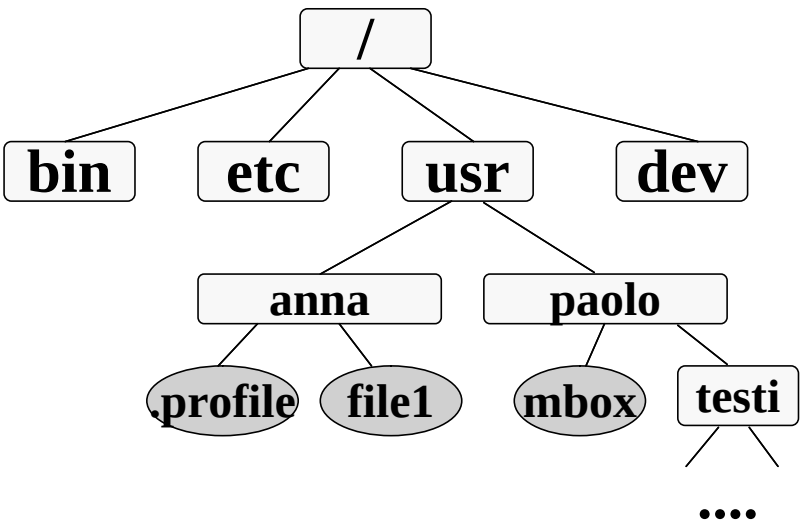
Il File System di Unix

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**

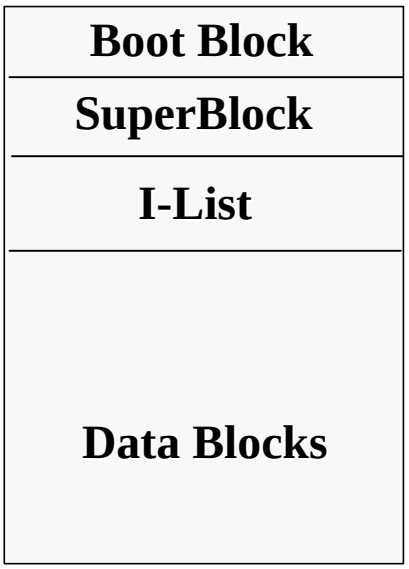
ma

- 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

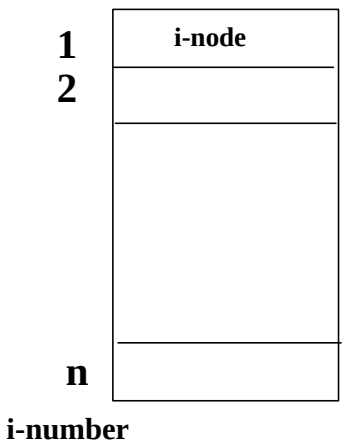


Il File System
Organizzazione Fisica

- **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.

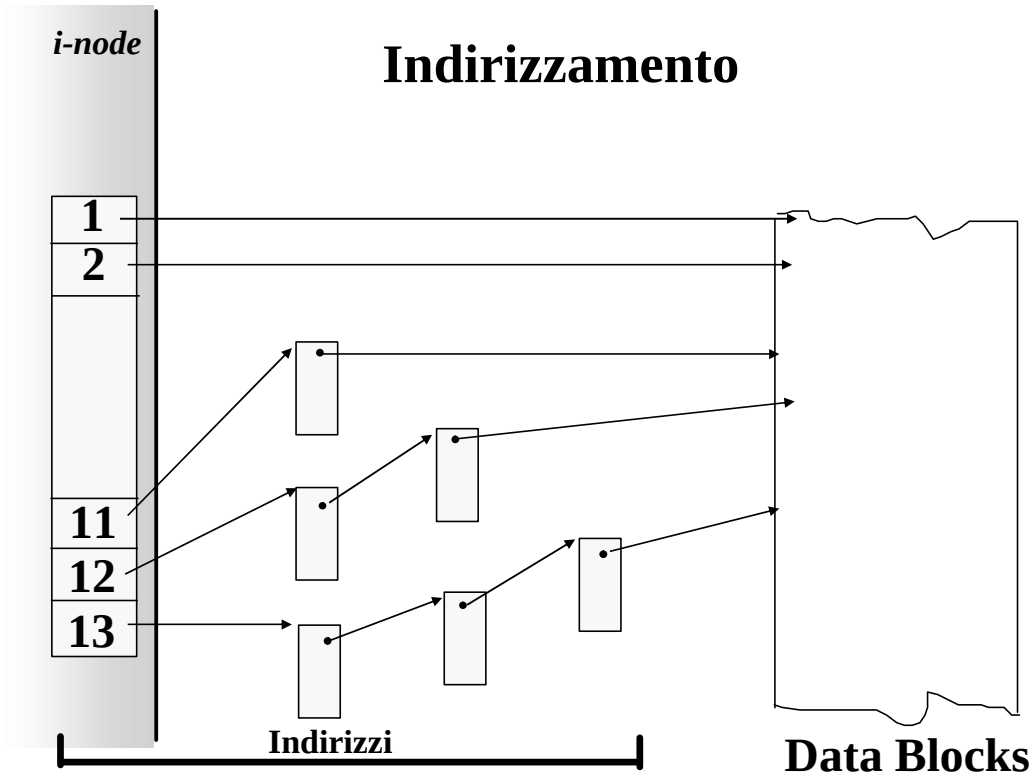
- Tra gli **attributi** contenuti nell'*i-node* vi sono:
  - **tipo** di file:
    - ✓ **ordinario**
    - ✓ **direttorio**
    - ✓ file **speciale**, per i dispositivi.
  - **proprietario, gruppo** (user-id, group-id)
  - **dimensione**
  - **data**
  - 12 bit di **protezione**
  - numero di **links**
  - **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° indirzzo**: tre livelli di *indirettezza*

Indirizzamento



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*512)byte=5120 byte =5 KB sono accessibili direttamente
- 128 blocchi di dati sono sono accessibili con ***indirezione singola*** (mediante il puntatore 11): **(128*512)byte = 65536 byte=64KB**
- 128*128 blocchi di dati sono accessibili con ***indirezione doppia*** (mediante il puntatore 12): **128*128*512 byte= 8MB**
- 128*128*128 blocchi di dati sono accessibili con ***indirezione tripla*** (mediante il puntatore 13): **128*128*128*512 byte= 1GB**

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**

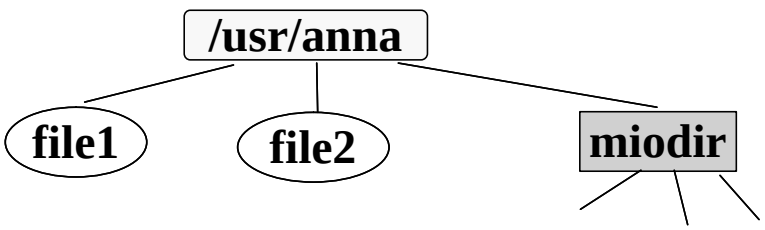
Il Direttorio

- 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**

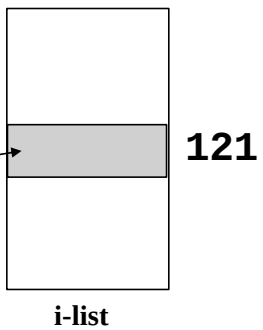
- ogni record rappresenta un file appartenente al direttorio:
  - per ogni file (o direttorio) appartenente al direttorio considerato, viene memorizzato il suo nome relativo, al quale viene associato il valore del suo ***i-number*** (che lo identifica univocamente)

Il Direttorio



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

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



Gestione di File
in Unix

Quali sono i meccanismi di accesso al file system?

Come vengono realizzati ?

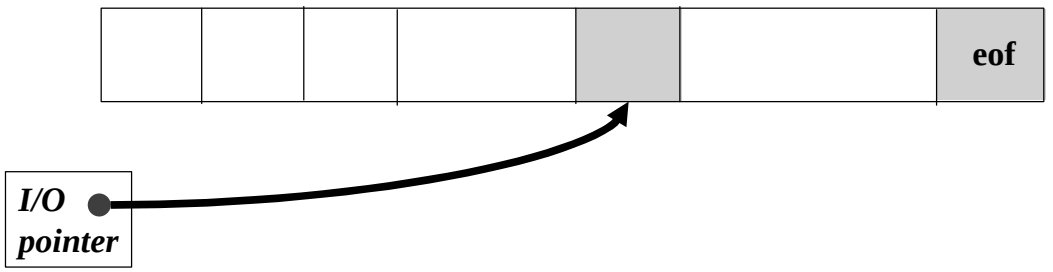
Vanno considerati:

- **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

Gestione di File
Concetti Generali

- accesso sequenziale
- assenza di strutturazione:

  file = sequenza di bytes (*stream*)
- posizione corrente: **I/O Pointer**



Gestione di File
Concetti Generali

- vari **modi** di accesso (***lettura, scrittura, lettura/scrittura***, etc.)

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

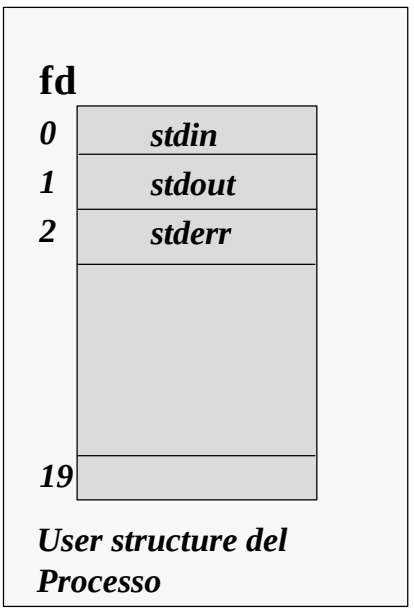
<apertura File>
<accesso al File>
<chiusura File>

Gestione di 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**



Gestione di 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.

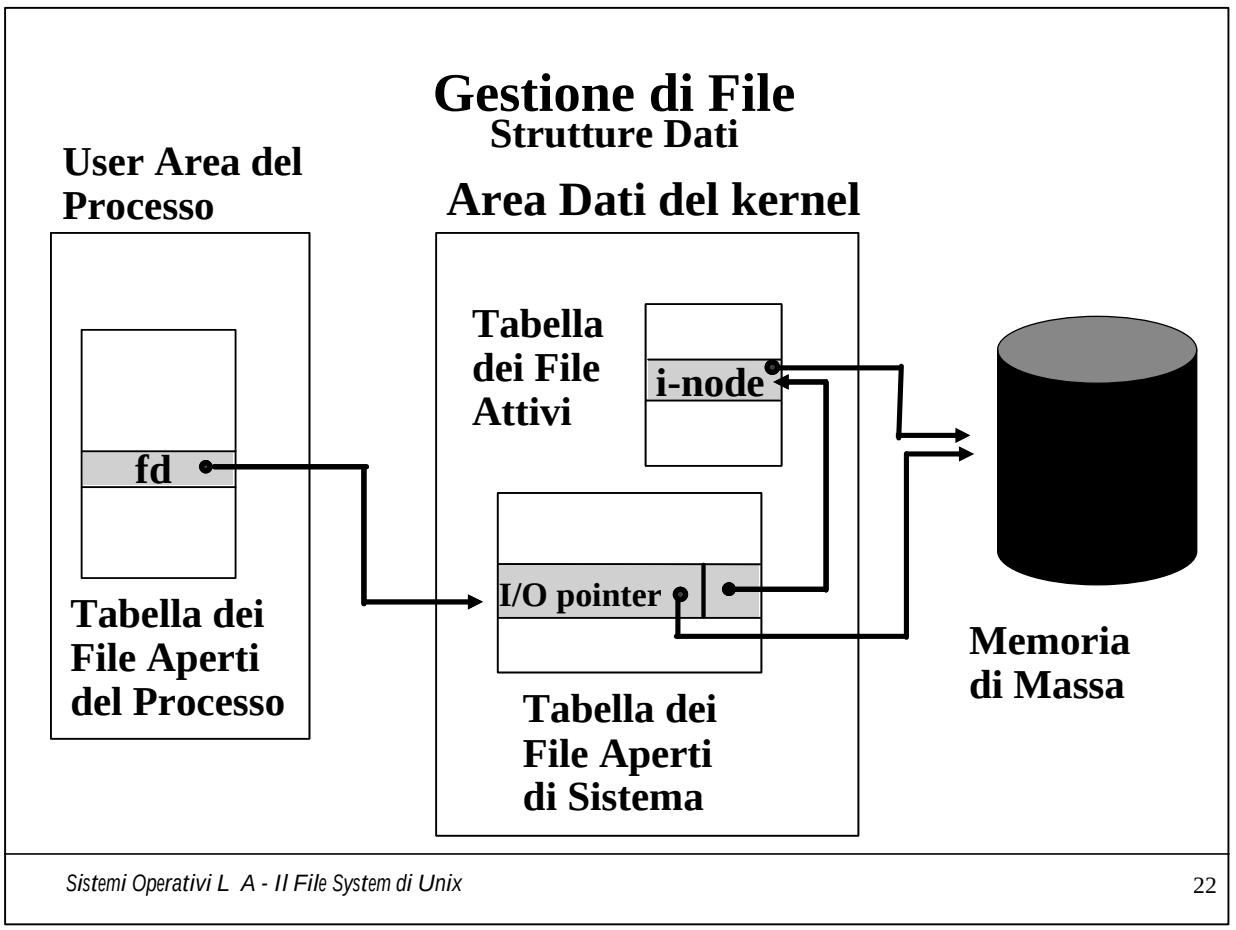
- la **tabella dei file aperti di sistema:** ha un elemento per ogni operazione di apertura relativa a file aperti (e non ancora chiusi); ogni elemento contiene:
  - ✓**l'I/O pointer**, che indica la posizione corrente all'interno del file
  - ✓un puntatore all'***i-node*** del file nella tabella dei file attivi
  - se due processi aprono separatamente lo stesso file *F*, la tabella conterrà due elementi distinti associati a *F*.

Gestione di 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
i-node
fd
I/O pointer
Tabella dei File Aperti del Processo
Tabella dei File Aperti di Sistema
Memoria di Massa

Gestione di File
Tabella dei File Aperti di Sistema

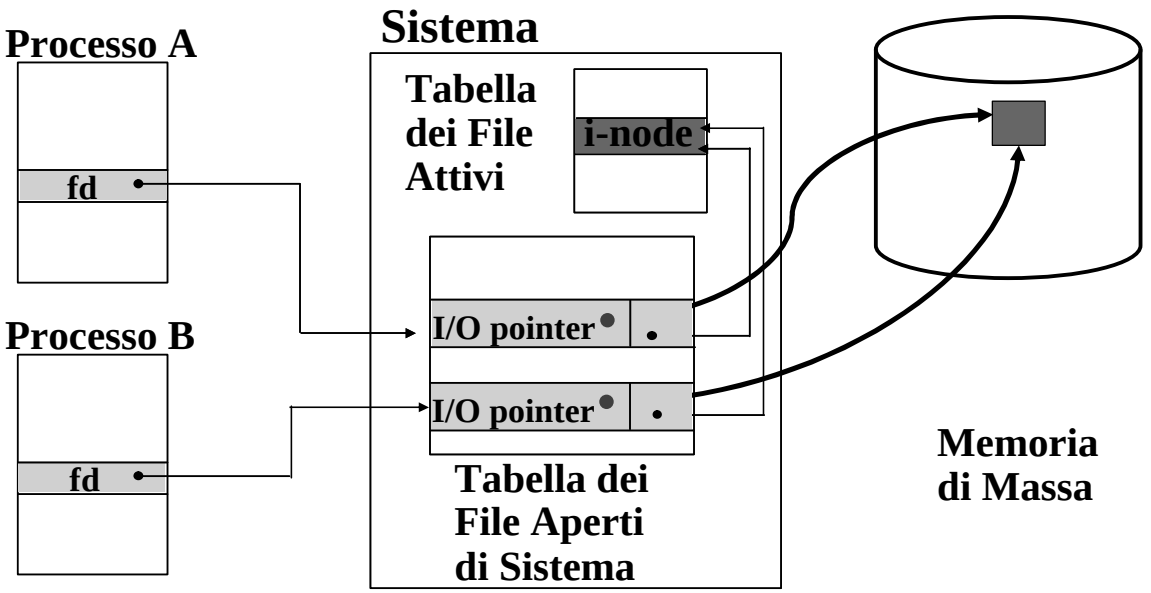
- un elemento per ogni "**apertura**" di file: a processi diversi che accedono allo stesso file, corrispondono entry distinte
- ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)

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

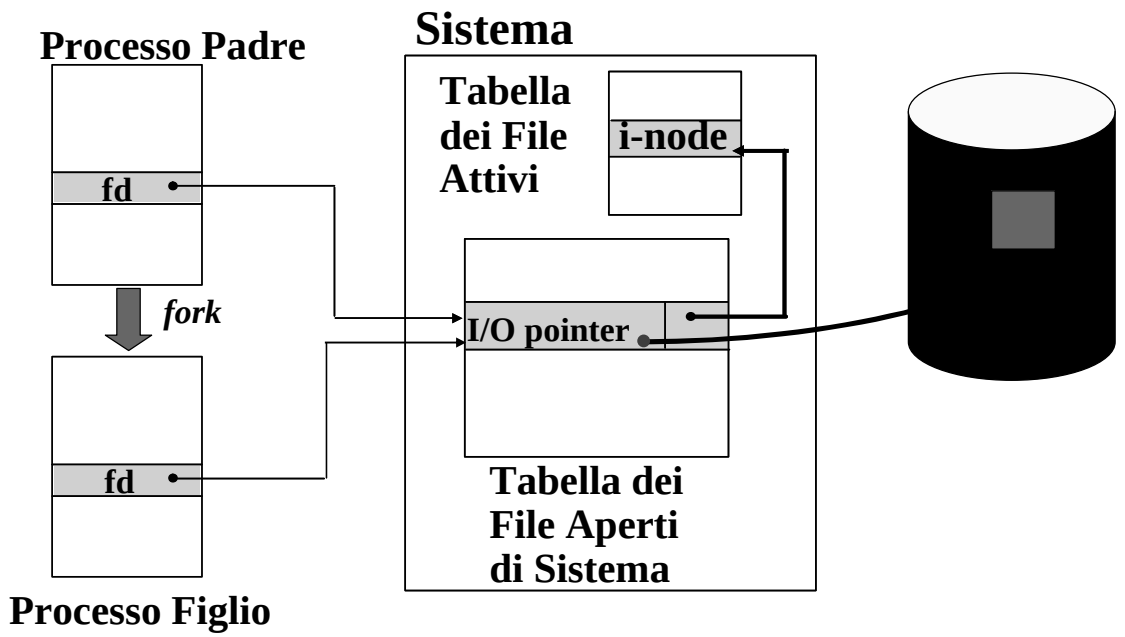
Gestione di File
Tabella dei File Attivi

- 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 <u>I/O pointer distinti</u>**



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



Gestione di 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`**
  - cancellazione: **`unlink`**
  - linking: **`link`**
  - accesso diretto: l**`seek`**

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'opzione di apertura determina la **creazione** del file: 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|O_TRUNC, 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 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 (<=n) che rappresenta il numero di caratteri effettivamente letti

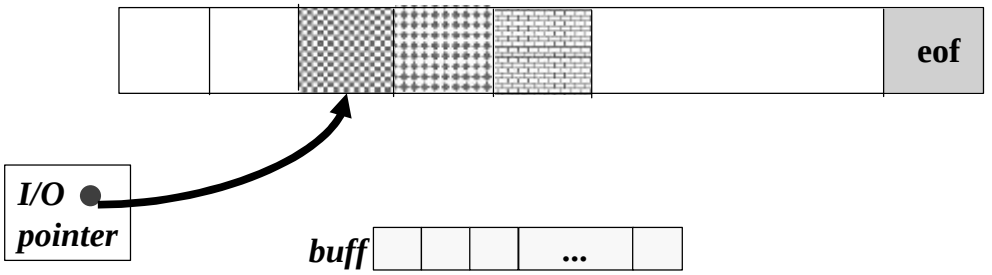
❒ è previsto un carattere di *End-Of-File* che marca la fine del file (da tastiera `:^D`)

Lettura di File: `read`

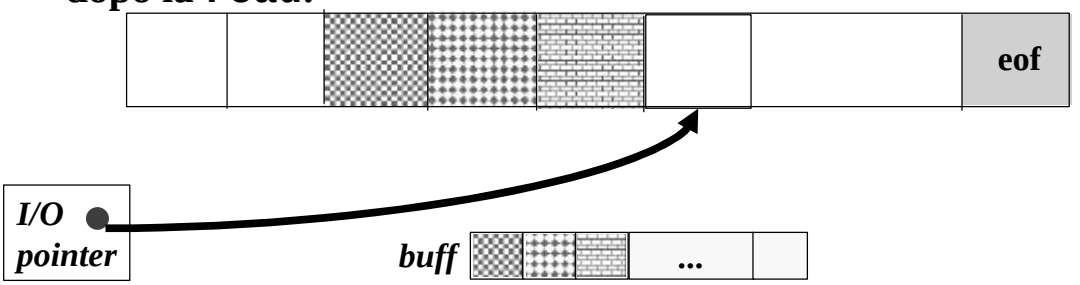
- Se **`read(fd, buff, n)`** ha successo:
  - a partire dal valore corrente dell'I/O pointer, vengono letti (al piu`) `n` bytes dal file **`fd`** e memorizzati all'indirizzo **`buff`**.
  - *L'I/O pointer* viene spostato avanti di **`n`** bytes

Lettura di File: `read`

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



- **dopo la `read`:**



Scrittura di File: `write`

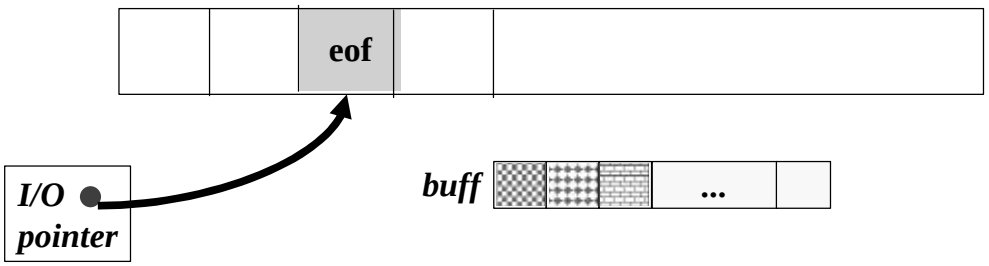
```
int write(int fd,char *buf,int n);
```

- **`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

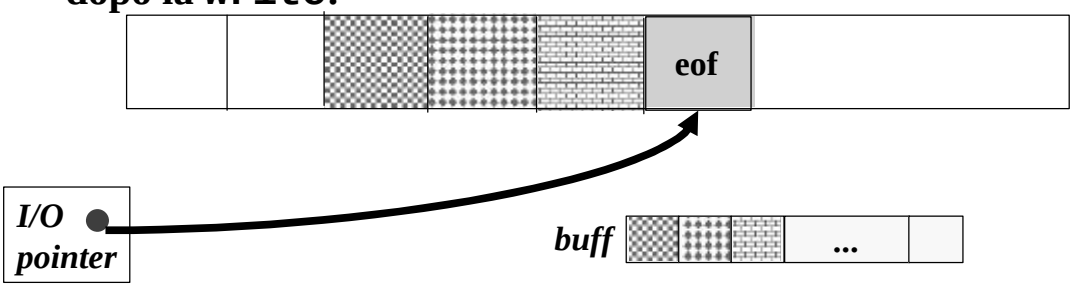
- in caso di **successo**, restituisce un intero positivo (= = n) che rappresenta il numero di caratteri effettivamente scritti

Lettura 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;
 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);
```

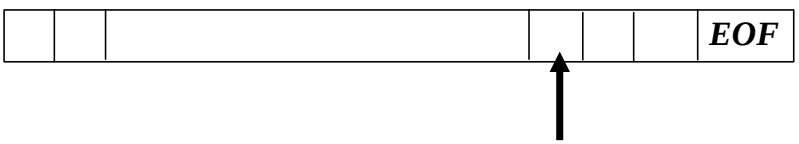
- **`fd`** è il file descriptor del file
- **`offset`** è lo spostamento (in byte) rispetto all'origine
- **`origine`** può valere:
  - ✓ 0: inizio file **(SEEK_SET)**
  - ✓ 1: posizione corrente **(SEEK_CUR)**
  - ✓ 2 :fine file**(SEEK_END)**

❒ 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 */

...
}
```



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); 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

La 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**.
- Fallisce (ad esempio), se:
  - **`oldname`** non esiste
  - **`newname`** esiste gia` (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, altrimenti -1.

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

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

Lettura 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 **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 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[1]; /* 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);}
```