p.es., un catalogo contiene coppie ⟨*nome file*, *i-number* dello *i-node* del file⟩
- ecco i contenuti principali dello *i-node* di un file
  - 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 con *creat* e *link* (di solito tale numero è uno, ma può essere maggiore)
  - la dimensione (in byte) del file (per i file di tipo normale o catalogo)
  - i puntatori ai blocchi dati del volume che costituiscono il file (di tipo normale o catalogo)
  - il tipo di file speciale non ha blocchi dati e consiste solo in un *i-node* contenente (in luogo dei puntatori ai blocchi dati) i numeri *major e minor*
- dimensione del blocco di volume / file
  - tra 1 K byte e 64 K byte, ma inferiori alla dimensione di pagina
  - nell'architettura x64 sono ammessi solo tre valori: 1, 2 o 4 K byte

i-node e accesso ai blocchi dati del file (in *ext2*)

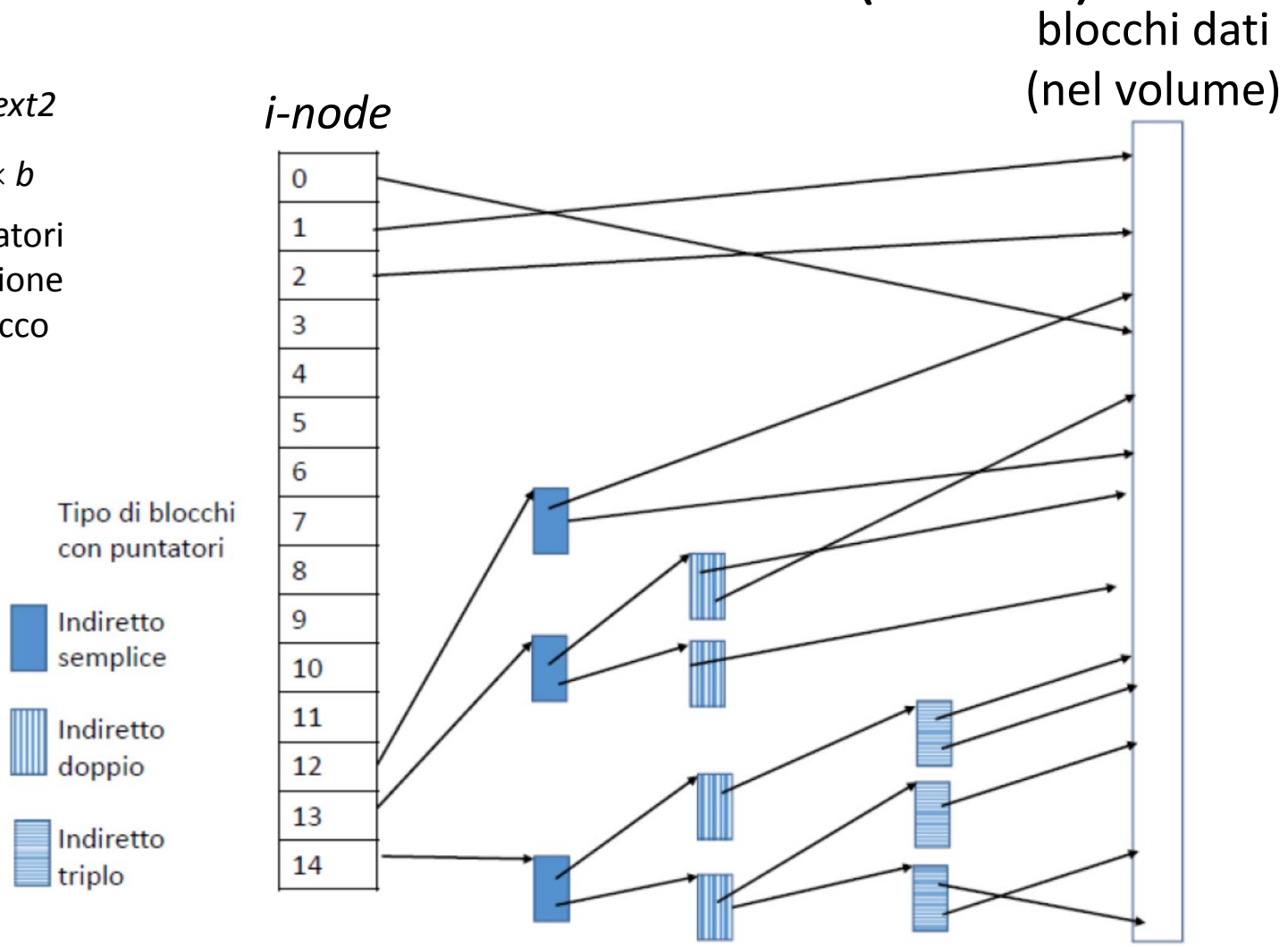
massima dimensione di un file in *ext2*

$$\left((b / 4)^3 + (b / 4)^2 + b / 4 + 12 \right) \times b$$

dove $b / 4$ è il numero di puntatori contenuto in un blocco di indirezione e dipende dalla dimensione di blocco

sul volume, in una serie di blocchi consecutivi riservata allo scopo (circa il 10 % del totale di blocchi disponibili), sta memorizzata l'intera *i-list* contenente tutti gli *i-node* dei file presenti nel FS

una parte della *i-list* e delle *struct dentry* sta in memoria (nelle *cache i-node* e *dentry*)



suddivisione del volume in *block group* (in *ext2*)

- il meccanismo di *block group* mira a memorizzare in blocchi contigui le informazioni correlate tra loro, in base all'ipotesi che esse vengano accedute insieme
- *block group*: è una serie di blocchi indicizzata da un intervallo continuo di LBA
  - ipotesi: LBA numericamente più vicini lo sono anche in termini di tempi di accesso
  - allora un *block group* costituisce un insieme di blocchi che sono accessibili insieme più rapidamente di un insieme casuale di blocchi
- è compito del dispositivo e del suo *driver* fare in modo che questa ipotesi sia verificata
- la dimensione dei *block group* in blocchi è determinata dal numero di byte di un blocco:
  - se dim. blocco = 1 K byte, la dimensione del gruppo è 8×1 K = 8192 blocchi
  - se dim. blocco = 4 K byte, la dimensione è 32768
- i gruppi sono numerati partendo da 0 e sono consecutivi, senza interruzioni tra di loro
- il FS tenta di allocare tutti i blocchi di un file nello stesso *block group* del catalogo

contenuto del *block group* (in *ext2*)

- il *superblock* del volume è logicamente uno solo, ma per ragioni di affidabilità può essere replicato in numerosi *block group*
- la tabella degli *i-node* (*i-list*) è logicamente una sola ed è suddivisa in parti uguali nei diversi *block group*
  - il parametro *inodes_per_group* indica il numero di *i-node* assegnato a ciascuna porzione di tabella degli *i-node*
  - è possibile risalire da un numero di *i-node* (*inode_number*) al gruppo *bg* nella cui tabella è definito, tramite la seguente trasformazione (gli *i-node* sono numerati da 1, non da 0)

$$bg = (inode\_number - 1) \,/\, inodes\_per\_group$$

- lo specifico *i-node* si troverà nella porzione di tabella degli *i-node* del gruppo nella posizione indicata dallo spiazzamento seguente

$$offset = [(inode\_number - 1) \,\%\, inodes\_per\_group] \times inode\_size$$

meccanismo di *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 dello *extent*
  - la dimensione dello *extent*
  - il blocco del volume (LBA) di inizio dello *extent*
- vantaggi del meccanismo di *extent*
  - riduce il numero di puntatori necessari (ma questo non è molto significativo, perché lo spazio occupato dai puntatori è solo una piccola parte di quello occupato dai dati)
  - migliora le prestazioni, dato che non richiede di gestire i puntatori, anche indiretti
  - favorisce l'allocazione contigua dei file, che è molto utile specialmente per file utilizzati sequenzialmente
  - aumenta la dimensione massima dei file fisicamente mappabili

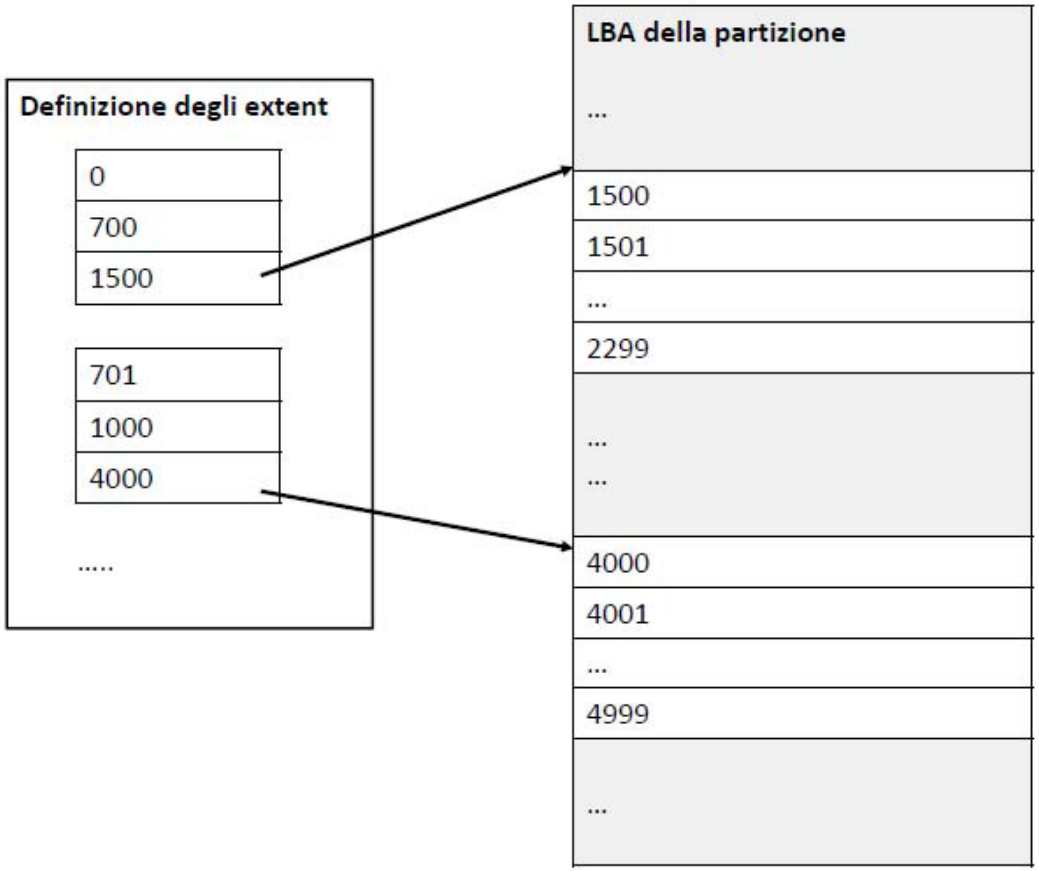
extent – esempio

primo *extent*

- 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

secondo *extent*

- 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



device driver – gestore di periferica

- i *device driver* (gestori di periferica) sono moduli software che realizzano l'interfacciamento e la gestione dei dispositivi periferici
- interagiscono con il *filesystem* perché tutte le periferiche sono viste come file speciali
- interagiscono con il nucleo del sistema operativo per gestire la sincronizzazione tra periferica e calcolatore, e il trasferimento dati tra i due
- sono la parte di sistema operativo che viene aggiornata con frequenza maggiore
- in LINUX c'è un *device driver* (gestore) per ogni tipo di periferica installata

file speciale e *device driver*

- ciascun dispositivo è associato a un file speciale (blocco o carattere) ed è identificato da una coppia di numeri ⟨*major*, *minor*⟩
- solo l'amministratore di sistema (*root*) può creare file speciali, tramite la funzione

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

- i numeri *major* e *minor* sono contenuti nello *i-node* che rappresenta il file speciale (lo *i-node* di un file speciale non contiene puntatori a blocchi di dati, dato che questi non esistono)
- tutte le periferiche dello stesso tipo, cioè gestite dallo stesso *driver*, hanno lo stesso numero *major* e pertanto condividono gli stessi servizi
- l'accesso alle periferiche è attuato tramite le chiamate di accesso a file (*open*, *close*, *read*, *write*, ecc) con specificato il descrittore relativo al file speciale
- l'esecuzione del servizio richiesto è parametrizzata tramite il numero *minor*

struttura del *device driver* – 1

- le funzioni principali di un *device driver* sono
  - inizializzazione del dispositivo alla partenza del sistema operativo
  - gestione dello stato della periferica (in servizio / fuori servizio)
  - ricezione e / o invio di dati dalla / verso la periferica
  - gestione degli errori
  - gestione degli interrupt da periferica
- ogni *device driver* può essere visto come costituito da
  - una routine di inizializzazione che esegue delle operazioni di inizializzazione del *driver*
  - un insieme di routine che costituiscono i servizi eseguibili e implementati per quel tipo di periferica
  - la routine di risposta attivata dall'interrupt lanciata dalla periferica, il cui indirizzo viene inserito nel corrispondente *vettore di interrupt*

struttura del *device driver* – 2

ciascun *device driver* ha associata una «tabella delle operazioni», realizzata tramite la *struct file_operations*, che contiene i puntatori alle routine di servizio del *driver* stesso

```
struct file_operations {
     int  (* lseek)   ( )  // modifica posizione corrente
     int  (* read)    ( )  // leggi da file o periferica
     int  (* write)   ( )  // scrivi su file o periferica
     ...
     int  (* ioctl)   ( )  // comando di I/O speciale
     ...
     int  (* open)    ( )  // apri file o periferica
     void (* release) ( )  // rilascia file o periferica
     ...
} / * file_operations */
```

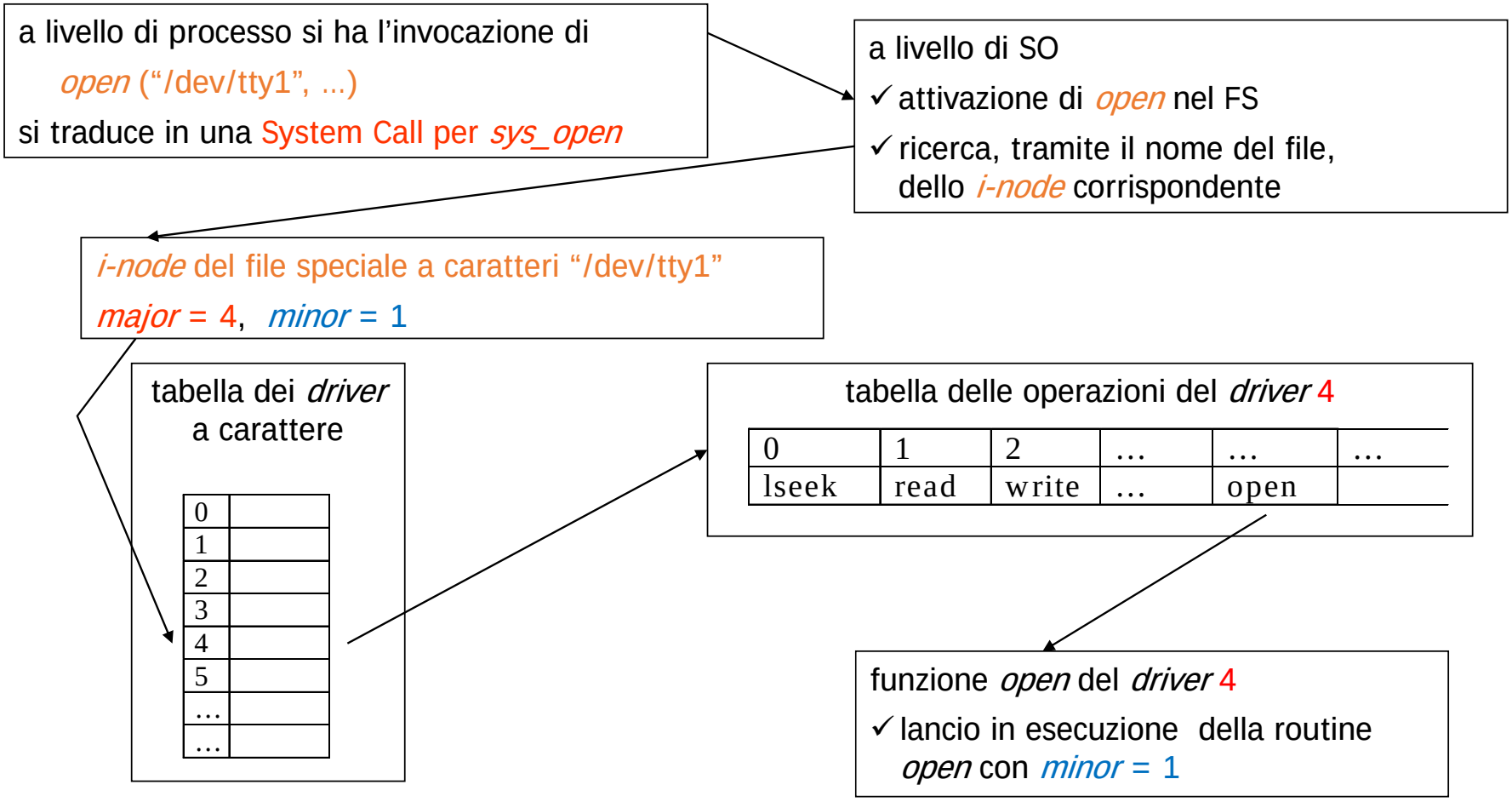
strutture dati per accedere alle funzioni del *device driver*

- alla partenza del SO viene attivata una funzione di inizializzazione per ogni *device driver* installato
- la funzione restituisce al nucleo il puntatore alla propria tabella delle operazioni
- l'interfaccia tra sistema operativo e *driver* è descritta tramite due tabelle
  - *block device switch table* tabella dei *driver* per i dispositivi a *blocchi*
  - *character device switch table* tabella dei *driver* per i dispositivi a *carattere*
- ciascun tipo di dispositivo ha una riga, nella tabella appropriata, che indirizza al *device driver* corrispondente

chiamata di sistema e *device driver*

- le chiamate di sistema fanno riferimento a un descrittore di file (o al nome) che consente, tramite la tabella dei file aperti, di identificare lo *i-node* corrispondente
- lo *i-node* identifica il tipo (a carattere o a blocchi) di file speciale e indirizza alla riga corretta della tabella dei *driver* per dispositivi a blocchi o a quella per dispositivi a carattere tramite il numero *major* contenuto nello *i-node* stesso
- nel *driver* a carattere il servizio richiesto identifica la colonna della tabella delle operazioni associata al driver
- al servizio viene passato come parametro il numero *minor,* anch'esso contenuto nello *i-node*, per l'identificazione univoca del dispositivo

indirizzamento della routine di servizio del *device driver*

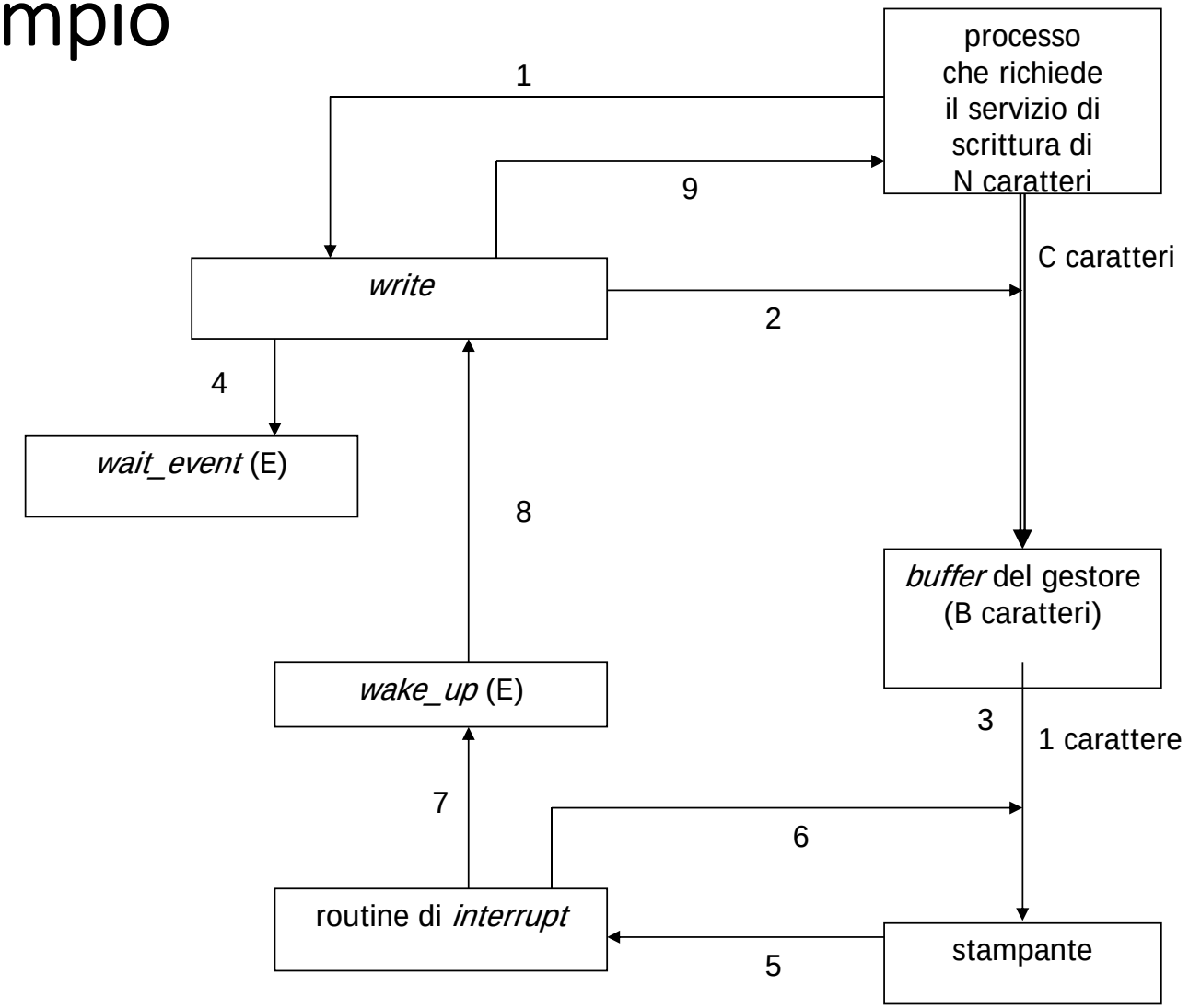


principio di funzionamento per *device driver* a carattere

lettura e scrittura

- nel caso di periferiche gestite con *interrupt*, l'interrupt si verifica nel contesto di un processo diverso da quello che ha invocato il servizio della periferica
- le routine del *device driver* possono memorizzare temporaneamente i dati che devono inviare (o ricevere) alla (dalla) periferica in un *buffer del driver* appositamente allocato nella memoria del SO (*kernelspace*)

esempio


processo
che richiede
il servizio di
scrittura di
N caratteri
1
9
C caratteri
write
2
4
wait_event (E)
8
buffer del gestore
(B caratteri)
wake_up (E)
3
1 carattere
7
6
routine di interrupt
5
stampante