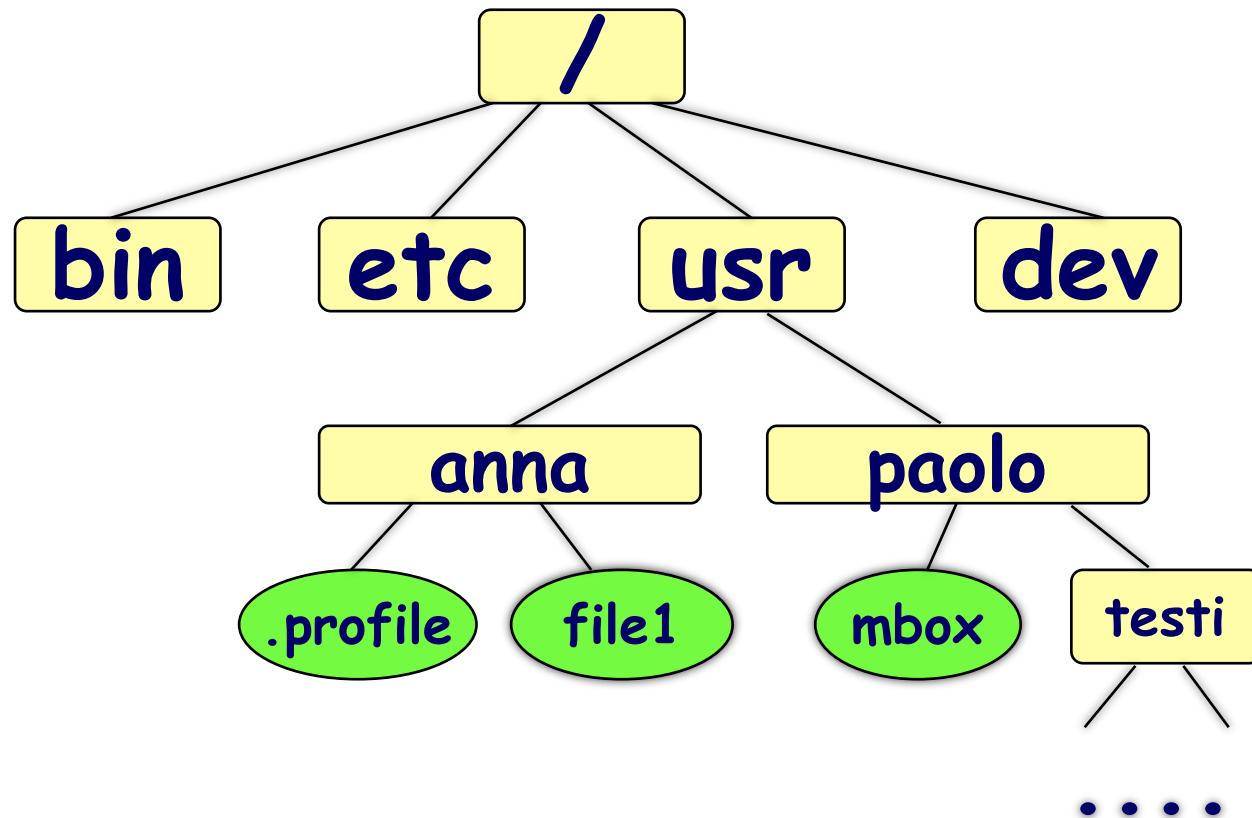


# Il File System di Unix

# Il File System di UNIX

## Organizzazione logica



# Il File System di UNIX

- 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*)

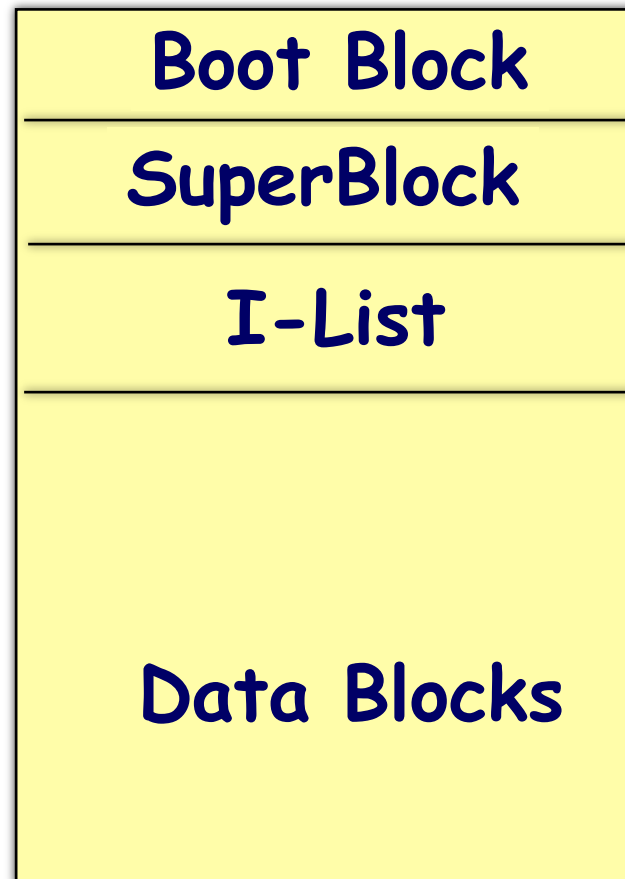
# Il File System

## Organizzazione Fisica

- Il metodo di allocazione utilizzato in Unix è ad **indice** (*a più livelli di indirizzamento*)
- formattazione del disco in **blocchi** fisici (**dimensione del blocco**: 512- 4096 Bytes).
- La superficie del disco File System è partizionata in **4 regioni**:
  - boot block
  - super block
  - i-list
  - data blocks

# **Il File System**

## **Organizzazione Fisica**



# Il File System

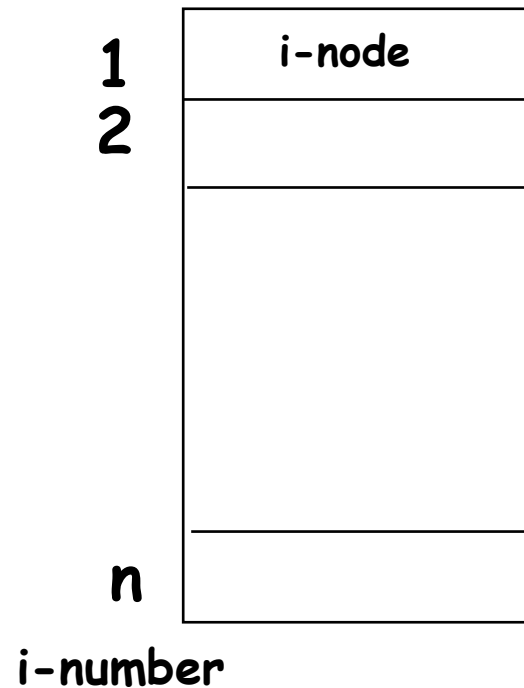
## Organizzazione Fisica

- **Boot Block:** contiene le procedure di inizializzazione del sistema (da eseguire al *bootstrap*)
- **SuperBlock:** 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:** è l'area del disco effettivamente disponibile per la memorizzazione dei file; contiene:
  - i blocchi allocati
  - i blocchi liberi (organizzati in una lista collegata)

# Il File System

## Organizzazione Fisica

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





# 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 links**
  - ▣ **13 -15 indirizzi di blocchi** ( a seconda della realizzazione)

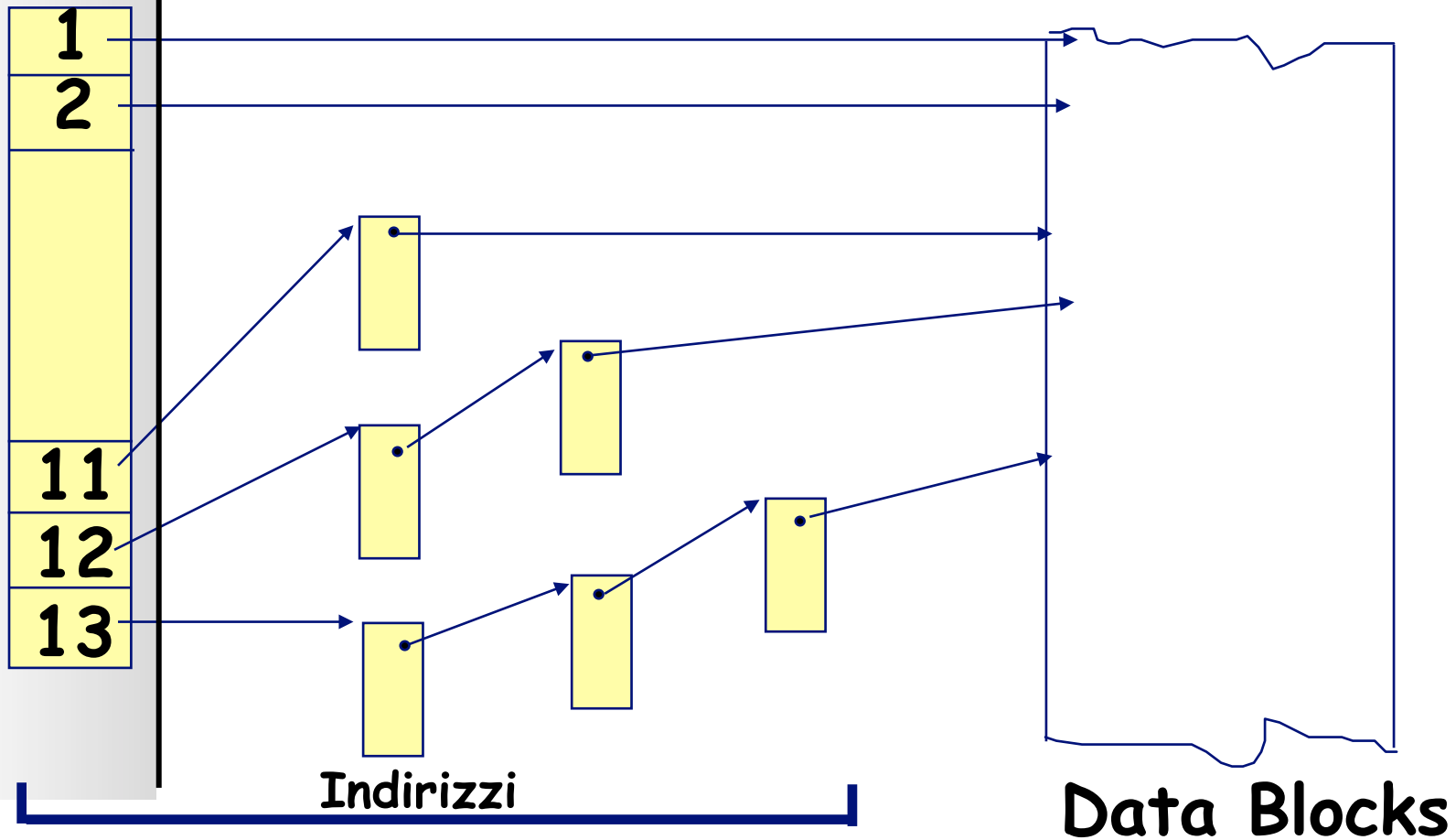
# Indirizzamento

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 (1 livello di *indirettezza*)
- 12° indirizzo:            due livelli di *indirettezza*
- 13° indirizzo:            tre livelli di *indirettezza*

# Indirizzamento

*i-node*



# Indirizzamento

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

## Quindi:

- 10 blocchi di dati sono accessibili **direttamente**
  - file di dimensioni minori di  $(10 \times 512) \text{ byte} = 5120 \text{ byte} = 5 \text{ KB}$  sono accessibili direttamente
- 128 blocchi di dati sono accessibili con *indirizzazione singola* (mediante il puntatore 11):  **$128 \times 512 \text{ byte} = 65536 \text{ byte} = 64 \text{ KB}$**
- $128 \times 128$  blocchi di dati sono accessibili con *indirizzazione doppia* (mediante il puntatore 12):  **$128 \times 128 \times 512 \text{ byte} = 8 \text{ MB}$**
- $128 \times 128 \times 128$  blocchi di dati sono accessibili con *indirizzazione tripla* (mediante il puntatore 13):  **$128 \times 128 \times 128 \times 512 \text{ byte} = 1 \text{ GB}$**

# Indirizzamento

- la dimensione massima del file è dell'ordine del **Gigabyte** :

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

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

# Protezione: controllo dell'accesso ai file

- esistono tre modalità di accesso ai file:  
*lettura, scrittura, esecuzione*
- il proprietario può *concedere* o *negare* agli altri utenti il permesso di accedere ai propri file
- esiste un utente *privilegiato* (**root**) che ha accesso incondizionato ad ogni file del sistema

## bit di protezione

- Ad ogni file sono associati **12 bit** di protezione (nell'i-node):

suid	sgid	sticky	r w x	r w x	r w x
------	------	--------	-------	-------	-------

U

G

O

# Bit di Protezione:

## lettura, scrittura, esecuzione

suid	sgid	sticky	r w x	r w x	r w x
------	------	--------	-------	-------	-------

U

G

O



9 bit di lettura (read), scrittura (write),  
esecuzione(execute) per:

- ▣ utente proprietario (User)
- ▣ utenti del gruppo (Group)
- ▣ tutti gli altri utenti (Others)



# bit di protezione: lettura, scrittura, esecuzione

Ad esempio, il file:

**pip**po

U	G	O
1 1 1	0 0 1	0 0 0
r w x	- - x	- - -

- è leggibile, scrivibile, eseguibile per il proprietario
- è solo eseguibile per gli utenti dello stesso gruppo
- nessun tipo di modalità per gli altri
- formato ottale: 111 => 7; 010 => 2; ... -rwx--x---  
=> 0710

# bit di protezione per file eseguibili

<b>suid</b>	<b>sgid</b>	<b>sticky</b>	r w x	r w x	r w x
-------------	-------------	---------------	-------	-------	-------

3 bit di permessi per file **eseguibili**:

- ❑ Set-User-ID (SUID)
- ❑ Set-Group-ID (SGID)
- ❑ Save-Text-Image (Sticky)

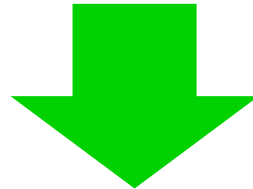
## SUID, SGID, Sticky

### File eseguibili:

- al processo che esegue un file eseguibile è associato dinamicamente uno User-ID (e Group-ID)
  - ➔ User-ID effettivo
- Default: User-ID (e Group-ID) dell'utente che lancia il processo
  - ➔ User-ID reale

## SUID e SGID

- **Set-User-ID (SUID)**: associa al processo che esegue il file lo User-Id del proprietario del file
- **Set-Group-ID (SGID)**: associa al processo che esegue il file il Group-Id del proprietario del file



Chi lancia il processo assume  
temporaneamente l'**identità** del  
**proprietario**

## SUID, SGID , e file /etc/passwd

- in Unix tutte le informazioni relative alla amministrazione del sistema sono rappresentate da **file** (di **root**)

**Esempio:** utenti, gruppi e password sono registrati nel file **/etc/passwd:**

```
root:Mz5DJvSXy:0:1:Operator:/:/bin/csh
```

```
paola:eLQZs:290:30:Paola Rossi:/home/paola:/bin/csh
```

**SUID, SGID, e il file /etc/passwd**  
**/etc/passwd è accessibile in scrittura solo dal proprietario:**

```
$ ls -l /etc/passwd  
-rw-r--r--  1 root  wheel  2888 23  Set   2007 /etc/passwd
```



- necessità di concedere l'accesso in scrittura ad ogni utente **solo** per le modifiche relative al proprio username

attraverso il comando **/bin/passwd** (di **root**)

## SUID, SGID, e il file /etc/passwd

il comando `/bin/passwd` modifica il file `etc/passwd`:

<code>/bin/passwd</code>	1	1	111	101	101
	SUID	SGID	U	G	O

⇒ chiunque lo esegue può accedere e modificare (in modo controllato) il file `/etc/passwd`, “vestendo i panni” del superutente (root)

# Bit di Protezione:

## Sticky bit

- **Save-Text-Image (Sticky)**: l'immagine del processo viene mantenuta in **area di swap** anche dopo che ha finito il proprio compito (il processo è terminato)  
-> maggior velocità di (ri) avvio

**Esempio:**

i comandi utilizzati frequentemente



# modifica dei bit di protezione

- Shell: comando `chmod`
- System Call: `int chmod()`

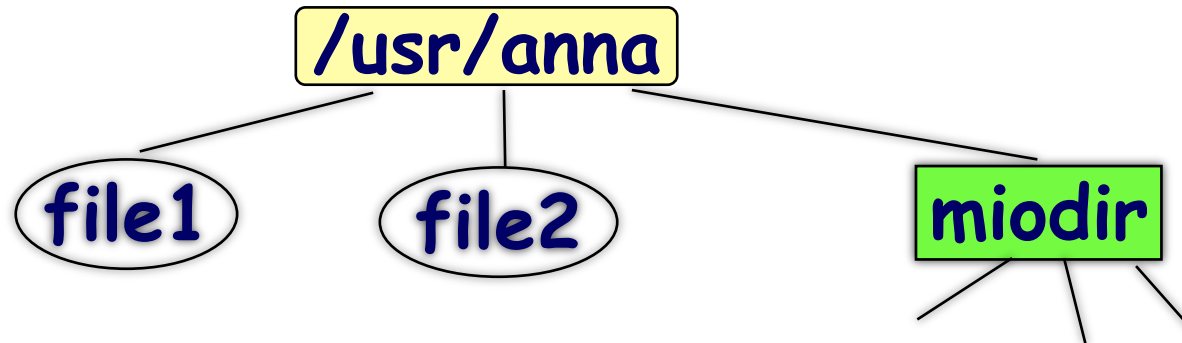
# Il Direttorio

- Anche il direttorio è rappresentato nel *file system* da un file.
- Ogni file-direttorio contiene un insieme di record logici con la seguente struttura:

nomerelativo	i-number
--------------	----------

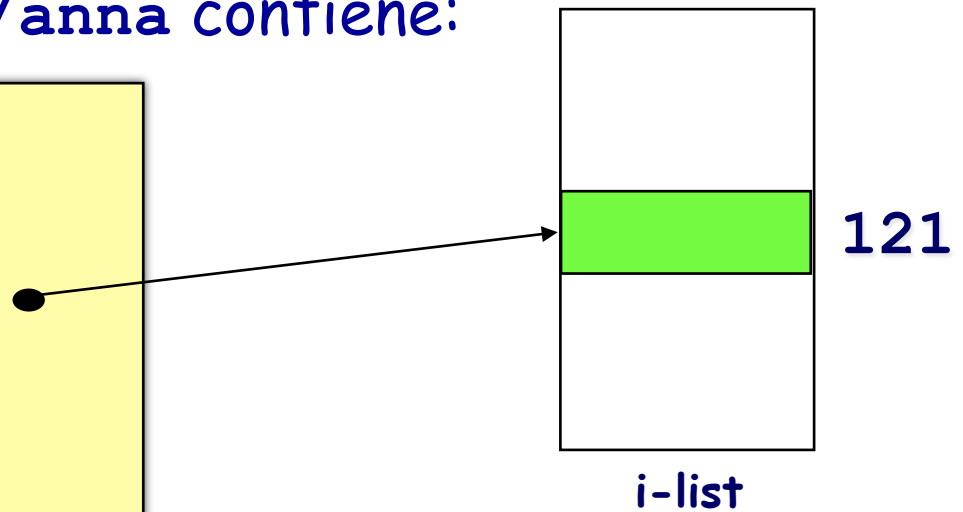
- ogni record rappresenta un file appartenente al direttorio:
  - ➔ per ogni file (o direttorio) appartenente al direttorio considerato, viene memorizzato il suo nome relativo, al quale viene associato il valore del suo *i-number* (che lo identifica univocamente)

# Il Direttorio



il file (direttorio) `/usr/anna` contiene:

<code>file1</code>	189
<code>file2</code>	133
<code>miodir</code>	121
<code>.</code>	110
<code>..</code>	89



# File system Linux

- Prime versioni: implementazione del file system di Minix (unix-like): limitazioni su:
    - indirizzabilità (64 MB per file e per filesystem!)
    - nomi dei file (16 caratteri)
    - gestione dei blocchi inefficiente (bitmap dei blocchi liberi e di inode)
  - **Ext FS (1992): indirizzabilità 1GB/4GB, gestione blocchi liberi con liste.**
  - **Ext 2 (1993): 16GB/4TB**
    - **Flessibilità:** l'amministratore può decidere
      - dimensione del blocco (1024-4096 bytes)
      - dimensione i-list
    - **Efficienza:**
      - **Gruppi di blocchi:** ogni gruppo include data blocks e inode memorizzati in tracce adiacenti e una **copia** delle strutture di controllo (superblock e descrittore filesystem) -> affidabilità
      - **fast symbolic links:** il link è memorizzato direttamente nell'i-node
      - **Preallocazione:** il filesystem prealloca blocchi a file prima che vengano effettivamente scritti
  - **Ext 3 (2001), Ext4 : estensioni journaled di ext2 -> tolleranza ai guasti.**
-

## EXT2 Block groups:

- il file system e' organizzato fisicamente in gruppi di blocchi :

boot block	block group1	block group2	...	block groupN
------------	--------------	--------------	-----	--------------

- La struttura di ogni gruppo e' la seguente:

Superblock	FS descriptors	...	group inode table	group data blocks
------------	----------------	-----	-------------------	-------------------

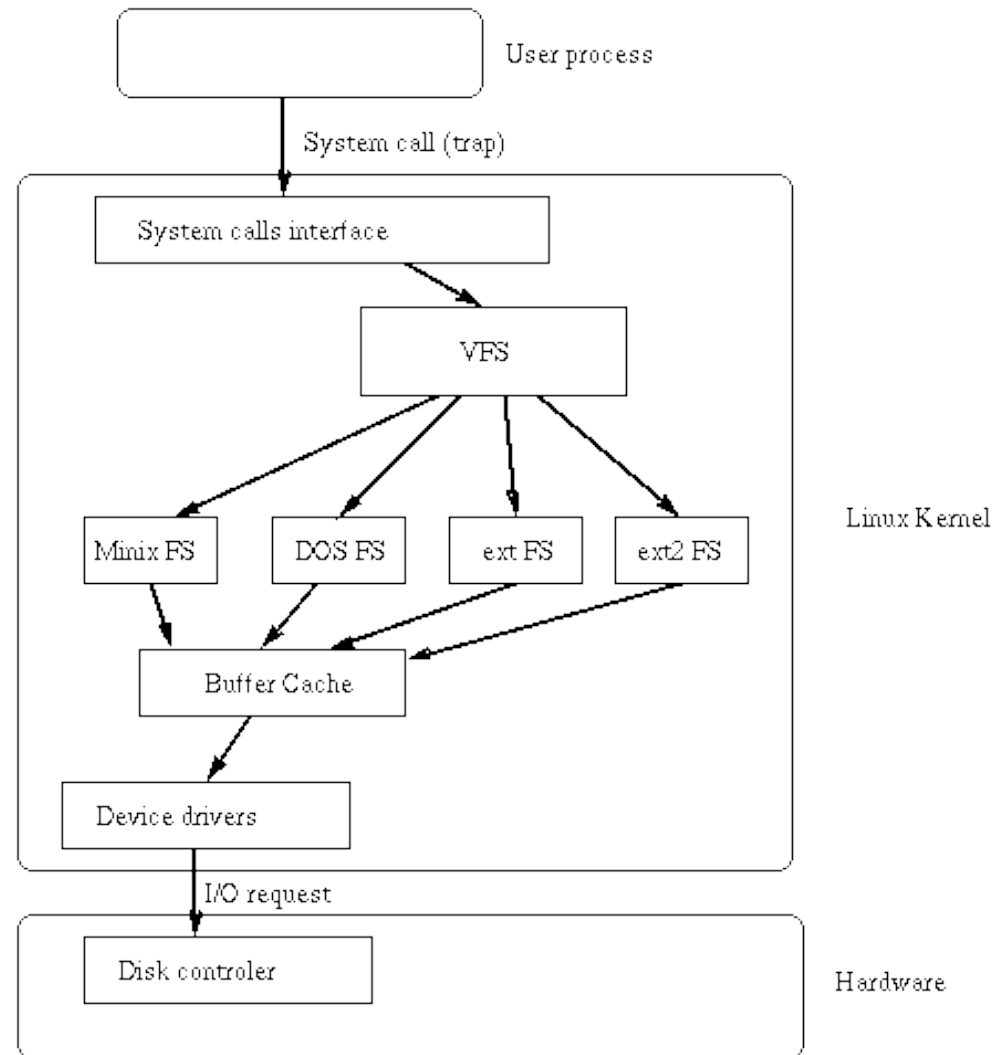
- ➔ localita' di inode e relativi file: tempi di accesso ridotti
- ➔ localita' dei blocchi allocati a uno stesso file
- ➔ Replicazione del superblock-> tolleranza ai guasti

# Virtual File System

Linux prevede l'integrazione con filesystem diversi da Ext2, grazie al Virtual File System (VFS):

- intercetta ogni system call relativa all'accesso al file system e, all'occorrenza, provvede al collegamento con file system "esterni":
- file e filesystems sono mappati su *internal objects* nel kernel, ognuno dei quali include informazioni specifiche sulla risorsa che rappresenta:
  - ▣ superblock objects
  - ▣ i-node objects
  - ▣ file objects

# VFS



# VFS: file system supportati

- Disk File system:
  - ▣ unix-like FS: SystemV, BSD, ecc.
  - ▣ Microsoft-like: DOS, VFAT (Win98), NTFS..
  - ▣ HFS (Apple)
  - ▣ JFS (IBM)
  - ▣ ...
- Network file systems:
  - ▣ NFS
  - ▣ SMB (Microsoft)
  - ▣ NCP (Novell Net Ware Core Protocol)
  - ▣ ...



# Accesso al file system

# Accesso a File in Unix

Quali sono i meccanismi di accesso al file system?

Come vengono realizzati ?

**Vanno considerati:**

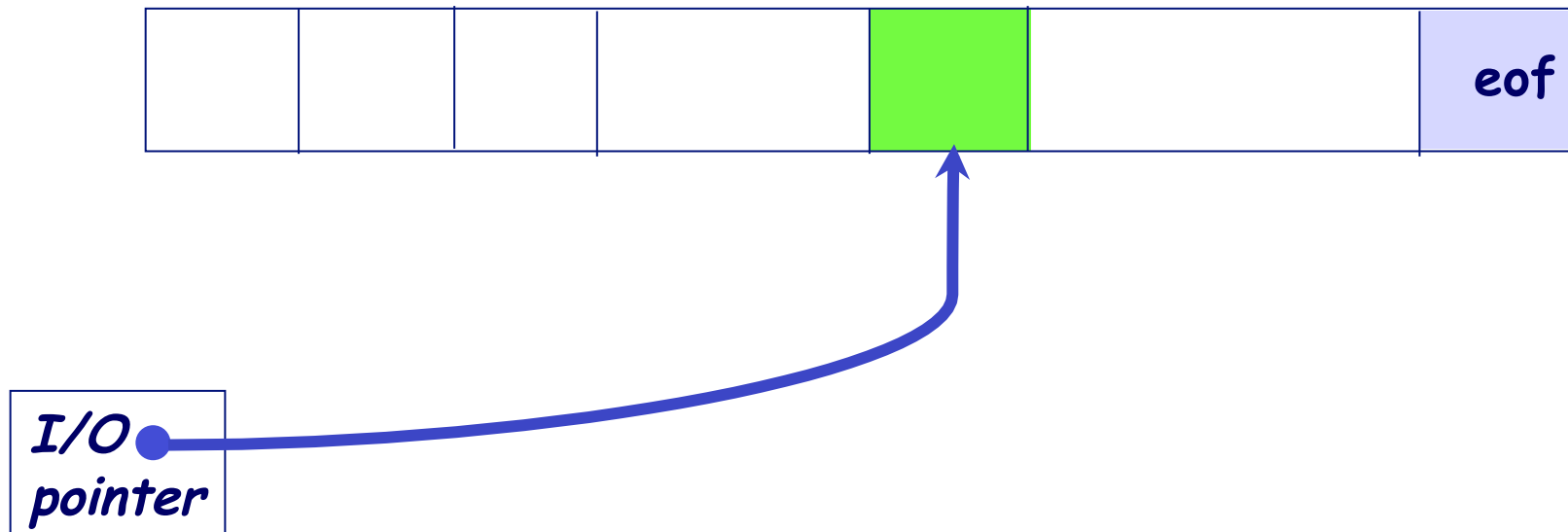
- strutture dati di sistema per il supporto all'accesso e alla gestione di file
- principali **system calls** per l'accesso e la gestione di file

# Accesso a File

## Concetti Generali

- accesso sequenziale
- il puntatore al file (**I/O Pointer**) registra la posizione corrente
- assenza di strutturazione:

**file = sequenza di bytes (stream)**



# Accesso a File

## Concetti Generali

- vari **modi** di accesso (*lettura, scrittura, lettura/scrittura, etc.*)
- accesso subordinato all'operazione di **apertura**:

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

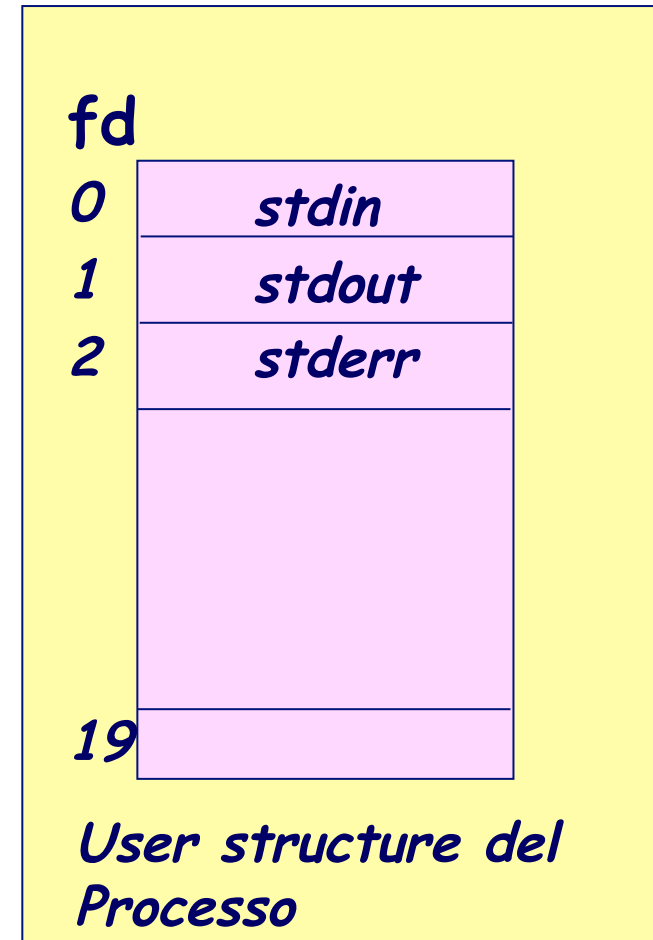
# Accesso a File

## File Descriptor

- A ogni processo è associata una **tabella dei file aperti** di dimensione limitata (tipicamente, 20 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**



# Accesso a File

## Strutture dati del Kernel

Per realizzare l'accesso ai file, il sistema operativo 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:
  - in questo modo si rendono più efficienti le operazioni di accesso 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:
  - » l'**I/O pointer**, che indica la 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  $F$ .

# Accesso a File

## Strutture dati

### Riassumendo:

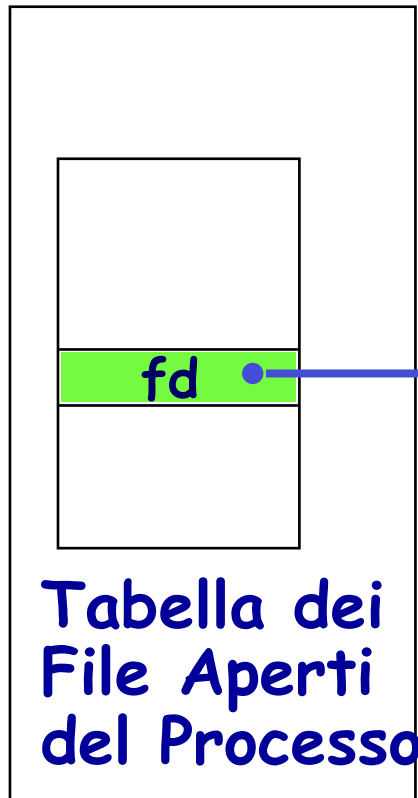
- **tabella dei file aperti di processo:** nella user area del processo, contiene un elemento per ogni file aperto dal processo
- **tabella dei file aperti di sistema:** contiene un elemento per ogni sessione di accesso a file nel sistema
- **tabella dei file attivi:** contiene un elemento per ogni file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

Quali sono le relazioni tra queste strutture?

# Gestione di File

## Strutture Dati

User Area del  
Processo



Area Dati del kernel

Tabella  
dei File  
Attivi

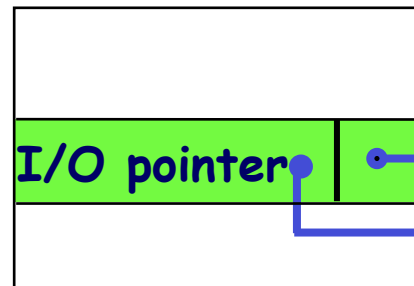
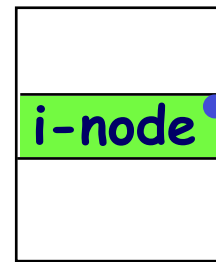
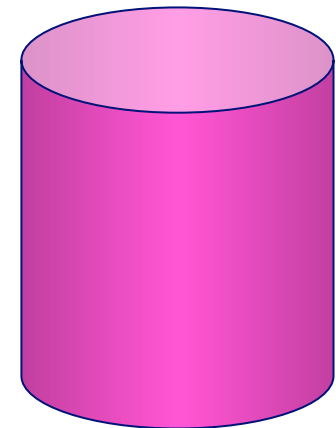


Tabella dei  
File Aperti  
di Sistema



Memoria  
di Massa



# Accesso a File

## 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)



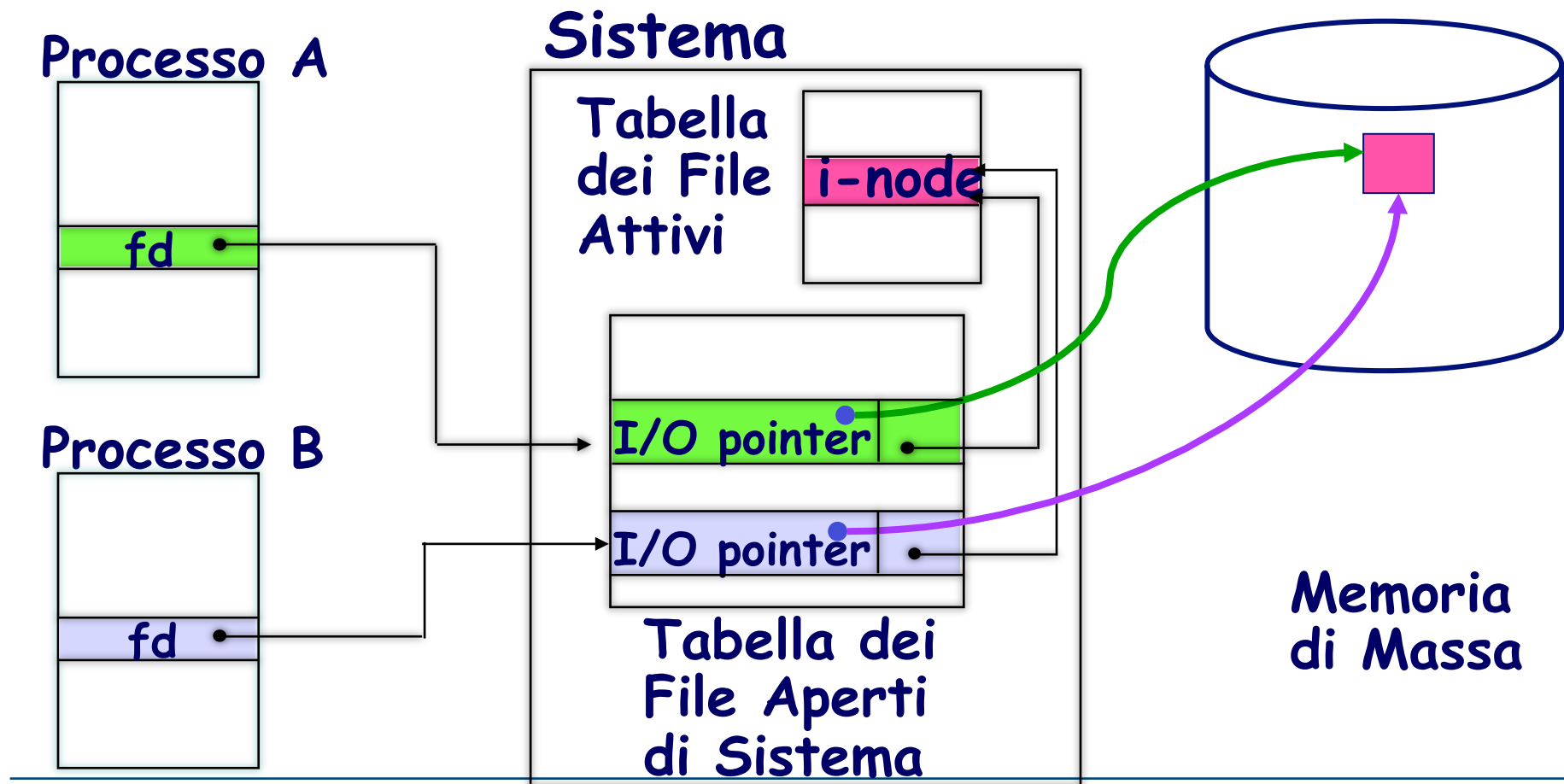
più processi possono accedere contemporaneamente allo stesso file,  
ma con **I/O pointer distinti** !

# Accesso a File

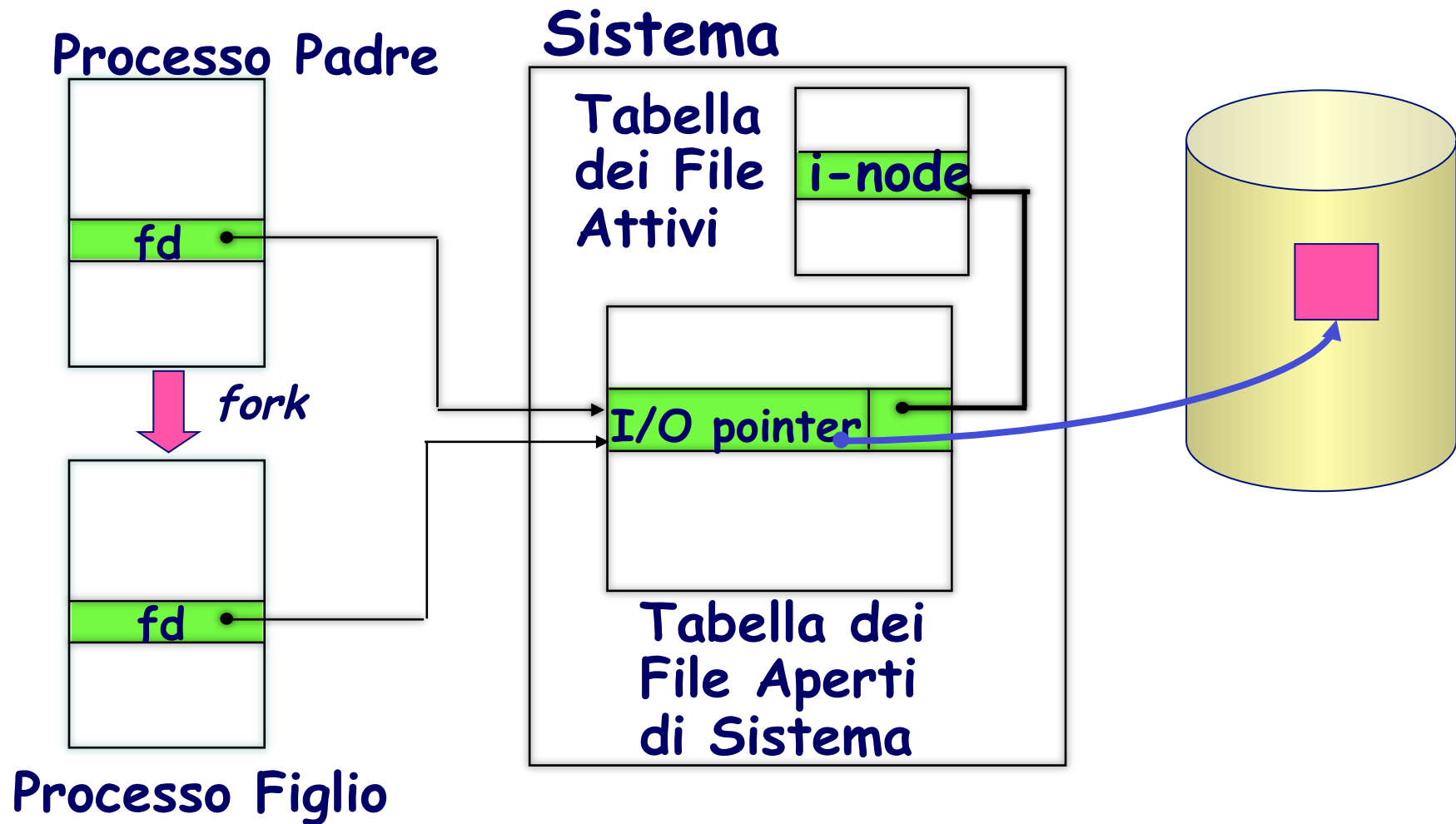
## Tabella dei File Attivi

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



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



# Accesso a file: *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`
  - ✓ accesso diretto: `lseek`

# System Calls

## Apertura di File

L'apertura di un file provoca:

- ❑ l'inserimento di un elemento (individuato da un file descriptor) nella prima posizione libera della Tabella dei file aperti del processo
- ❑ l'inserimento di un nuovo record nella Tabella dei file aperti di sistema
- ❑ la copia dell'*i-node* nella tabella dei file attivi (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 (ad esempio, codifica ottale).
- il valore restituito dalla `open` è il **file descriptor** associato al file, o -1 in caso di errore.
- Se la `open` ha successo, il file viene aperto nel modo richiesto, e l'**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, append
- Inoltre, è possibile **abbinare** ai tre modi precedenti, altri modi (mediante il connettore `|`); ad esempio:
  - `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` specifica i 12 bit di protezione per il nuovo file.
- il valore restituito dalla `creat` è il *file descriptor* associato al file, o -1 in caso di errore.
- Se la `creat` ha successo, il file viene aperto in scrittura, e l'I/O pointer posizionato sul primo elemento.

# Apertura di File: open, create

## Esempi:

```
#include <fcntl.h>

...

main()
{ int fd1, fd2, fd3;
  fd1=open("/home/anna/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); */
  }
  /*OMOGENEITA` : 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 la `close` ha successo:
    - il file viene memorizzato sul disco
    - viene eliminato l'elemento di indice `fd` dalla Tab. dei file aperti del processo.
    - Vengono *eventualmente* eliminati (se non condivisi con altri processi) gli elementi corrispondenti dalla Tab. dei file aperti di sistema e dalla tabella dei file attivi

# System Call: Lettura e Scrittura di File

## Caratteristiche:

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

## Lettura di File: read

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

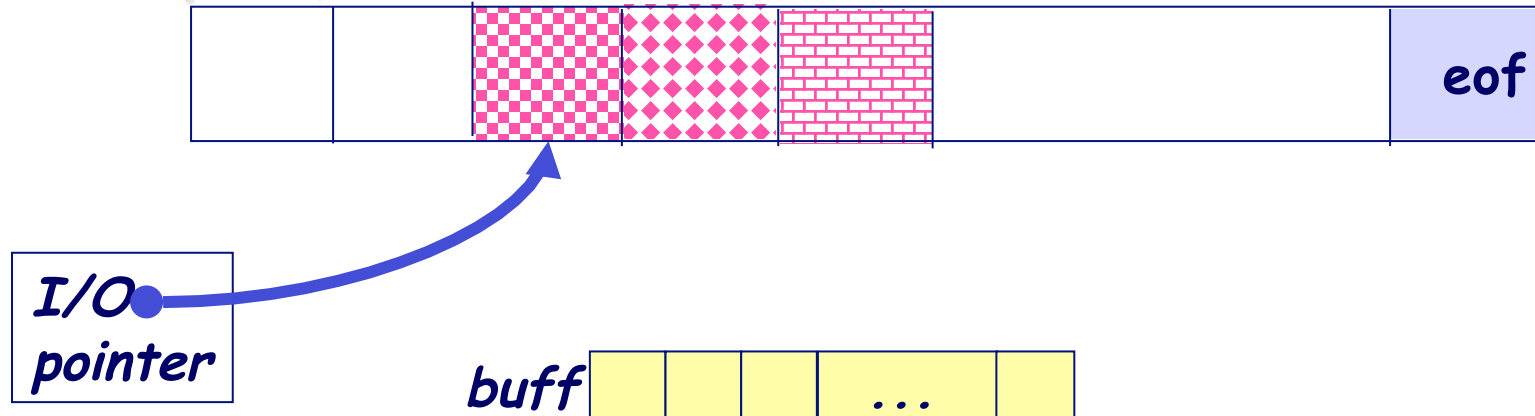
- `fd` è il file descriptor del file
- `buf` è l'area in cui trasferire i byte letti
- `n` è il numero di caratteri da leggere
- in caso di successo, restituisce un intero positivo ( $\leq n$ ) che rappresenta il numero di caratteri effettivamente letti
- è previsto un carattere di *End-Of-File* che marca la fine del file (da tastiera: ^D)

## Lettura di File: read

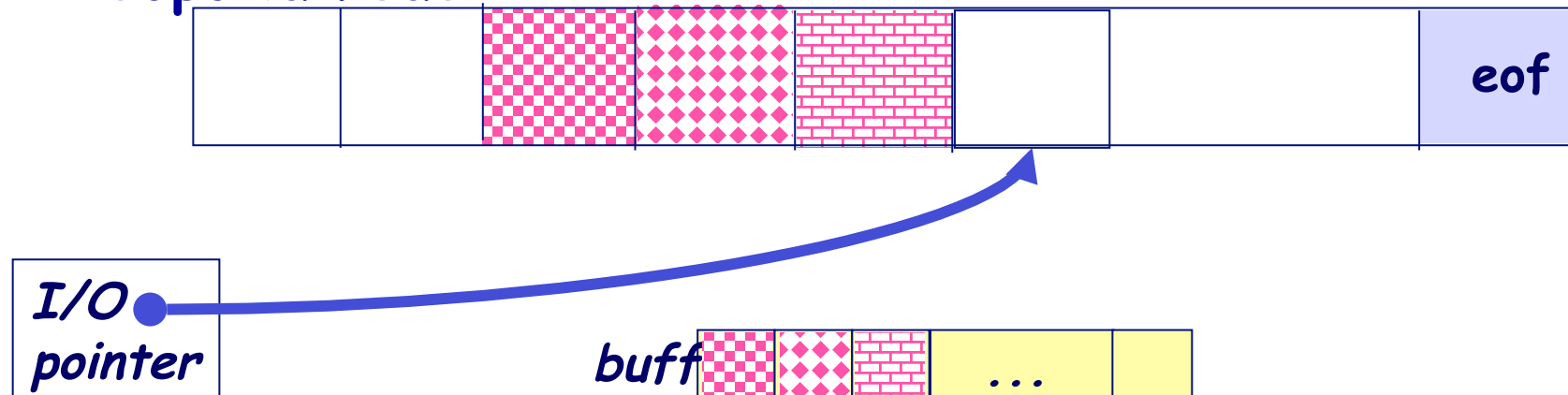
- Se `read(fd, buff, n)` ha successo:
  - a partire dal valore corrente dell'I/O pointer, vengono letti (al piu')  $n$  bytes dal file `fd` e memorizzati all'indirizzo `buff`.
  - *L'I/O pointer* viene spostato avanti di  $n$  bytes

# Lettura di File: read

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



- dopo la read:



## Scrittura di File: write

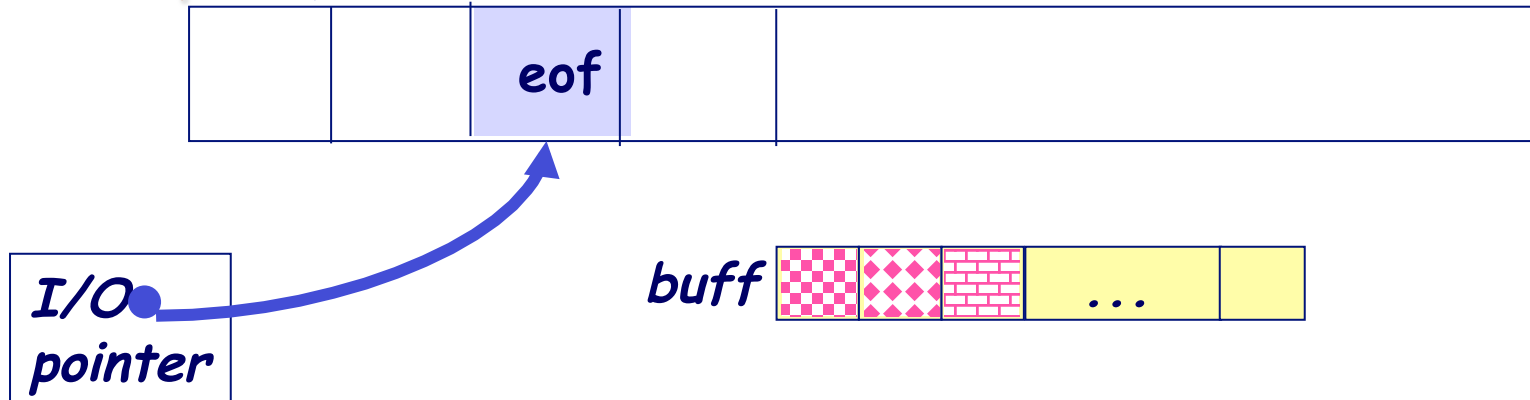
```
int write(int fd, char *buf, int n);
```

- `fd` è il file descriptor del file
  - `buf` è l'area da cui trasferire i byte scritti
  - `n` è il numero di caratteri da scrivere
- in caso di **successo**, restituisce un intero positivo  
(`= n`) che rappresenta il numero di caratteri effettivamente scritti

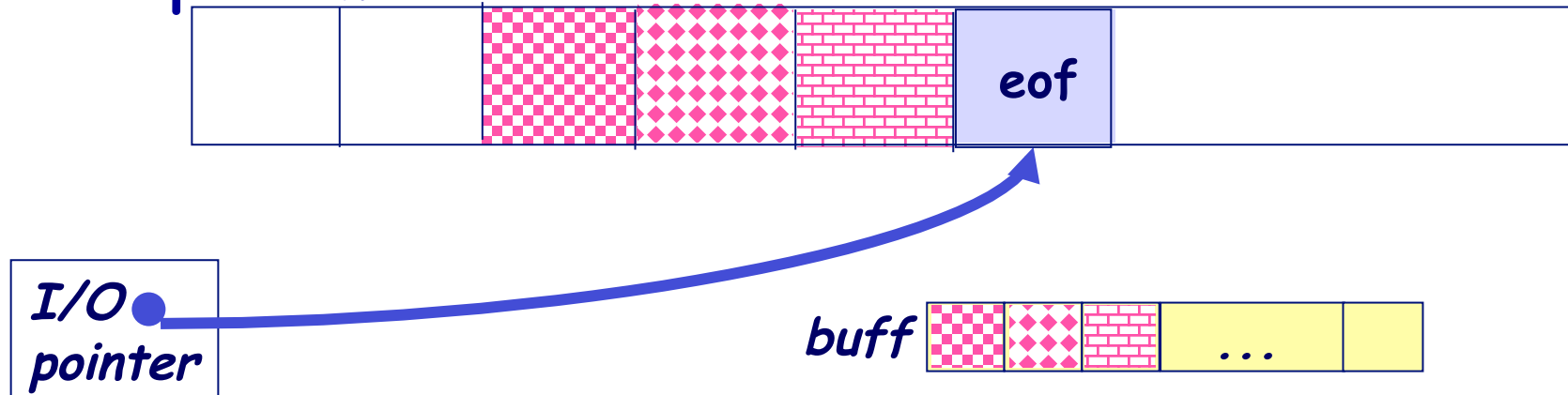


# Scrittura di File: write

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 su stdout */
 close(fd);
}
```

**Esempio:** comando mycp (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;
  int 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); }
```

```
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);
}
```

## "Accesso Diretto": lseek

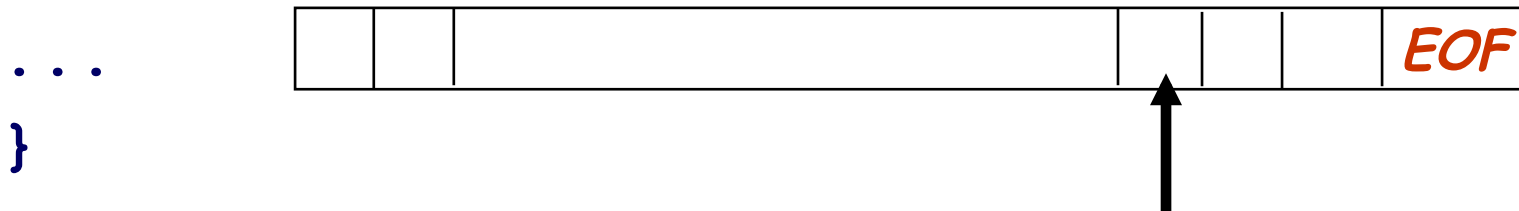
Per spostare l'I/O pointer:

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

- `fd` è il file descriptor del file
  - `offset` è lo 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
                    terz'ultimo byte
                    del file */
    ...
}
```



# Gestione di file: *system call*

- *I processi possono gestire i file tramite alcune **system call**, tra le quali:*
  - ✓ cancellazione: `unlink`
  - ✓ linking: `link`
  - ✓ Verifica dei diritti di accesso: `access`
  - ✓ Verifica degli attributi: `stat`
  - ✓ Modifica diritti di accesso: `chmod`
  - ✓ Modifica proprietario: `chown`

# unlink

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

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

- `name` è il 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), eliminando il nome specificato dalla struttura logica del file system; nel caso in cui il **numero dei link risulti 0**, allora il file viene **cancellato**.



# link

Per aggiungere un link a un file esistente:

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

- ❑ `oldname` è il nome del file esistente
- ❑ `newname` è il nome associato al nuovo link

Effetti:

- ❑ 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**.
- ❑ Fallisce (ad esempio), se:
  - ✓ `oldname` non esiste
  - ✓ `newname` esiste già (non viene sovrascritto!)
  - ✓ `oldname` e `newname` appartengono a file system diversi (in questo caso, usare softlinks mediante `symlink`).

# Esempio

Realizzazione del comando 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);}
```

# System call access

Per verificare i diritti di un utente di accedere a un file:

```
int access (char * pathname, int amode);
```

- il parametro `pathname` rappresenta il nome del file.
- Il parametro `amode` esprime il diritto da verificare e può essere:
  - » 04 read access
  - » 02 write access
  - » 01 execute access
  - » 00 existence
- `access` restituisce il valore 0 in caso di successo (diritto verificato), altrimenti -1.

**NB:** `access` verifica i diritti dell'utente, cioè fa uso del **real** uid del processo (non usa **effective** uid).

# System call stat

Per leggere gli attributi di un file (v.inode):

```
int stat(const char *path, struct stat *buf) ;
```

- il parametro `path` rappresenta il nome del file.
- il parametro `buf` è il puntatore a una **struttura di tipo `stat`**, nella quale vengono restituiti gli attributi del file (definito nell'header file `<sys/stat.h>`).

Ritorna 0, in caso di successo, -1 in caso di errore.

# Struttura stat

```
struct stat {  
    dev_t    st_dev;        /* ID of device containing file */  
    ino_t     st_ino;        /* i-number */  
    mode_t    st_mode;      /* protection */  
    nlink_t   st_nlink;     /* number of hard links */  
    uid_t     st_uid;       /* user ID of owner */  
    gid_t     st_gid;       /* group ID of owner */  
    dev_t     st_rdev;      /* device ID (if special file) */  
    off_t     st_size;      /* total size, in bytes */  
    blksize_t st_blksize;   /* blocksize for file system I/O */  
    blkcnt_t  st_blocks;    /* number of blocks allocated */  
    time_t    st_atime;     /* time of last access */  
    time_t    st_mtime;     /* time of last modification */  
    time_t    st_ctime;     /* time of last status change */  
};
```

---

## stat: st\_mode

Per interpretare il valore di `st_mode`, sono disponibili alcune costanti e macro (`<sys/stat.h>`); ad esempio:

- `S_ISREG(mode)`: è un file regolare? (flag `S_IFREG`)
- `S_ISDIR(mode)`: è una directory? (flag `S_IFDIR`)
- `S_ISCHR(mode)`: è un dispositivo a caratteri (file speciale)? (flag `S_IFCHR`)
- `S_ISBLK(mode)`: è un dispositivo a blocchi (file speciale)? (flag `S_IFBLK`)
- Ecc.

# stat: esempio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    struct stat sb;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(1);
    }
    if (stat(argv[1], &sb) == -1) {
        perror("stat");
        exit(1);
    }
}
```

```
printf("Tipo del file:\t");  
if (S_ISREG(sb.st_mode))    printf("file ordinario\n");  
if (S_ISBLK(sb.st_mode))    printf("block device\n");  
if (S_ISCHR(sb.st_mode))    printf("character device\n");  
if (S_ISDIR(sb.st_mode))    printf("directory\n");  
printf("I-number:\t%ld\n", (long) sb.st_ino);  
printf("Mode:\t%lo (octal)\n", (unsigned long) sb.st_mode);  
printf("numero di link:\t%ld\n", (long) sb.st_nlink);  
printf("Proprietario:\tUID=%ld    GID=%ld\n", (long) sb.st_uid,  
      (long) sb.st_gid);  
printf("I/O block size:\t %ld bytes\n", (long) sb.st_blksize);  
printf("dimensione del file:\t%ld bytes\n", (long)  
      sb.st_size);  
printf("Blocchi allocati: \t%ld\n", (long ) sb.st_blocks);  
exit(0);  
}
```



# Test:

```
$ ./provastat pippo.txt
```

```
Tipo del file: file ordinario
```

```
I-number: 13900906
```

```
Mode:      0644 (octal)
```

```
numero di link:      1
```

```
Proprietario:  UID=503      GID=503
```

```
I/O block size:      4096 bytes
```

```
dimensione del file:      1040 bytes
```

```
Blocchi allocati:      8
```

# System Call per la protezione: chmod

Per modificare i bit di protezione di un file:

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

- ❑ `pathname` è il nome del file
- ❑ `newmode` contiene i nuovi diritti

# System Call per la protezione: chown

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

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

- ▣ pathname è il nome del file
  - ▣ owner è l'uid del nuovo proprietario
  - ▣ group è il gid del gruppo
- 
- cambia proprietario/gruppo del file

# Gestione dei direttori

Vedremo alcune delle system call per la gestione dei direttori Unix. In particolare, analizzeremo:

- ▣ `chdir`: per cambiare direttorio (v. Comando `cd`)
- ▣ `opendir`, `closedir`: apertura e chiusura di direttori
- ▣ `readdir`: lettura di direttorio

Le primitive `opendir`, `readdir` e `closedir` fanno uso di tipi astratti e sono **indipendenti** da come il direttorio viene realizzato (Bsd, System V, Linux).

# Gestione di Direttori: `chdir`

- Per effettuare un cambio di direttorio (v. comando `cd`)

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

- ▣ `nomedir` è il nome del direttorio in cui entrare.
- ▣ Restituisce:
  - 0 in caso di **successo** (cioè il cambio di direttorio è avvenuto)
  - altrimenti restituisce -1 (in caso di **fallimento**)

# Accesso a direttori: lettura, scrittura

- Analogamente al file, Lettura/scrittura di un direttorio può avvenire soltanto dopo l'operazione di apertura (`opendir`).
- Una volta aperto, un direttorio può essere acceduto:
  - ▣ lettura (`readdir`): da tutti i processi con il diritto di lettura sul direttorio
  - ▣ scrittura: solo il kernel può scrivere sul direttorio
- L'operazione di apertura restituisce un puntatore a `DIR` (v. `FILE`):
  - ▣ `DIR` è un tipo di dato astratto predefinito (`<dirent.h>`) che consente di riferire (mediante puntatore) un direttorio aperto.

# Apertura di Direttori: opendir

Per aprire un direttorio:

```
#include <dirent.h>
```

```
DIR *opendir (char *nomedir);
```

- `nomedir` è il nome del direttorio da aprire
- la funzione 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 (in caso di insuccesso)

# Chiusura di un direttorio

Per chiudere un direttorio:

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

- Questa primitiva effettua la chiusura del direttorio riferito dal puntatore `dir`.
- Ritorna:
  - ▣ 0 in caso di successo
  - ▣ -1 in caso di fallimento



# Letture di un direttorio

Un direttorio aperto può essere letto con **readdir**:

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

- `dir` è il puntatore al direttorio da leggere (valore restituito da `opendir`)
- La funzione **readdir** restituisce:
  - un puntatore diverso da **NULL** se la lettura ha avuto **successo**
  - altrimenti **readdir** restituisce **NULL** (in caso di **insuccesso**)
- In caso di **successo**, la **readdir** legge un elemento dal direttorio dato e lo memorizza all'indirizzo puntato da **descr**.
- **descr** punta ad una struttura di tipo **dirent** (dichiarato in **dirent.h**).

# L'elemento del direttorio: dirent

Il generico elemento (file o direttorio) del direttorio 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; /* nome del file */  
}
```

- la stringa che parte da `d_name` rappresenta il nome del file (o direttorio) nel direttorio aperto; `d_namelen` rappresenta la lunghezza del nome
  - ➡ possibilità di nomi con lunghezza variabile

# Gestione di Direttori: creazione

## Creazione di un direttorio

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

- `pathname` è il nome del direttorio da creare
- `mode` esprime i bit di protezione
- `mkdir` restituisce il valore 0 in caso di successo, altrimenti un valore negativo.
- In caso di **successo**, crea e inizializza un direttorio con il nome e i diritti specificati; vengono sempre creati i file:
  - `.` (link al direttorio corrente)
  - `..` (link al direttorio 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;}
```

# Esempio: realizzazione del comando ls (continua)

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

## Esempio: esplorazione di una gerarchia

- Si vuole operare in modo ricorsivo su una gerarchia di direttori alla ricerca di un file con nome specificato.

Per esplorare la gerarchia utilizzeremo le funzioni per cambiare direttorio `chdir` e le funzioni `opendir`, `readdir` e `closedir`.

- Si preveda una sintassi del tipo:

`ricerca radice file`

- ▣ `radice`: nome del direttorio "radice" della gerarchia
- ▣ `file`: nome del file da ricercare nella gerarchia (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(".."); } /* salgo 1 livello */
  }
  closedir(dir);
}

```



# La Comunicazione tra Processi in Unix

# Interazione tra processi Unix

- I processi Unix non possono condividere memoria (modello ad ambiente locale)
- L'interazione tra processi può avvenire:
  - mediante la **condivisione di file**:
    - ✓ complessità : realizzazione della sincronizzazione tra i processi.
  - attraverso specifici **strumenti di *Inter Process Communication***:
    - per la comunicazione tra processi sulla stessa macchina:
      - » pipe (tra processi della stessa gerarchia)
      - » fifo (qualunque insieme di processi)
    - per la comunicazione tra processi in nodi diversi della stessa rete:
      - » socket

# pipe

La pipe è un canale di comunicazione tra processi:

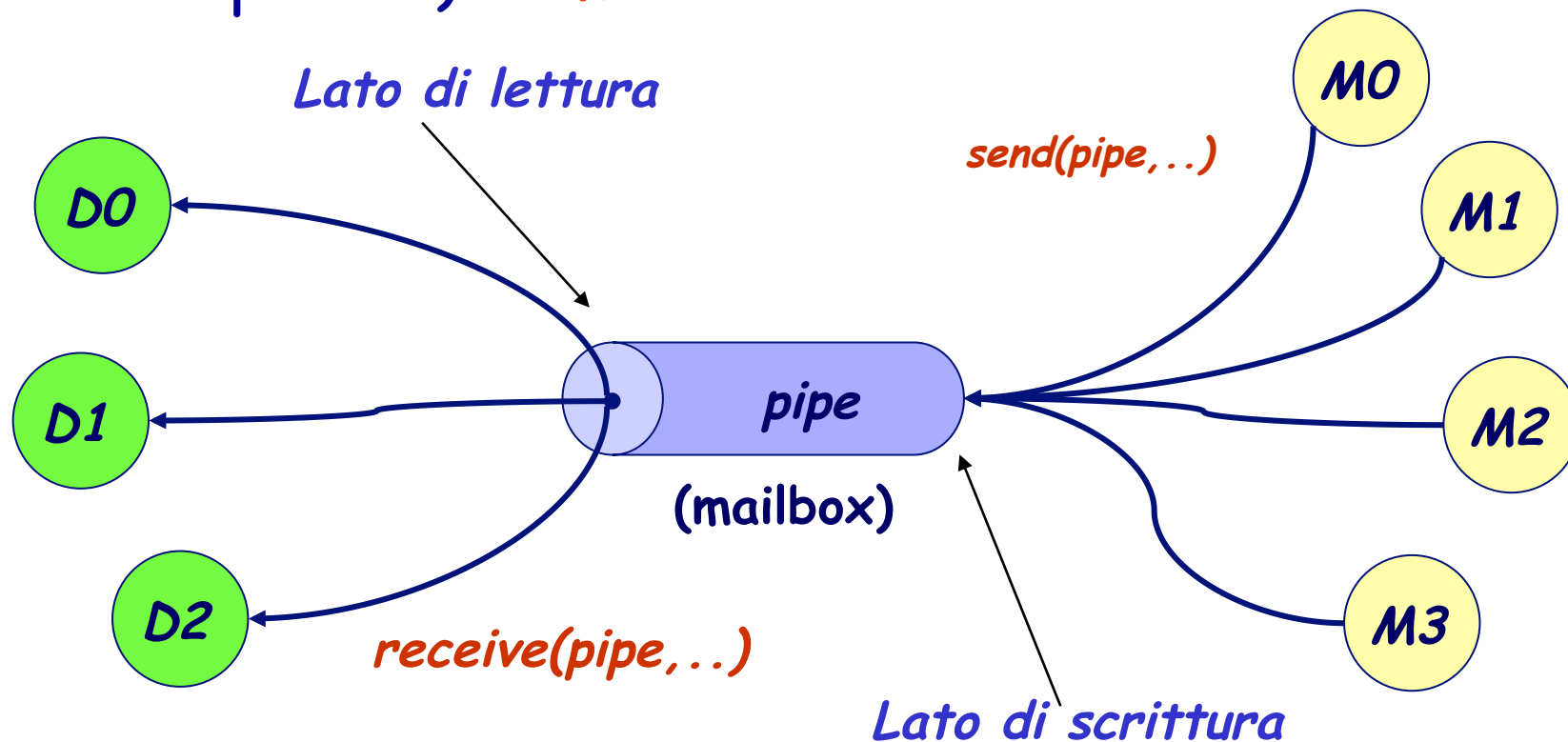
- ❑ **unidirezionale**: accessibile ad un estremo in lettura ed all'altro in scrittura
- ❑ **molti-a-molti**:
  - più processi possono spedire messaggi attraverso la stessa pipe
  - più processi possono ricevere messaggi attraverso la stessa pipe
- ❑ **capacità limitata**:
  - la **pipe** è in grado di gestire l'**accodamento** di un numero limitato di messaggi, gestiti in modo FIFO: il limite è stabilito dalla **dimensione** della pipe (es.4096 bytes).

⇒ **comunicazione asincrona**

- ❑ **OMOGENEITA'** con i FILE: accesso con le stesse system call dei file.

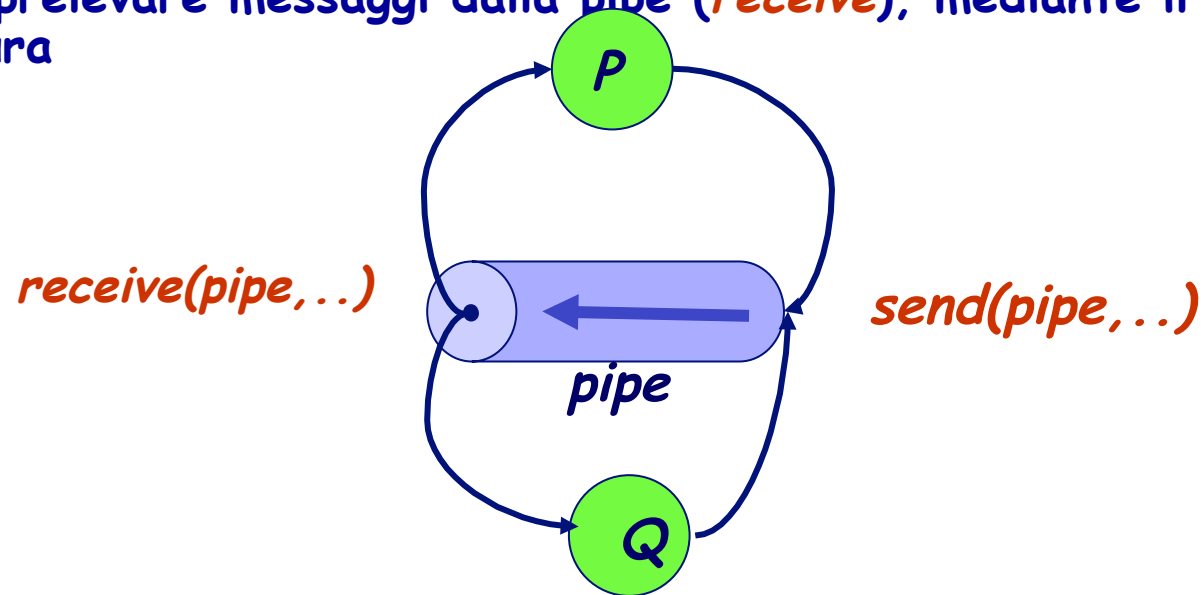
# Comunicazione attraverso pipe

- Mediante la pipe, la comunicazione tra processi è **indiretta** (senza naming esplicito): *mailbox*



# Pipe: unidirezionalità/bidirezionalità

- Uno stesso processo può:
  - sia depositare messaggi nella pipe (*send*), mediante il lato di scrittura
  - che prelevare messaggi dalla pipe (*receive*), mediante il lato di lettura



☞ la pipe può anche consentire una comunicazione “*bidirezionale*” tra i processi P e Q (ma va rigidamente disciplinata !)

# System call pipe

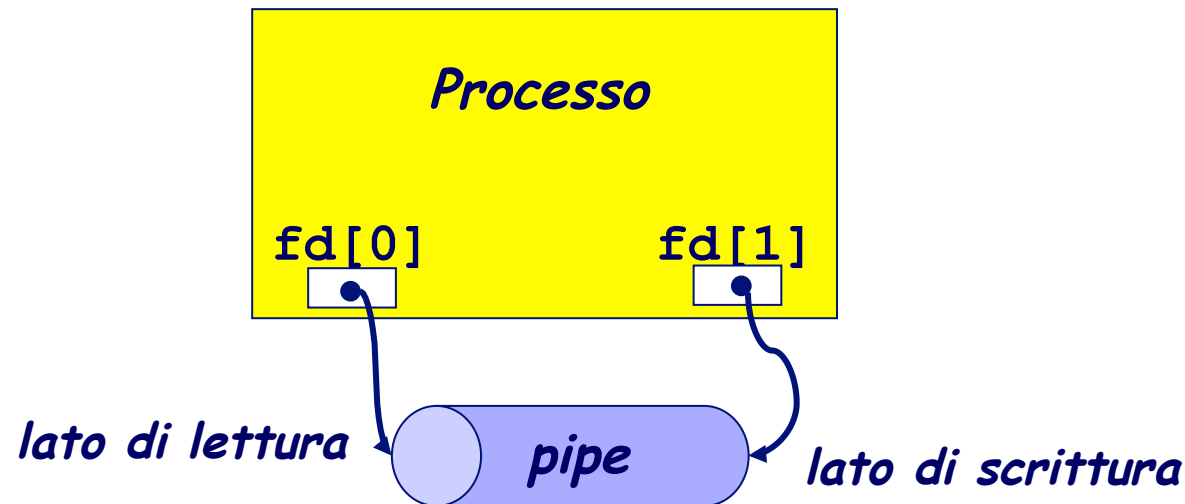
- Per creare una pipe:

```
int pipe(int fd[2]);
```

- `fd` è il puntatore a un vettore di 2 **file descriptor**, che verranno inizializzati dalla system call in caso di successo:
  - ✓ `fd[0]` rappresenta il lato di **lettura** della pipe
  - ✓ `fd[1]` è il lato di **scrittura** della pipe
- la system call `pipe` restituisce:
  - un valore negativo, in caso di fallimento
  - 0, se ha successo

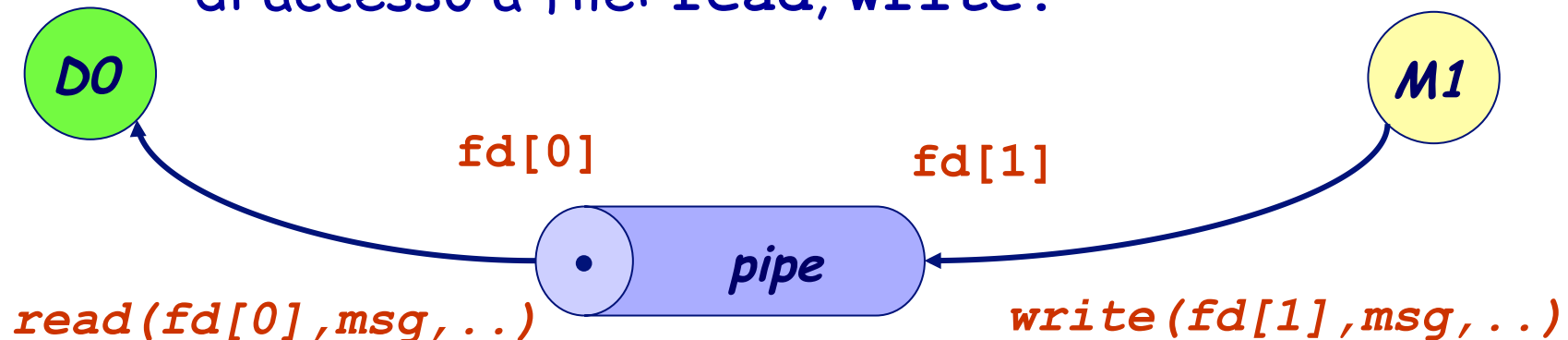
# Creazione di una pipe

- Se `pipe(fd)` ha successo:
  - ▣ vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a `fd[0]` e `fd[1]`:
    - ✓ `fd[0]`: lato di lettura (*receive*) della pipe
    - ✓ `fd[1]`: lato di scrittura (*send*) della pipe



# Omogeneità con i file

- Ogni lato di accesso alla pipe è visto dal processo in modo **omogeneo** al file (file descriptor):
  - ▣ si può accedere alla pipe mediante le system call di accesso a file: `read`, `write`:



- ✓ `read`: realizza la *receive*
- ✓ `write`: realizza la *send*



# Sincronizzazione dei processi comunicanti

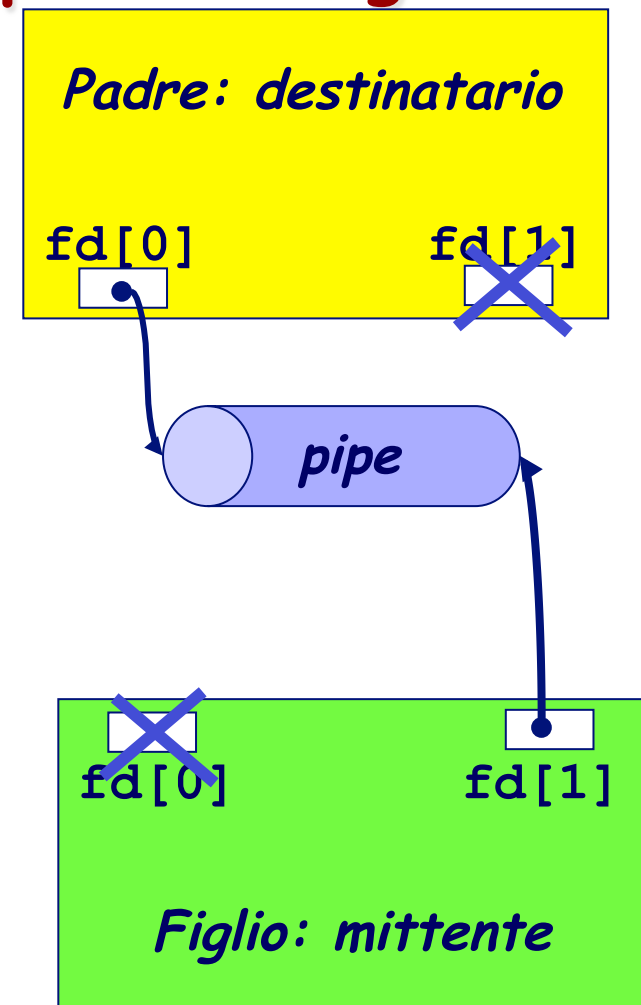
- Il canale (la *pipe*) ha capacità limitata: come nel caso di produttore/consumatore è necessario sincronizzare i processi:
  - se la *pipe* è **vuota**: un processo che legge si blocca
  - se la *pipe* è **piena**: un processo che scrive si blocca
- ✦ Sincronizzazione automatica: read e write da/verso pipe possono essere **sospensive** !

# Quali processi possono comunicare mediante pipe?

- Per mittente e destinatario il riferimento al canale di comunicazione è un file descriptor:
  - ✦ Soltanto i processi appartenenti a una stessa **gerarchia** (cioè, che hanno un **antenato** in comune) possono scambiarsi messaggi mediante pipe; ad esempio, possibilità di comunicazione:
    - tra **processi fratelli** (che ereditano la pipe dal processo padre)
    - tra un processo **padre** e un processo **figlio**;
    - tra **nonno** e **nipote**
    - etc.

## Esempio: comunicazione tra padre e figlio

```
main()
{int pid;
 char msg[]="ciao babbo";
 int fd[2];
 pipe(fd);
 pid=fork();
 if (pid==0)
 {/* figlio */
  close(fd[0]);
  write(fd[1], msg, 10);
  ...}
 else /* padre */
 { close(fd[1]);
  read(fd[0], msg, 10);
  ...
 }}
```



Ogni processo chiude il lato della pipe che non usa.

# Chiusura di pipe

- Ogni processo può chiudere un estremo della pipe con una **close**.
- Un estremo della pipe viene *effettivamente* chiuso (cioè, la comunicazione non è più possibile) quando tutti i processi che ne avevano visibilità hanno compiuto una **close**
- Se un processo P:
  - tenta una **lettura** da una pipe vuota il cui *lato di scrittura* è *effettivamente* chiuso: **read** ritorna 0
  - tenta una **scrittura** da una pipe il cui *lato di lettura* è *effettivamente* chiuso: **write** ritorna -1, ed il segnale **SIGPIPE** viene inviato a P (*broken pipe*).

# Esempio

```
/* Sintassi: progr N
padre(destinatario) e figlio(mittente) si scambiano una
sequenza di messaggi di dimensione (DIM) costante; la
lunghezza della sequenza non e` nota a priori; il
destinatario decide di interrompere la sequenza di
scambi di messaggi dopo N secondi */
```

```
#include <stdio.h>
```

```
#include <signal.h>
```

```
#define DIM 10
```

```
int fd[2];
```

```
void fine(int signo);
```

```
void timeout(int signo);
```

# Esempio

```
main(int argc, char **argv)
{int pid, N;
  char messaggio[DIM]="ciao ciao ";
  if (argc!=2)
  {   printf("Errore di sintassi\n");
      exit(1);}
  N=atoi(argv[1]);
  pipe(fd);
  pid=fork();
  if (pid==0) /* figlio */
  {   signal(SIGPIPE, fine);
      close(fd[0]);
      for(;;)
          write(fd[1], messaggio, DIM);
  }
```

## Esempio

```
else if (pid>0) /* padre */
{
    signal(SIGALRM, timeout);
    close(fd[1]);
    alarm(N);
    for(;;)
    {
        read(fd[0], messaggio, DIM);
        write(1, messaggio, DIM);
    }
}
}/* fine main */
```

# Esempio

```
/* definizione degli handler dei segnali */
void timeout(int signo)
{ int stato;
  close(fd[0]); /* chiusura effettiva del lato di
                 lettura*/

  wait(&stato);
  if ((char)stato!=0)
    printf("Term. inv. figlio (segnale %d)\n",
           (char)stato);
  else printf("Term. Vol. Figlio (stato %d)\n",
              stato>>8);

  exit(0);
}

void fine(int signo)
{ close(fd[1]);
  exit(0);
}
```

---



# System call dup

- Per duplicare un elemento della tabella dei file aperti di processo:

**int dup(int fd)**

- ▣ fd è il file descriptor del file da duplicare

L'effetto di una `dup` è copiare l'elemento `fd` della tabella dei file aperti nella prima posizione libera (quella con l'indice minimo tra quelle disponibili).

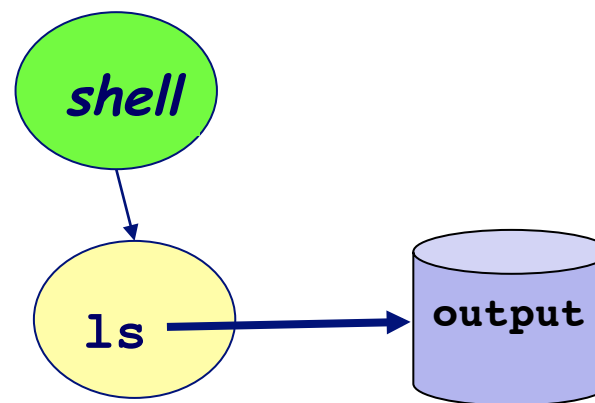
- Restituisce il nuovo file descriptor (del file aperto copiato), oppure `-1` (in caso di errore).

## Dup & ridirezione

- Mediante la dup si puo` realizzare la ridirezione di comandi su file; ad esempio:

```
$ ls -lR > output
```

- ➔ Viene creato un processo che andrà ad eseguire il comando `ls` in modo tale che
- ▣ Lo std. output di `ls` sia ridiretto nel file `output`



## Esempio: ridirezione di un comando su file

`/* sintassi: programma com f significa: com > f */`

```
main(int argc, char **argv)
{ int pid, fd, status;
  pid=fork();
  if (!pid) /* processo figlio: com1 */
  {   fd=open(argv[2], O_WRONLY);
      close(1);
      dup(fd); /* fd sostituisce il disp. di stdout */
      close(fd);
      execlp(argv[1], argv[1], (char *)0);
      exit(-1);
  }
  wait(&status);
  if ((char)status!=0)
      printf("figlio term. per segnale%d\n", (char)status);
}
```

## Esempio: mediante la dup è possibile ridirigere stdin e stdout su pipe

```
main()
{ int pid, fd[2]; char msg[3]="bye";
  pipe(fd);
  pid=fork();
  if (!pid) /* processo figlio */
  { close(fd[0]);
    close(1);
    dup(fd[1]); /* ridirigo stdout sulla pipe */
    close(fd[1]);
    write(1,msg, sizeof(msg)); /*scrivo su pipe*/
    close(1);
  }else /*processo padre
  { close(fd[1]);
    read(fd[0], msg, 3);
    close(fd[0]);}}
```

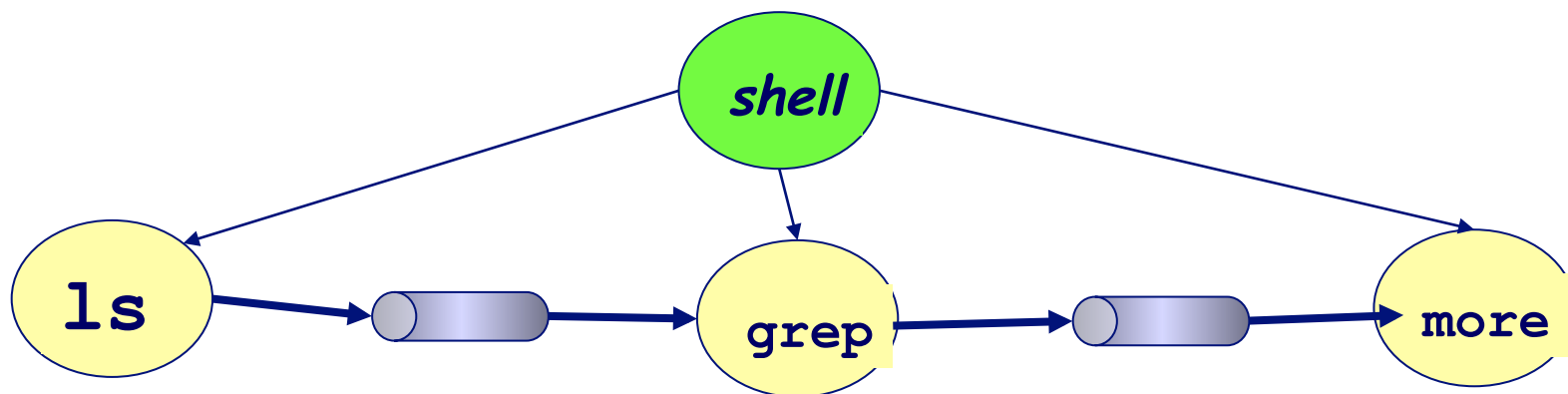
## Dup & piping

- Mediate la dup si puo' realizzare il *piping* di comandi; ad esempio:

```
$ ls -lR |grep Jun |more
```

➔ Vengono creati 3 processi (uno per ogni comando), in modo che:

- ▣ Lo std. output di `ls` sia ridiretto nello std. input di `grep`
- ▣ Lo std. output di `grep` sia ridiretto nello std. input di `more`



## Esempio: piping di 2 comandi senza argomenti

/\* sintassi: programma com1 com2 significa:  
com1|com2 \*/

```
main(int argc, char **argv)
{ int pid1, pid2, fd[2], i, status;
  pipe(fd);
  pid1=fork();
  if (!pid1) /* primo processo figlio: com2 */
  {   close(fd[1]);
      close(0);
      dup(fd[0]); /* ridirigo stdin sulla pipe */
      close(fd[0]);
      execlp(argv[2], argv[2], (char *)0);
      exit(-1);
  }
```

```

else /*processo padre
{  pid2=fork();
   if (!pid2) /* secondo figlio: com1 */
   {   close(fd[0]);
       close(1);
       dup(fd[1]); /* ridirezione pipe-stdout */
       close(fd[1]);
       execlp(argv[1], argv[1], (char *)0);
       exit(-1);
   }
   for (i=0; i<2;i++)
   {   wait(&status);
       if((char)status!=0)
           printf("figlio terminato per segnale%d\n",
               (char)status);
   }
   exit(0);
}

```

# Pipe

- La pipe ha due svantaggi:
  - ▣ consente la comunicazione solo tra processi in relazione di parentela
  - ▣ non è persistente: viene distrutta quando terminano tutti i processi che la usano.

Per realizzare la comunicazione tra una coppia di processi non appartenenti alla stessa gerarchia?

***FIFO***



# fifo

- È una pipe con nome nel file system:
  - canale unidirezionale del tipo *first-in-first-out*
  - è rappresentata da un file nel file system: **persistenza**, **visibilità** potenzialmente globale
  - ha un proprietario, un insieme di diritti ed una lunghezza
  - è creata dalla system call `mkfifo`
  - è aperta e acceduta con le stesse system call dei file

# Creazione di una fifo: `mkfifo`

Per creare una fifo:

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

- ❑ `pathname` è il nome della fifo
- ❑ `mode` esprime i permessi

Restituisce:

- ❑ 0, in caso di successo
- ❑ un valore negativo, in caso contrario

# Apertura/Chiusura di fifo

- Una volta creata, una fifo può essere aperta (come tutti i file), mediante una `open`; ad esempio, un processo destinatario di messaggi:

```
int fd;
```

```
fd=open("myfifo", O_RDONLY);
```

- Per chiudere una fifo, si usa la `close`:

```
close(fd);
```

- Per eliminare una fifo, si usa la `unlink`:

```
unlink("myfifo");
```

## Accesso a fifo

- Una volta aperta, la fifo può essere acceduta (come tutti i file), mediante `read/write`; ad esempio, un processo destinatario di messaggi:

```
int fd;  
char msg[10];  
fd=open("myfifo", O_RDONLY);  
read(fd, msg, 10);
```