Università di Ferrara
Laurea Triennale in Informatica
A.A. 2021-2022

Sistemi Operativi e Laboratorio

8. Il File System di Unix

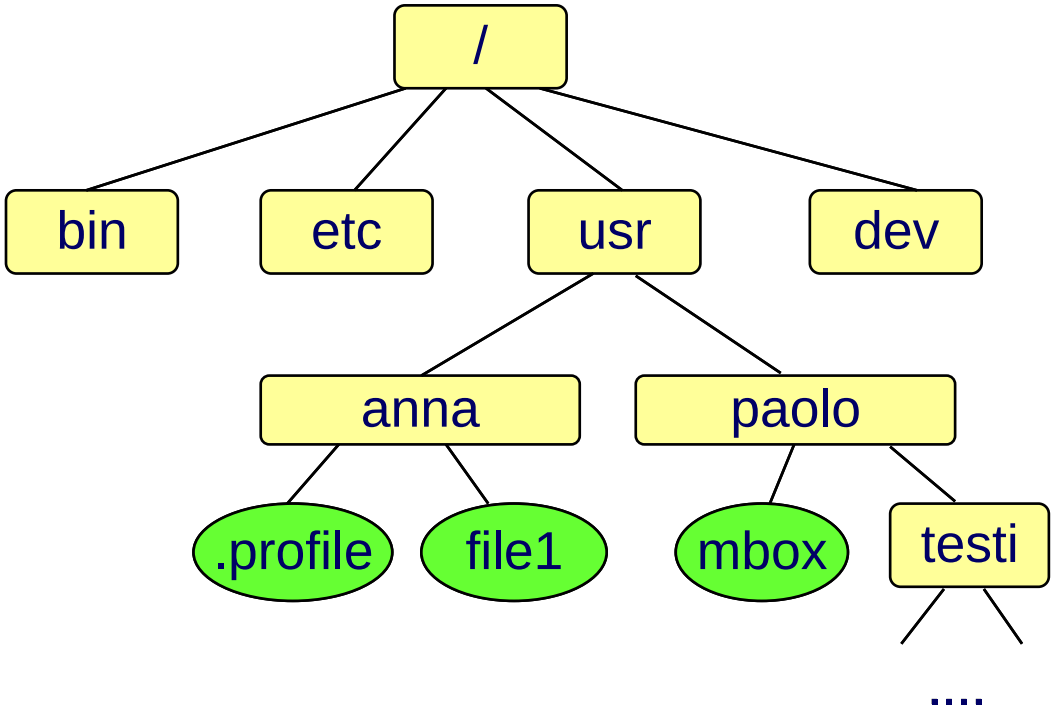
Prof. Carlo Giannelli

http://www.unife.it/scienze/informatica/insegnamenti/sistemi-operativi-laboratorio

http://docente.unife.it/carlo.giannelli

https://ds.unife.it/people/carlo.giannelli/

UNIX file system: organizzazione logica



- **Omogeneità**: tutto è file
- Tre categorie di file:
  - file **ordinari**
  - **direttori**
  - **dispositivi fisici**: file speciali (nel direttorio ***/dev***)

Nome, i-number, i-node

- **Ad ogni file** possono essere associati **uno o più nomi simbolici**

ma

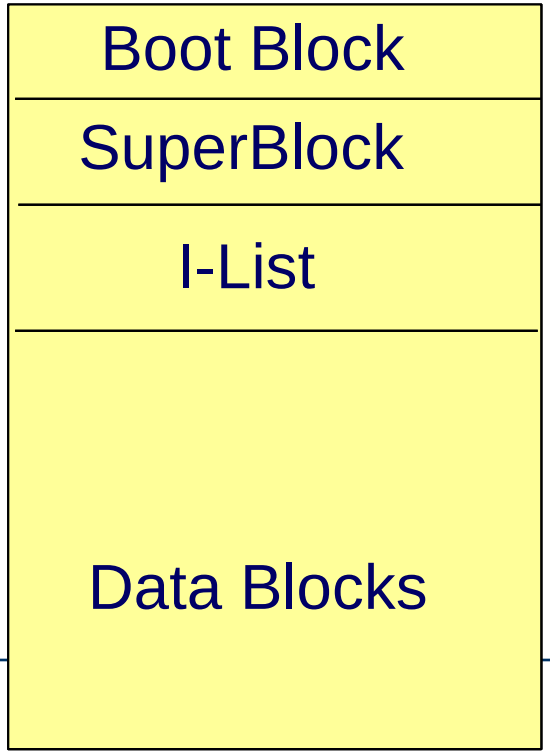
- **Ad ogni file** è associato **uno ed un solo descrittore (i-node),** univocamente identificato da un **intero (i-number)**

UNIX file system: organizzazione fisica

- **Metodo di allocazione** utilizzato in UNIX è a **indice** (a più livelli di indirizzamento)

- Formattazione del disco in **blocchi fisici (dimensione del blocco: 512B-4096B)**

- Superficie del disco partizionata in **4 regioni:**
  - **boot block**
  - **super block**
  - **i-list**
  - **data blocks**

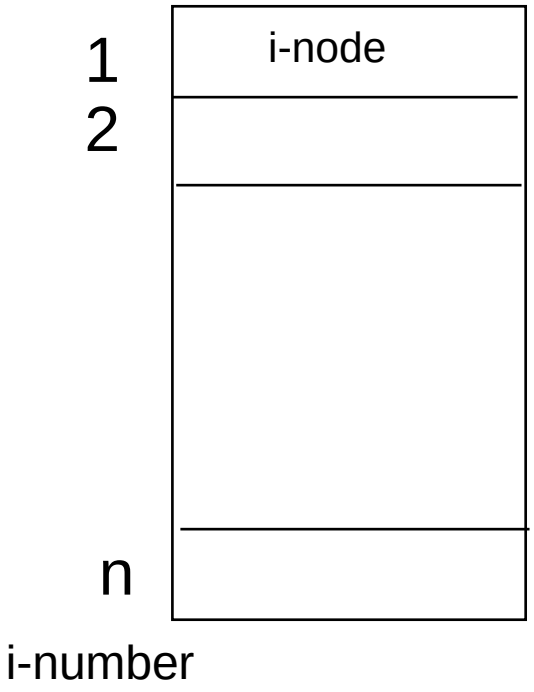


UNIX file system: organizzazione fisica

- **Boot Block** contiene le **procedure di inizializzazione** del sistema (da eseguire al bootstrap)
- **Super Block** fornisce
  - i limiti delle 4 regioni
  - il puntatore a una **lista dei blocchi liberi**
  - il puntatore a una **lista degli i-node liberi**
- **Data Blocks** - area del disco effettivamente disponibile per la memorizzazione dei file. Contiene:
    - i **blocchi allocati**
    - i **blocchi liberi** (organizzati in una **lista collegata**)

UNIX file system: organizzazione fisica

i-List contiene la **lista di tutti i descrittori (i-node)** dei file normali, direttori e dispositivi presenti nel file system (accesso con l'indice **i-number**)



i-node

i-node è il descrittore del file

Tra gli **attributi** contenuti nell'i-node vi sono:

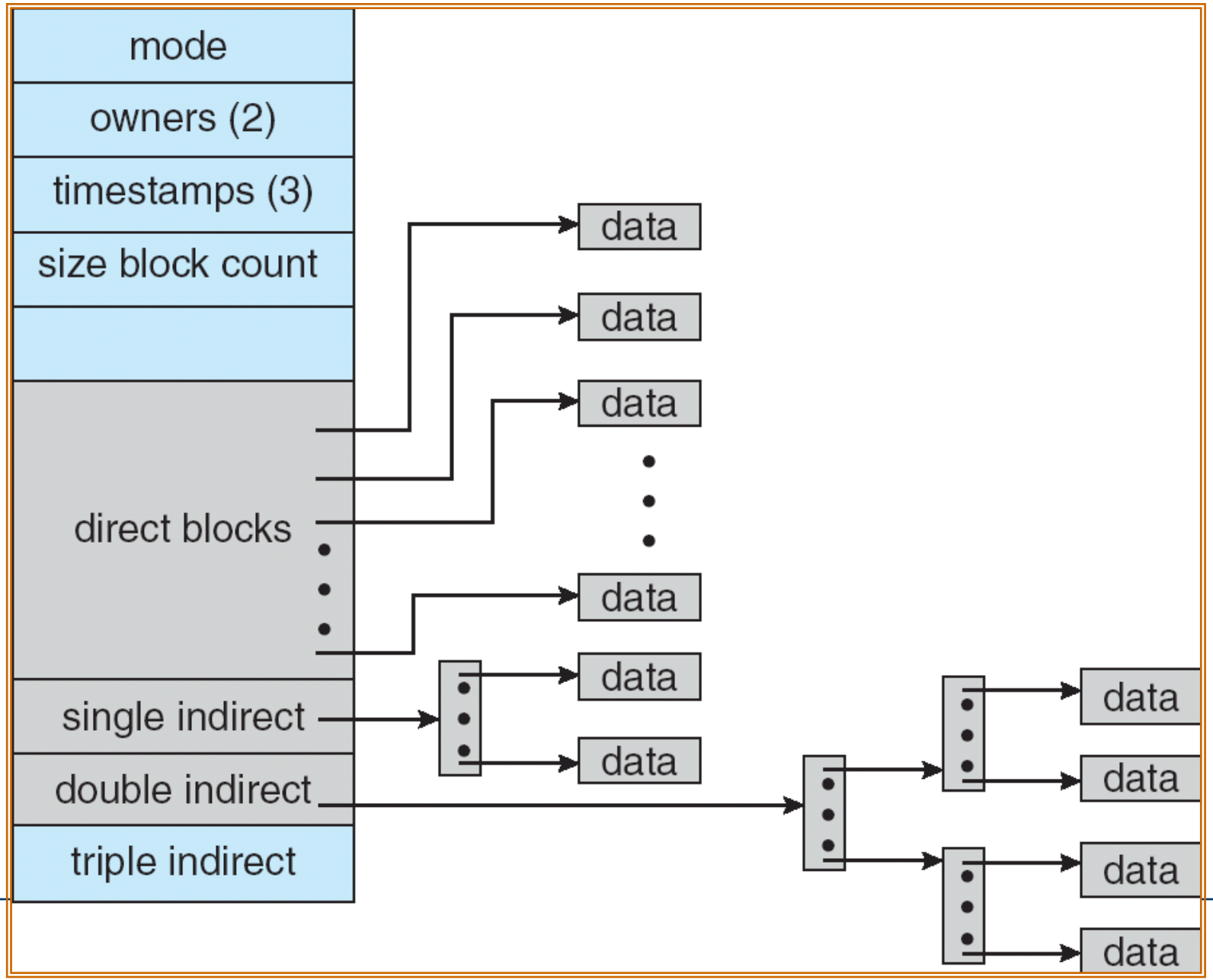
- **tipo** di file:
  - **ordinario**
  - **direttorio**
  - file **speciale**, per i dispositivi
- **proprietario, gruppo** (user-id, group-id)
- **dimensione**
- **data**
- 12 bit di **protezione**
- numero di **link**
- **13-15 indirizzi** di blocchi (a seconda della realizzazione)

Indirizzamento

L'allocazione del file **NON è** su blocchi fisicamente contigui. **Nell'i-node sono contenuti puntatori a blocchi** (ad esempio **13**)**,** dei quali:

- **i primi 10 indirizzi** riferiscono blocchi di dati (indirizzamento diretto)
- **11° indirizzo**: indirizzo di un blocco contenente a sua volta indirizzi di blocchi dati (primo livello di indirettezza)
- **12° indirizzo**: secondo livello di indirettezza
- **13° indirizzo**: terzo livello di indirettezza

Indirizzamento



Indirizzamento

Se: dimensione del blocco **512B**=0,5KB
indirizzi di 32 bit (4 byte)
→ 1 blocco contiene **128** indirizzi

- **10 blocchi di dati sono accessibili direttamente**
  file di dimensioni minori di 10*512 B=5KB sono accessibili direttamente
- **128 blocchi di dati sono sono accessibili con indirettezza singola (mediante il puntatore 11)**: 128*512 B=64KB
- **128*128 blocchi di dati sono accessibili con indirettezza doppia (mediante il puntatore 12):** 128*128*512 B=8MB
- **128*128*128 blocchi di dati sono accessibili con indirettezza tripla (mediante il puntatore 13):**
  128*128*128*512 B=1GB

Indirizzamento

La dimensione massima del file realizzabile è dell'ordine del **GB**

Dimensione massima = 1GB+ 8MB+64KB+5KB

→ vantaggio aggiuntivo: **l'accesso a file di piccole dimensioni è più veloce** rispetto al caso di file grandi

Breve parentesi su Linux file system

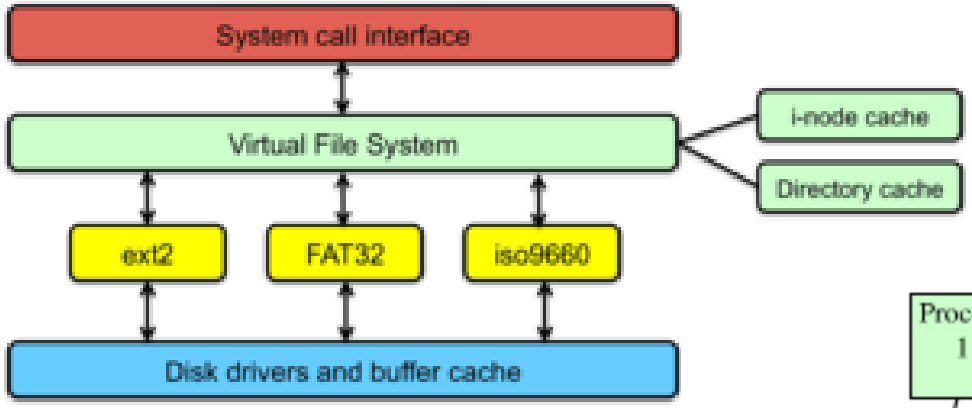
All’utente finale il file system Linux appare come una struttura ad albero gerarchico che **obbedisce alla semantica UNIX**

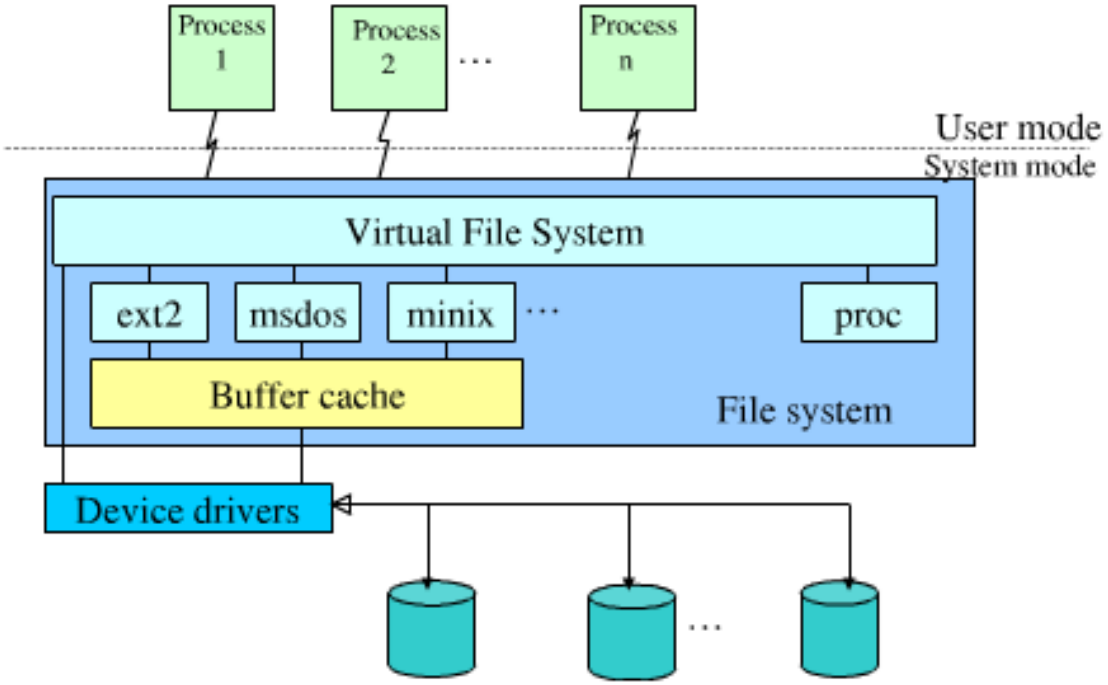
Internamente, il kernel di Linux è capace di gestire **differenti file system multipli** fornendo un livello di **astrazione uniforme: Virtual File System (VFS)**

Linux VFS sfrutta:

- un insieme di **descrittori** che definiscono come un file debba **apparire** (inode object, file object, file system object)
- un insieme di funzionalità software per gestire questi oggetti

Virtual File System (VFS)





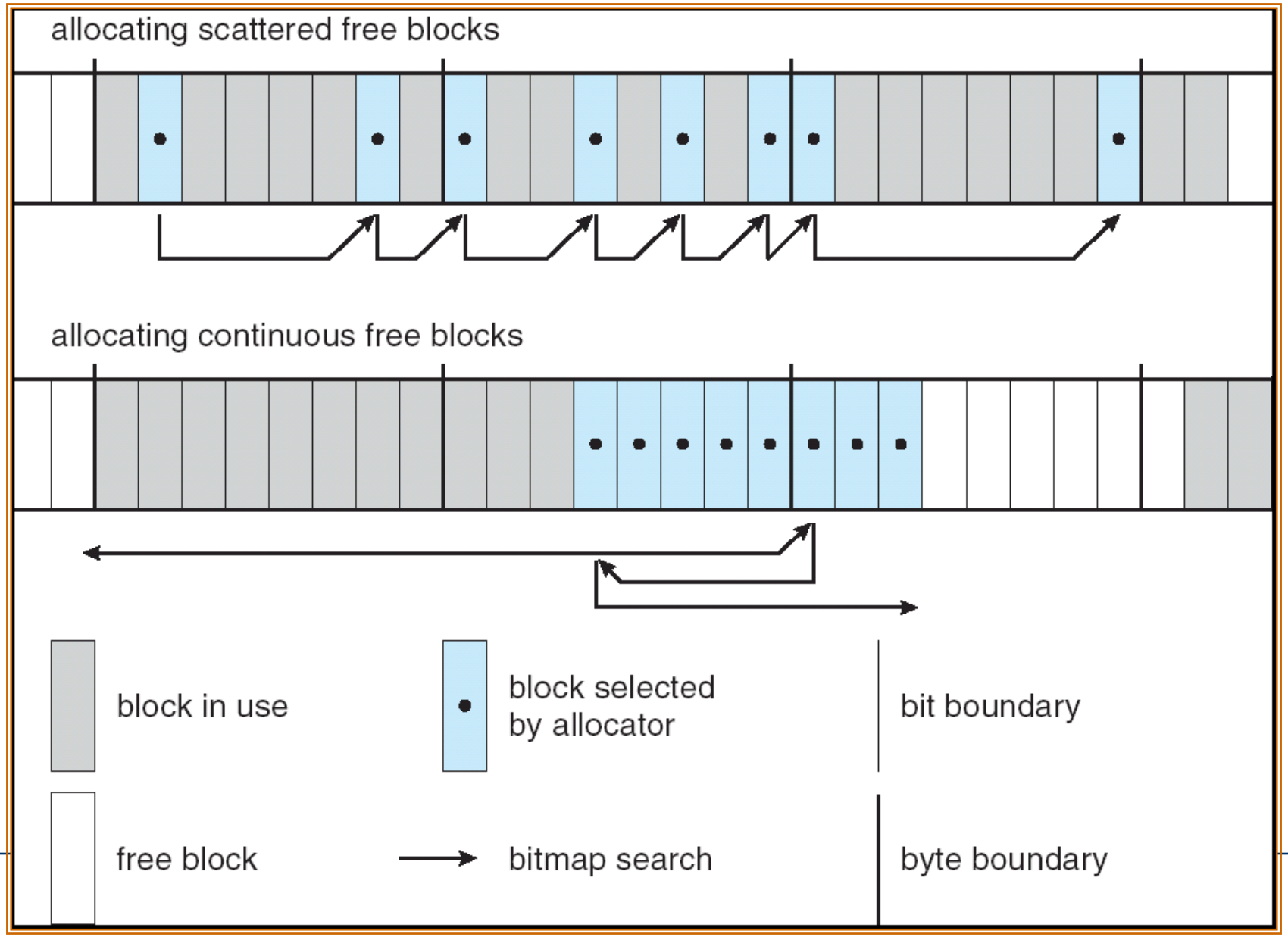
Linux Ext2fs

Ext2fs usa un meccanismo molto simile a quello standard UNIX per identificare i data block appartenenti a un file specifico

Le differenze principali riguardano le **strategie di allocazione dei blocchi:**

- Ext2fs usa default block size di 1KB, ma è anche in grado di supportare blocchi da 2KB e 4KB
- Ext2fs tenta di collocare **blocchi logicamente adiacenti** di un file su **blocchi fisicamente adiacenti su disco**. In questo modo, è teoricamente possibile realizzare **una singola operazione di I/O che operi su diversi blocchi logicamente adiacenti**

Linux Ext2fs



Linux: proc file system

- **proc** file system non serve a memorizzare dati, ma per **funzioni tipicamente di monitoraggio**
- È uno **pseudo-file system** che fornisce accesso alle **strutture dati di kernel** mantenute dal SO: i suoi contenuti sono tipicamente calcolati on demand
- **proc** realizza una **struttura a directory** e si occupa di **mantenerne i contenuti** invocando le funzioni di monitoring correlate

Esempi: **`version`**, **`dma`**, **`stat/cpu`**, **`stat/swap`**, **`stat/processes`**

Parentesi su MS file system 1/2

La struttura fondamentale per il **file system di MS WinXP (NTFS) è il volume**

- basato su una **partizione logica di disco**
- può occupare una porzione di disco, un disco intero, o differenti porzioni su differenti dischi

Tutti i descrittori (**metadati** riguardanti il volume) sono memorizzati in un **file normale**

NTFS usa **cluster come unità di allocazione dello spazio disco**

- un cluster è costituito da un numero di settori disco che è **potenza di 2**
- **frammentazione interna ridotta** rispetto a 16-bit FAT perchè dimensione cluster minore

Parentesi su MS file system 2/2

- Un file NTFS non è una semplice sequenza di byte, come in MS-DOS o UNIX, ma un **oggetto strutturato con attributi**
- Ogni file NTFS è descritto da **una o più entry di un array** memorizzato in un **file speciale** chiamato **Master File Table (MFT)**
- Ogni file NTFS su una unità disco ha **ID unico** (64 bit)
- Lo spazio dei nomi di NTFS è organizzato in una **gerarchia di directory**. Per motivi di efficienza, esiste un **indice per ottimizzare l'accesso (B+ tree)**

Directory

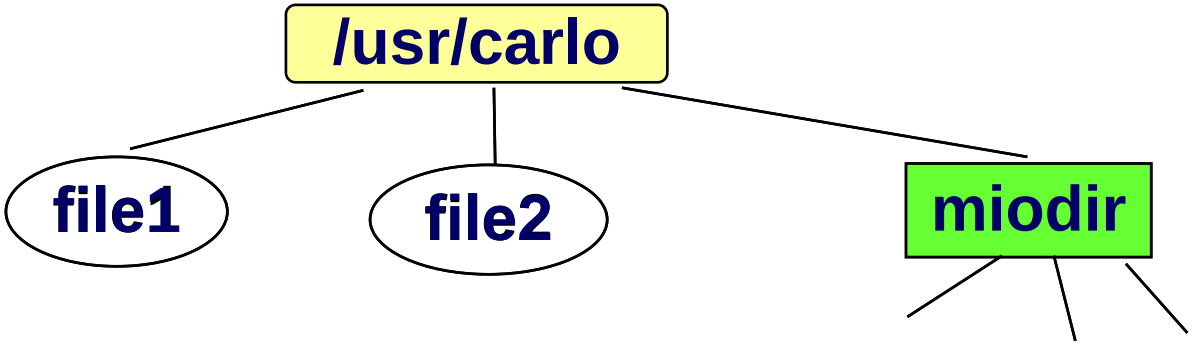
Anche le directory sono rappresentate nel file system da file

- Ogni file-directory contiene un insieme di **record logici con struttura**

nomerelativo	i-number

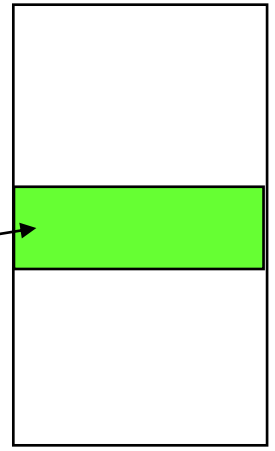
- Ogni **record rappresenta un file** appartenente alla directory
  - per ogni file (o directory) appartenente alla directory considerata, viene **memorizzato il suo nome relativo**, a cui viene associato il relativo **i-number** (che lo identifica univocamente)

Directory



file (directory) **`/usr/carlo`** contiene:

`file1`	`189`
`file2`	`133`
`miodir`	`121`
`.`	`110`
`..`	`89`



Gestione file system in UNIX

Quali sono i **meccanismi di accesso**
al file system?
Come vengono realizzati?

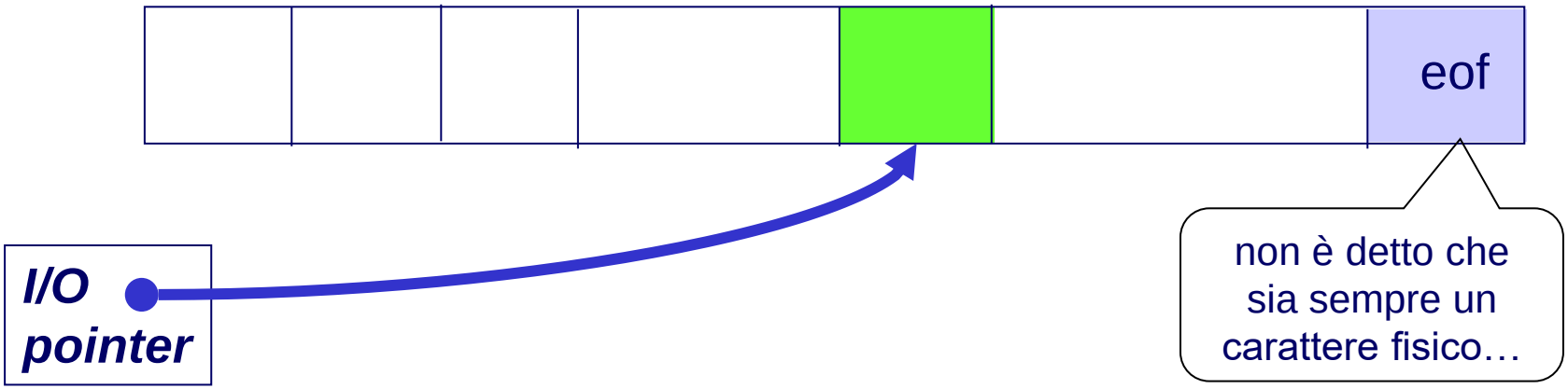
Vanno considerati:

- **strutture dati di sistema** per il supporto all'accesso e alla gestione di file
- **principali system call** per l'accesso e la gestione di file

Gestione file system: concetti generali

- **Accesso sequenziale**
- **Assenza di strutturazione**:

  file = sequenza di byte (stream)
- Posizione corrente: **I/O Pointer**



Gestione file system: concetti generali

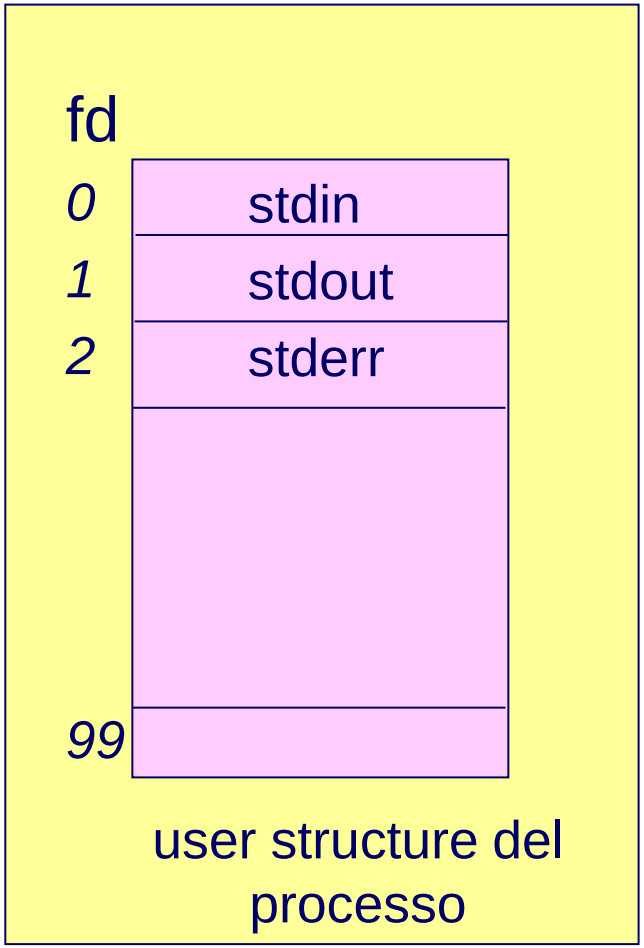
- **Varie modalità di accesso** (lettura, scrittura, lettura/scrittura, …)

- **Accesso subordinato all'operazione di apertura:**

<apertura File>
<accesso al File>
<chiusura File>

File descriptor

- A **ogni processo è associata una tabella dei file aperti** di dimensione limitata (attorno al centinaio di elementi)
- Ogni **elemento della tabella** rappresenta **un file aperto dal processo** ed è individuato da un indice intero: **file descriptor**
- I **file descriptor 0,1,2** individuano rispettivamente **standard input, output, error** (aperti automaticamente)
- La **tabella dei file aperti** del processo è allocata nella sua **user structure**



Strutture dati del kernel

Per realizzare l'accesso ai file, SO utilizza **due strutture dati globali**, allocate nell'area dati del **kernel**

- la **tabella dei file attivi**: per ogni file aperto, contiene una copia del suo i-node → **più efficienti le operazioni** su file evitando accessi al disco per ottenere attributi dei file acceduti
- la **tabella dei file aperti di sistema:** ha **un elemento per ogni operazione di apertura relativa a file aperti** (e non ancora chiusi). Ogni elemento contiene:
  - **I/O pointer**, posizione corrente all'interno del file
  - un puntatore all'**i-node** del file nella tabella dei file attivi

  → se **due processi aprono separatamente** lo stesso file F, la tabella conterrà **due elementi distinti associati a F**

Strutture dati del kernel

Riassumendo:

- **tabella dei file aperti di processo:** nella **user area** del processo, contiene **un elemento per ogni file aperto dal processo**
- **tabella dei file aperti di sistema:** in area di SO, contiene **un elemento per ogni sessione di accesso** a file nel sistema
- **tabella dei file attivi**: in area di SO, contiene **un elemento per ogni file aperto nel sistema**

Quali sono le relazioni tra queste strutture?

Strutture dati del kernel

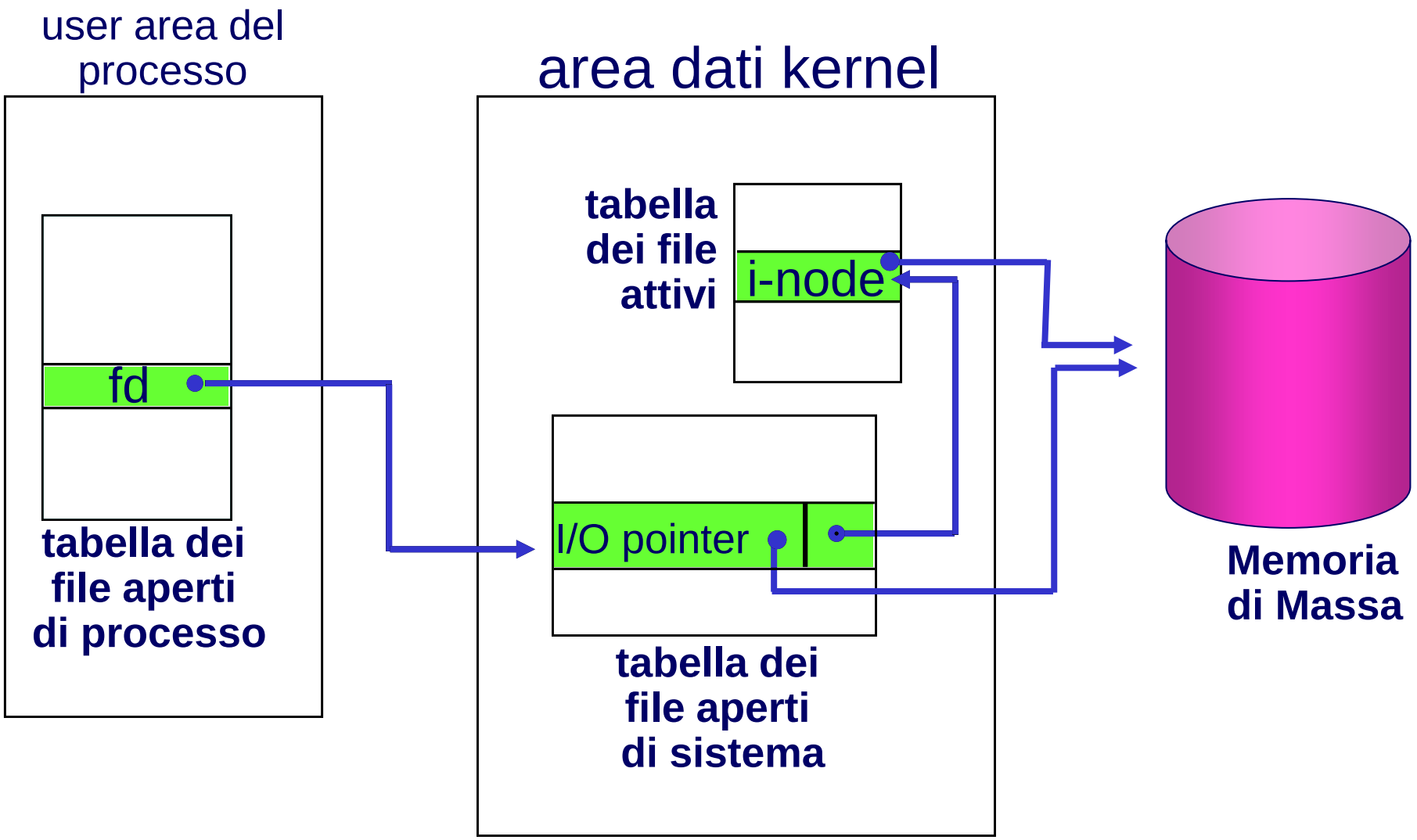


Tabella dei file aperti di sistema

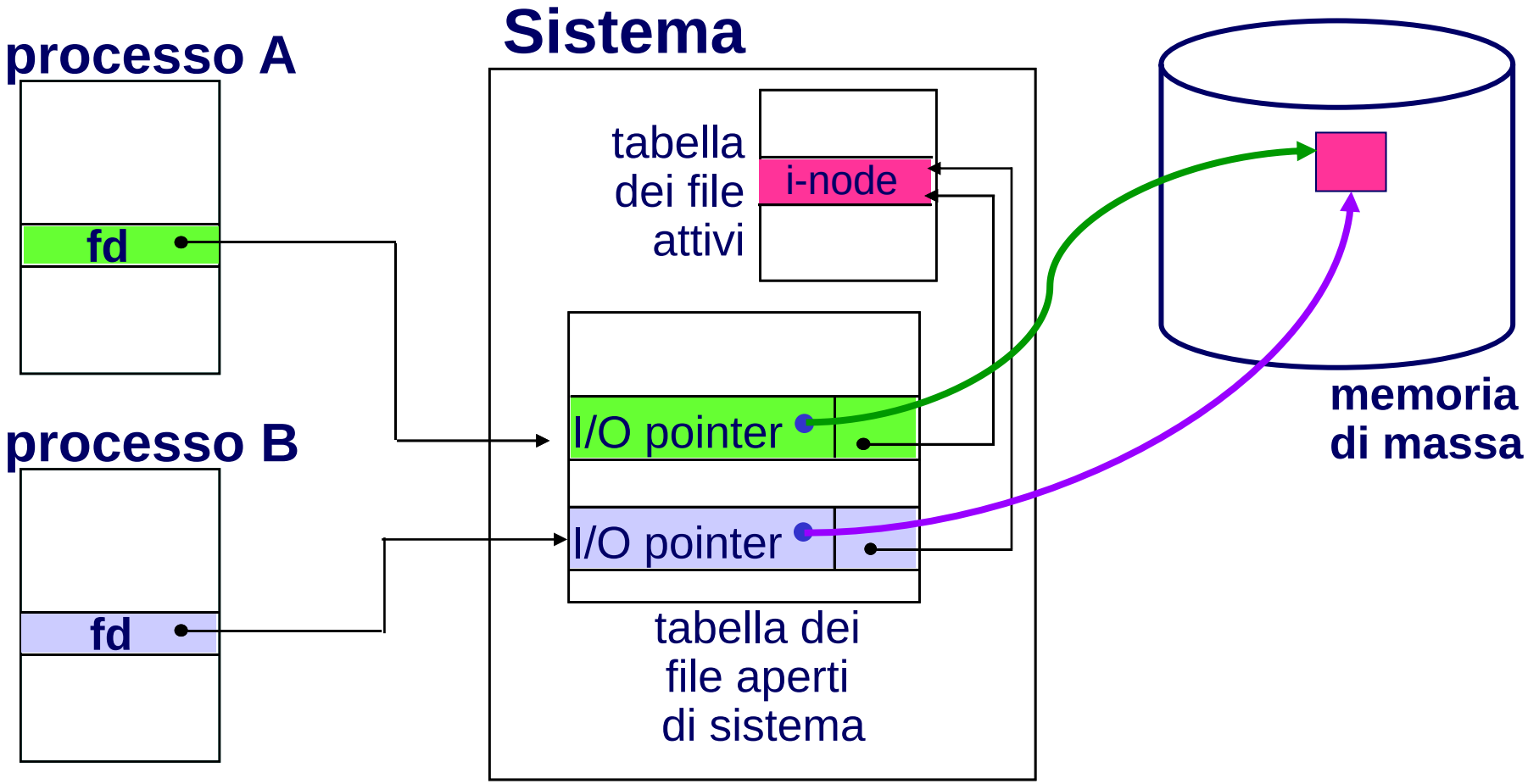
- **Un elemento per ogni "apertura"** di file: a processi diversi che accedono allo stesso file corrispondono **entry distinte**
- Ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)

più processi possono accedere contemporaneamente allo stesso file, ma hanno ***I/O pointer distinti***

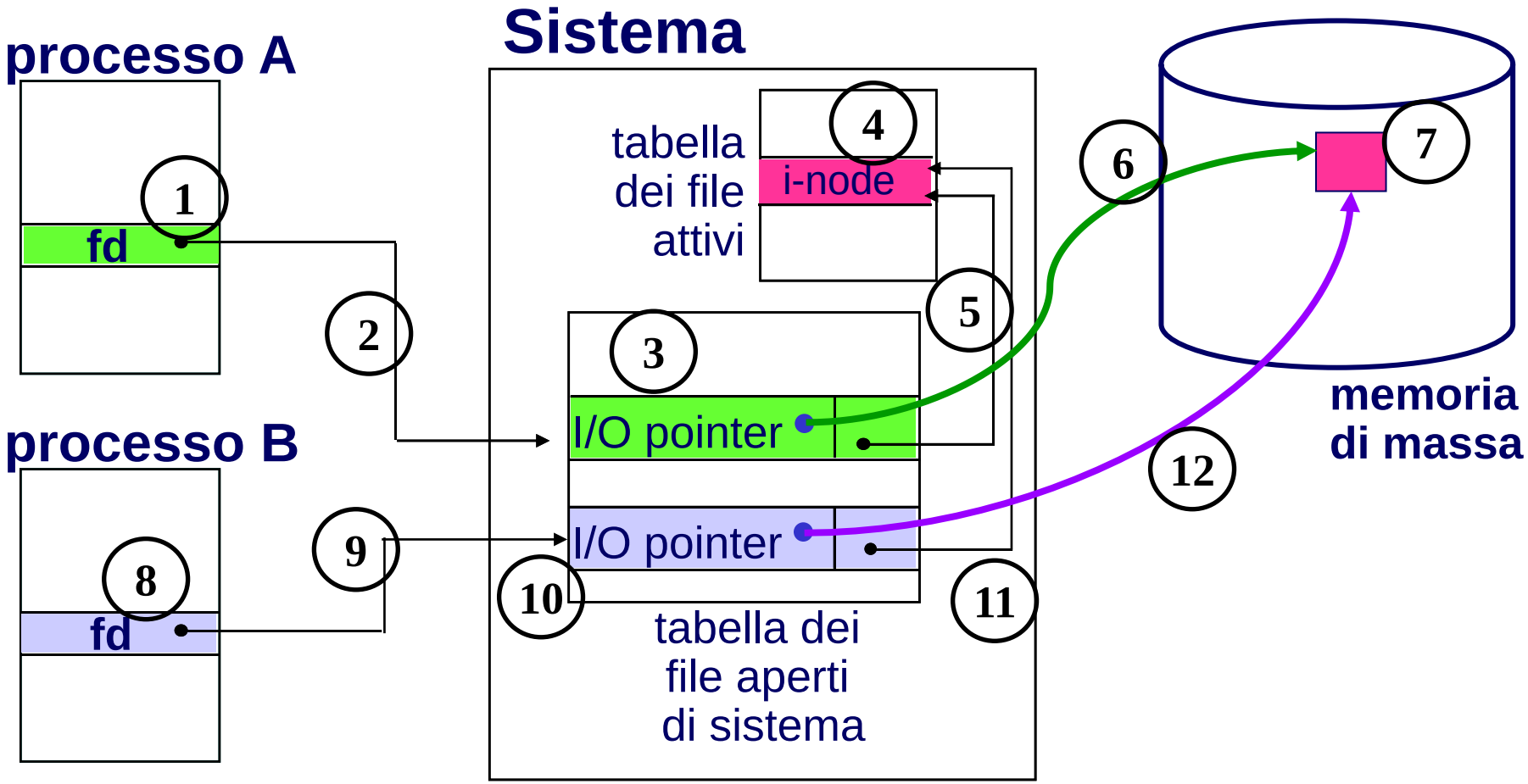
Tabella dei file attivi

- L'operazione di **apertura** provoca la **copia dell'i-node in memoria centrale** (se il file non è già in uso)
- La **tabella dei file attivi** contiene gli **i-node di tutti i file aperti**
- Il numero degli elementi è pari al numero dei file aperti (anche da più di un processo)

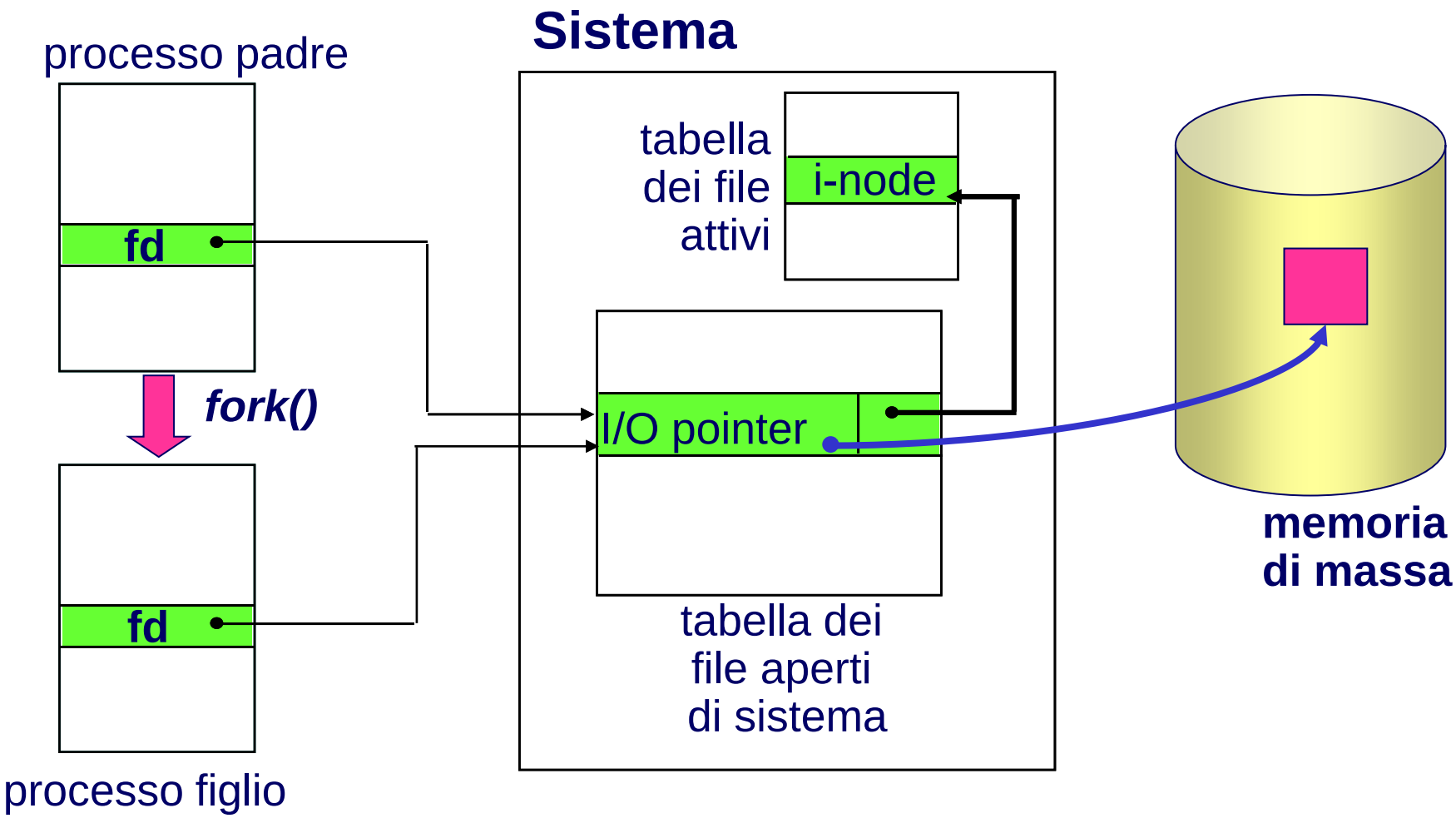
Esempio: processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti



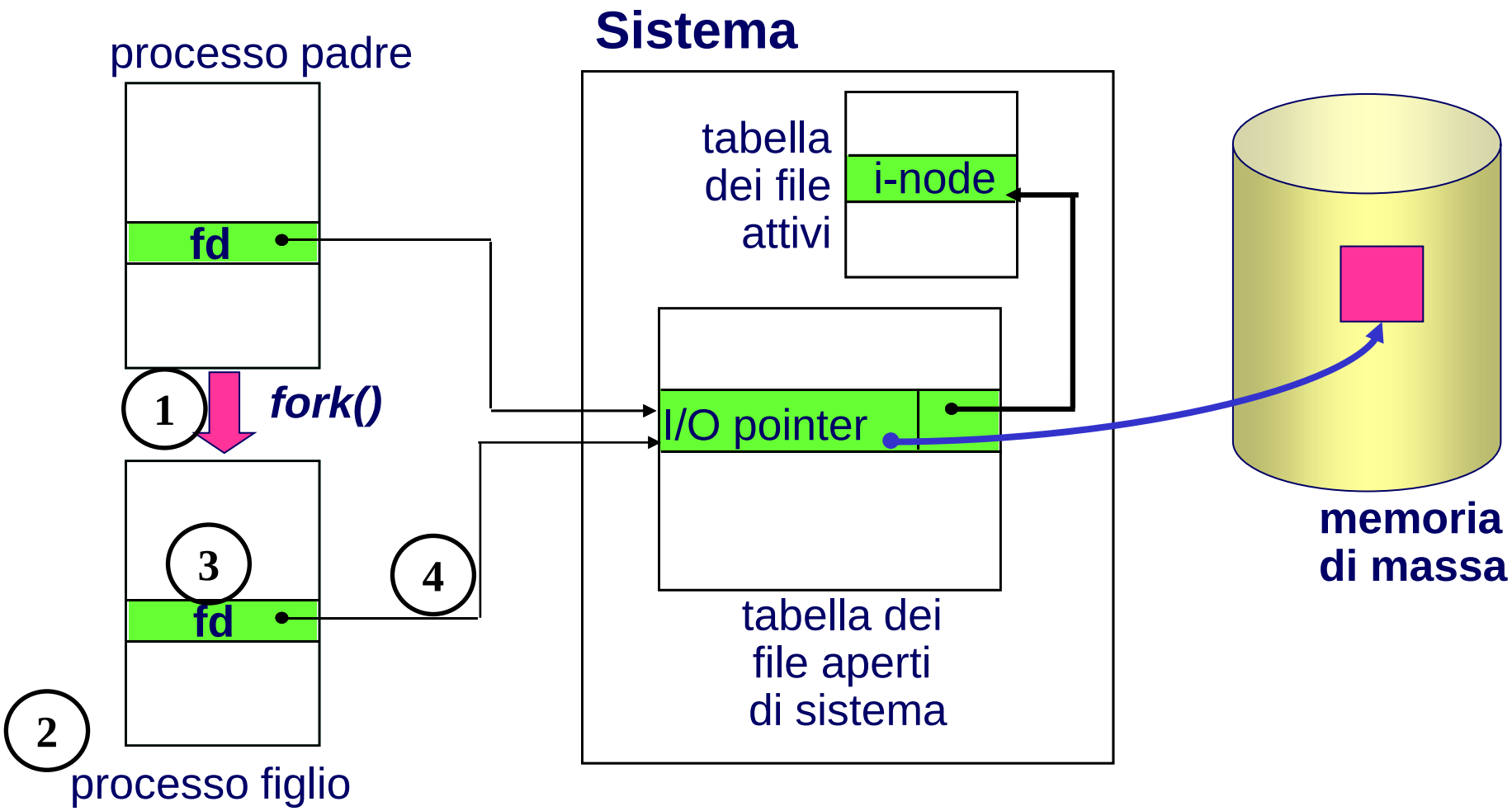
Esempio: processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti



Esempio: processi padre e figlio condividono I/O pointer di file aperti prima della creazione



Esempio: processi padre e figlio condividono I/O pointer di file aperti prima della creazione



Gestione file system UNIX: *system call*

UNIX permette ai processi di accedere a file, mediante un insieme di **system call**, tra le quali:

- **apertura/creazione: `open()`, `creat()`**
- **chiusura: `close()`**
- **lettura: `read()`**
- **scrittura: `write()`**
- cancellazione: **`unlink()`**
- linking: **`link()`**
- accesso diretto: l**`seek()`**

System Call: apertura di file

L'apertura di un file provoca:

- **inserimento di un elemento** (individuato da un file descriptor) nella prima posizione libera della **tabella dei file aperti del processo**
- **inserimento di un nuovo record nella tabella dei file aperti di sistema**
- la **copia dell'i-node nella tabella dei file attivi** (solo se il file non è già in uso)

Apertura di file: `open()`

Per aprire un file:

```
int open(char nomefile[], int flag, [int mode]);
```

- **`nomefile`** è il nome del file (relativo o assoluto)
- **`flag`** esprime il **modo di accesso**; ad esempio **`O_RDONLY`**, per accesso in lettura, **`O_WRONLY`**, per accesso in scrittura
- **`mode`** è un parametro richiesto soltanto se l'apertura determina **la creazione del file** (flag **`O_CREAT`**): in tal caso, **`mode`** specifica i **bit di protezione**

- **Valore restituito dalla `open()` è il file descriptor associato al file,** -1 in caso di errore
  - se **`open()`** ha successo, il file viene aperto nel modo richiesto e **I/O pointer posizionato sul primo elemento** (tranne nel caso di O_APPEND)

Apertura di file: `open()`

Modi di apertura (definiti in `<fcntl.h>`)

- **`O_RDONLY` (=0), accesso in lettura**
- **`O_WRONLY` (=1), accesso in scrittura**
- **`O_APPEND` (=2), accesso in scrittura in modalità append (in fondo al file)**, sempre da associare a O_WRONLY

Inoltre, è possibile **abbinare** ai tre modi precedenti, **altri modi (mediante il connettore |):**

- **`O_CREAT`**, per accesso in scrittura. Se il file non esiste, viene creato, è necessario fornire il parametro **`mode`, per esprimere i bit di protezione**
- **`O_TRUNC`**, per accesso in scrittura: la lunghezza del file viene troncata a 0

Apertura di file: `creat()`

Per creare un file:

```
int creat(char nomefile[], int mode);
```

- **`nomefile`** è il nome del file (relativo o assoluto) da creare
- **`mode`** è necessario e specifica i **12 bit di protezione** per il nuovo file

- **Valore restituito da `creat()` è il file descriptor** associato al file, -1 in caso di errore
- se **`creat()`** ha successo, il file viene aperto in scrittura e **l/O pointer posizionato sul primo elemento** (se file esistente, viene cancellato e riscritto da capo)

Apertura di file: open(), creat()

```
#include <fcntl.h>
...
main(){
  int fd1, fd2, fd3;
  fd1=open("/home/carlo/ff.txt", O_RDONLY);
  if (fd1<0) perror("open fallita");
  ...
  fd2=open("f2.new",O_WRONLY);
  if (fd2<0) {
      perror("open in scrittura fallita:");
      fd2=open("f2.new",O_WRONLY|O_CREAT, 0777);
      // è equivalente a: fd2=creat("f2.new", 0777);
  }
  /*omogeneità apertura dispositivo di output:*/
  fd3=open("/dev/prn", O_WRONLY);
  ...
}
```

Chiusura di file: `close()`

Per chiudere un file aperto:

```
int close(int fd);
```

`fd` è il file descriptor del file da chiudere

Restituisce l'esito della operazione (0, in caso di successo, <0 in caso di insuccesso)

Se **`close()`** ha successo:

- file memorizzato sul disco
- eliminato l'elemento di indice ***`fd` dalla tabella dei file aperti di processo***
- ***eventualmente eliminati*** gli elementi dalla tabella dei file aperti di sistema e dei file attivi

System call: lettura e scrittura

Caratteristiche:

- accesso mediante **file descriptor**
- ogni **operazione di lettura (o scrittura) agisce sequenzialmente** sul file, **a partire dalla posizione corrente di I/O pointer**
- possibilità di alternare operazioni di lettura e scrittura
- **atomicità** della singola operazione
- **operazioni sincrone,** cioè con attesa del completamento logico dell'operazione

Lettura di file: `read()`

```
int read(int fd,char *buf,int n);
```

- **`fd`** file descriptor del file
- **`buf`** area in cui trasferire i byte letti
- **`n`** numero di caratteri da leggere

- **In caso di successo, restituisce un intero positivo (<=n) che rappresenta il numero di caratteri effettivamente letti**

Per indicare da tastiera (in fase di input) la volontà di terminare il file, <**CTRL-D**>

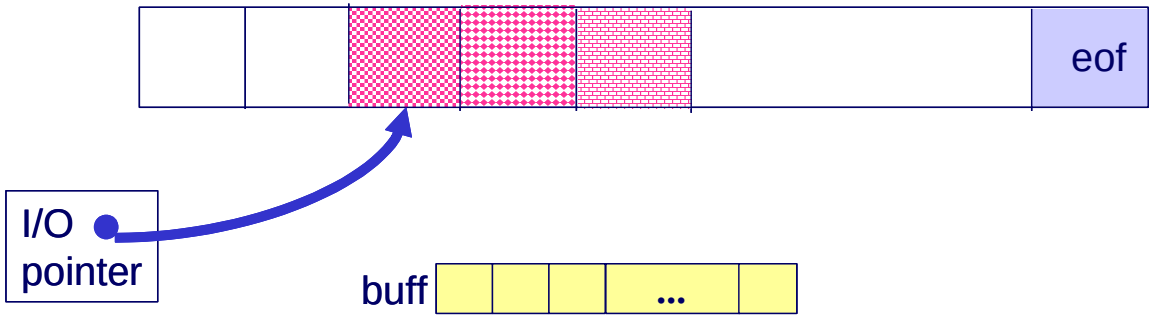
Lettura di file: read()

Se **`read(fd, buff, n)`** ha successo:

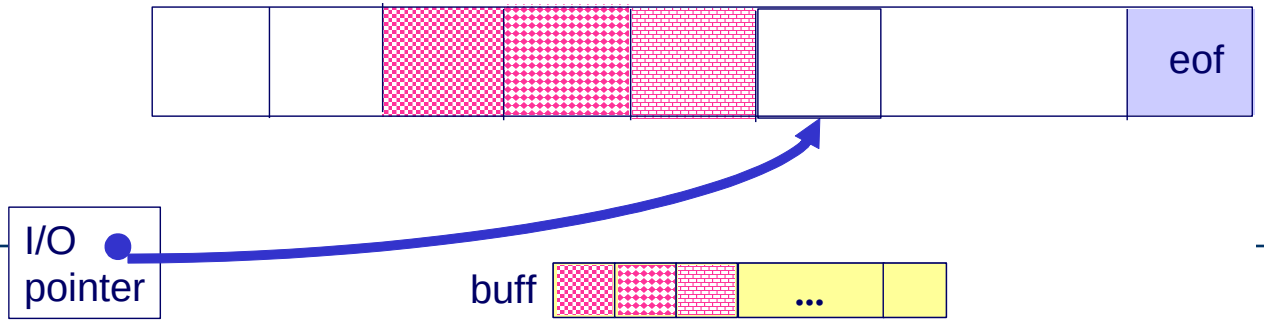
- a partire dal valore corrente di I/O pointer, vengono letti al più **`n`** bytes dal file **`fd`** e memorizzati all'indirizzo **`buff`**
- **I/O pointer** viene spostato **avanti di `n` bytes**

Ad esempio: prima di

`read(fd, buff, 3)`



dopo la `read()`:



Scrittura di file: `write()`

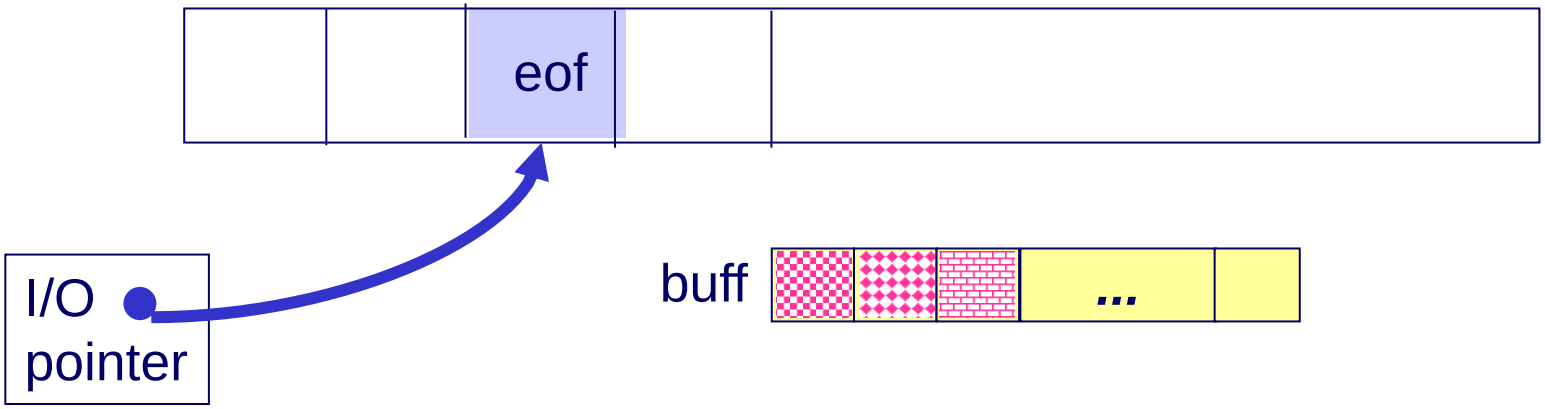
```
int write(int fd,char *buf,int n);
```

- **`fd`** file descriptor del file
- **`buf`** area da cui trasferire i byte scritti
- **`n`** numero di caratteri da scrivere

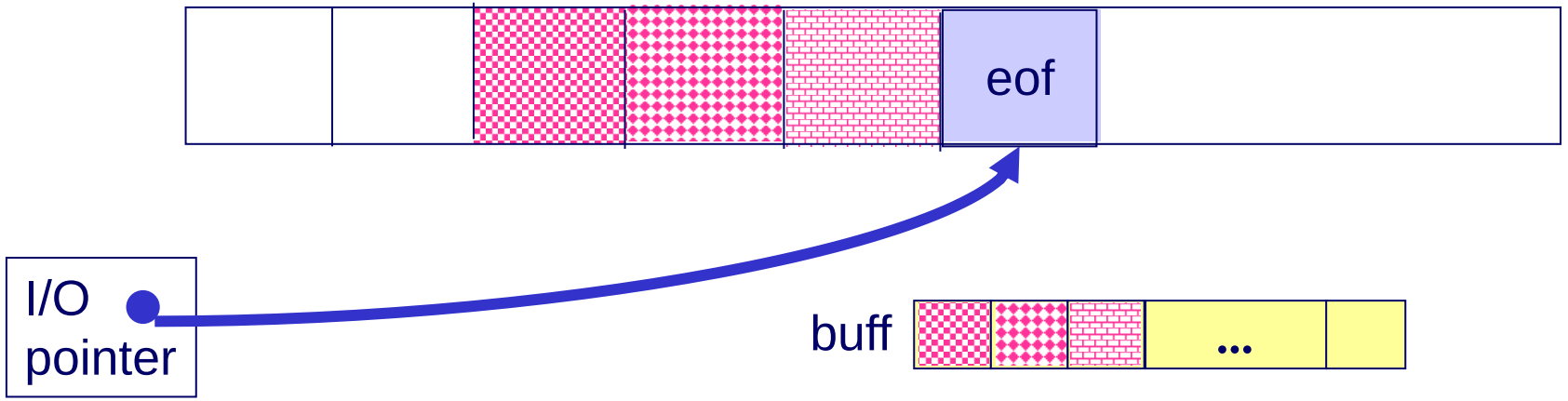
In caso di **successo**, restituisce un intero positivo (==n) che rappresenta il **numero di caratteri effettivamente scritti**

Scrittura di file: write()

Ad esempio: prima di `write(fd, buff, 3)`



dopo la `write()`



Esempio: **read()** & **write()**

Visualizzazione sullo standard output del contenuto di un file

```
#include <fcntl.h>
main(){
  int fd,n;
  char buf[10];
  if((fd=open("/home/miofile",O_RDONLY))<0){
      perror("errore di apertura:");
      exit(-1);
  }
  while ((n=read(fd, buf, 10))>0)
     write(1,buf,n); // scrittura sullo standard output
  close(fd);
}
```

Esempio:
comando cp(copia argv[2] in argv[1])

```
#include <fcntl.h>  …        #include <stdio.h>
#define BUFDIM 1000 …        #define perm 0777

main (int argc, char **argv){
  int status, infile, outfile, nread;
  char buffer[BUFDIM];
  if (argc != 3) { printf (" errore \n"); exit (1); }
  if ((infile=open(argv[2], O_RDONLY)) <0){
     perror("apertura sorgente: "); exit(1); }
  if ((outfile=creat(argv[1], perm )) <0){
     perror("apertura destinazione:");
     close (infile);
     exit(1);
  }
```

Esempio:

comando `cp` (copia argv[2] in argv[1])

```
    while((nread=read(infile, buffer, BUFDIM)) >0 ){
       if(write(outfile, buffer, nread)< nread){
           close(infile);
           close(outfile);
           exit(1);
       }
    }
    close(infile);
    close(outfile);
    exit(0);
} // fine main
```

Accesso diretto: `lseek()`

Per spostare I/O pointer:

```
lseek(int fd, int offset, int origine);
```

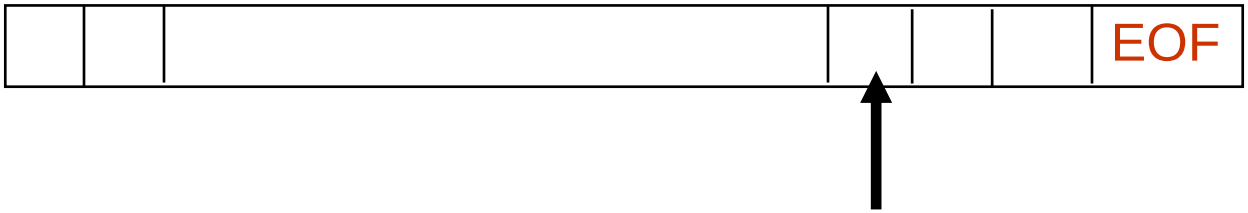
- **`fd`** file descriptor del file
- **`offset`** spostamento (in byte) rispetto all'origine
- **`origine`** può valere:
  - ✓ 0: inizio file **(`SEEK_SET`)**
  - ✓ 1: posizione corrente **(`SEEK_CUR`)**
  - ✓ 2: fine file **(`SEEK_END`)**

In caso di successo, restituisce un intero positivo che rappresenta la **nuova posizione**

Esempio: lseek()

```
#include <fcntl.h>
main(){
    int fd,n; char buf[100];
    if(fd=open("/home/miofile",O_RDWR)<0)
     ...;
     lseek(fd,-3,2); /* posizionamento sul
                        terzultimo byte del
      file */

...
}
```



Cancellazione di file: `unlink()`

Per **cancellare un file, o decrementare il numero dei suoi link**:

```
int unlink(char *name);
```

- **name** nome del file

ritorna 0 se OK, altrimenti -1

In generale, l'effetto della system call **unlink()** è **decrementare di 1 il numero di link del file dato (nell'i-node);** solo nel caso in cui il numero dei **link risulti 0**, allora il file viene effettivamente **cancellato**

Aggiungere nomi a file esistenti: `link()`

Per aggiungere un link a un file esistente:

```
int  link(char *oldname, char * newname);
```

- **`oldname`** nome del file esistente
- **`newname`** nome associato al nuovo link

`link()`

- **incrementa il numero dei link** associato al file nell'**i-node**
- aggiorna il direttorio (aggiunta di un nuovo elemento)
- ritorna 0 in caso di successo; -1 se fallisce

ad esempio fallisce se:

- **`oldname`** non esiste
- **`newname`** esiste già (non viene sovrascritto)
- **`oldname`** e **`newname`** appartengono a unità disco diverse (in questo caso, usare link simbolici mediante **`symlink`**)

Esempio

Realizzazione del comando bash `mv`

```
main (int argc,char ** argv){
   if (argc != 3){
      printf ("Sintassi errata\n"); exit(1); }

   if (link(argv[1], argv[2]) < 0){
      perror ("Errore link"); exit(1);}

   if (unlink(argv[1]) < 0){
      perror("Errore unlink"); exit(1);}

   exit(0);
} // fine main
```

Diritti di accesso a file: `chmod()`

Per modificare i bit di protezione di un **file**:

```
int  chmod (char *pathname, char *newmode);
```

- **pathname** nome del file
- **newmode** contiene i nuovi diritti

Ad esempio:
```
chmod("file.txt", "0777")
chmod("file.txt", "o-w")
chmod("file.txt", "g+x")
```

Diritti di accesso a file: `chown()`

Per cambiare il proprietario e il gruppo di un **file**:

```
int  chown (char *pathname, int owner, int group);
```

- **`pathname`** nome del file
- **`owner`** uid del nuovo proprietario
- **`group`** gid del gruppo

Cambia proprietario/gruppo del file

Gestione directory

Alcune system call a disposizione in C/UNIX per la gestione delle directory

- **`chdir()`**: per cambiare directory (come comando shell cd)
- **`opendir()`**, **`closedir()`**: apertura e chiusura di directory
- **`readdir()`**: lettura di directory

Le system call sopra fanno uso di tipi astratti per la descrizione della struttura dati directory e sono **indipendenti da come il direttorio viene realizzato** (BSD, System V, Linux)

Gestione di directory

Per effettuare un cambio di directory

```
int chdir (char *nomedir);
```

- **`nomedir`** nome della directory in cui entrare

restituisce 0 in caso di successo (cioè cambio di directory avvenuto), -1 in caso di fallimento

- Analogamente ai file normali, **lettura/scrittura di una directory** possono avvenire solo dopo l'operazione di apertura **`opendir()`**

- Apertura restituisce un puntatore a **`DIR`**:
  - **`DIR`** è un **tipo di dato astratto predefinito (`<dirent.h>`)** che consente di riferire (mediante puntatore) una directory aperta

Apertura e chiusura di directory: `opendir()`, `closedir()`

Per aprire un direttorio:

```
#include <dirent.h>
DIR  *opendir (char *nomedir);
```

`nomedir` nome del direttorio da aprire

restituisce un valore di tipo puntatore a **`DIR`**:

- diverso da NULL se l'apertura ha successo: per gli accessi successivi, si impiegherà **questo valore per riferire il direttorio**
- altrimenti restituisce NULL

Chiusura del direttorio riferito dal puntatore **`dir`**:

```
#include <dirent.h>
int closedir (DIR *dir);
```

Restituisce 0 in caso di successo, -1 altrimenti

Lettura di una directory

Una directory aperta può essere letta con **`readdir()`**:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *descr;
descr = readdir(DIR *dir);
```

`dir` puntatore alla directory da leggere (valore restituito da **`opendir`**)

Restituisce:

- un puntatore diverso da **NULL** se la lettura ha avuto **successo**
- altrimenti **restituisce NULL** (in caso di **insuccesso**)

- In caso di successo, la **`readdir()`** legge un elemento dalla directory data e lo memorizza **all'indirizzo puntato da `descr`**
- **`descr`** punta ad una **struttura di tipo `dirent`** (dichiarata in **`dirent.h`**)

L'elemento nella directory: `dirent`

Il generico elemento (file o directory) in una directory è rappresentato da un record di tipo **`dirent`**:

```
struct dirent {
    long d_ino; /* i-number */
    off_t d_off; /* offset del prossimo */
    unsigned short d_reclen; /* lunghezza del record */
    unsigned short d_namelen; /* lunghezza del nome */
    char d_name[1]; /* nome del file */
}
```

La stringa che parte da **`d_name`** rappresenta il nome del file (o directory) nella directory aperta; **`d_namelen`** la lunghezza del nome

→ possibilità di nomi con lunghezza variabile

Gestione di directory: creazione

Creazione di una directory

```
int  mkdir (char *pathname, int mode);
```

- **pathname** nome del direttorio da creare
- **mode** esprime i bit di protezione

Restituisce il valore 0 in caso di successo, altrimenti -1

In caso di **successo**, crea e inizializza una directory con il nome e i diritti specificati. **Vengono sempre creati i file:**

- **.** **(link alla directory corrente)**
- **..** **(link alla directory padre)**

Esempio: realizzazione del comando ls

```
#include <stdlib.h>      …      #include <sys/types.h>
#include <dirent.h>      …      #include <fcntl.h>

void  miols(char name[]){
   DIR *dir; struct dirent * dd;
   char buff[80];
   dir = opendir(name);
   while ((dd = readdir(dir)) != NULL){
      sprintf(buff, "%s\n", dd->d_name);
      write(1, buff, strlen(buff));
   }
   closedir(dir);
   return;
} // fine miols
```

Esempio: realizzazione di ls (continua)

```
main (int argc, char **argv){
    if (argc <= 1){
        printf("Errore\n");
        exit(1);
    }
    miols(argv[1]);
    exit(0);
} // fine main
```

Esempio: esplorazione di una gerarchia

Si vuole operare **in modo ricorsivo su una gerarchia di directory alla ricerca di un file con nome specificato**. Per esplorare la gerarchia si usino le system call **`chdir`**, **`opendir`**, **`readdir`** e **`closedir`**

Si preveda una sintassi di invocazione del programma eseguibile:

```
ricerca radice file
```

- **`radice`** nome directory “radice” della gerarchia
- **`file`** nome del file da ricercare (ad esempio, dato in modo assoluto)

Esempio: esplorazione di una gerarchia

```
#include <stdlib.h>      …      #include <stdio.h>
#include <sys/types.h>   …      #include <dirent.h>

void esplora (char *d, char *n);

main (int argc, char **argv){
   if (argc != 3){printf(”Errore par.\n"); exit (1);}
   if (chdir (argv[1])!=0){
       perror("Errore in chdir");
       exit(1);
   }
   esplora (argv[1], argv[2]);
}
```

```
void esplora (char *d, char *f){
  char nd [80]; DIR *dir; struct dirent *ff;
  dir = opendir(d);
  while ((ff = readdir(dir)) != NULL){
    if ((strcmp (ff->d_name, ".") != 0) &&
    (strcmp (ff->d_name, "..") !=0)) /* salto . e .. */
       if (chdir(ff->d_name) != 0){ /* è un file */
          if (strcmp (f, ff->d_name) == 0)
             printf("file %s nel dir %s\n", f, d);
       } else{ /*abbiamo trovato un direttorio */
          strcpy(nd, d); strcat(nd, "/");
          strcat(nd, ff-> d_name);
          esplora(nd, f);
          chdir("..");
       } // fine else
  } // fine while
  closedir(dir);
} // fine esplora
```

perché è necessario salire di un livello al ritorno dalla chiamata ricorsiva?