Corso di Fondamenti di Informatica 2
CdL Ingegneria Informatica
Ing. Franco Zambonelli

IL SISTEMA OPERATIVO UNIX: GESTIONE FILE IN PROGRAMMI C PER UNIX

Lucidi Realizzati in Collaborazione con:

Prof. Letizia Leonardi
Università di Modena

Prof. Antonio Corradi
Università di Bologna

Prof. Cesare Stefanelli
Università di Ferrara

I File in Unix

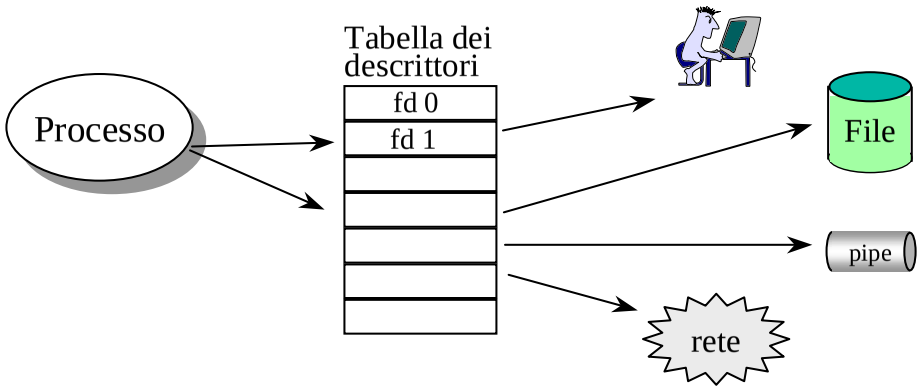
Un processo Unix vede tutto il mondo esterno (I/O) come un insieme di descrittori (da qui discende omogeneità tra file e dispositivi di Unix).

I file descriptor sono piccoli interi non negativi che identificano i file aperti

standard input, standard output, standard error sono associati ai file descriptor 0, 1, 2

Nuove operazioni di RICHIESTA producono nuovi file descriptor per un processo.

Numero massimo di fd per processo e per sistema



I processi interagiscono con l'I/O secondo il paradigma *open-read-write-close* (operazioni di prologo e di epilogo)

Flessibilità (possibilità di pipe e ridirezione)

System Call per operare a basso livello sui file
(`creat`, `open`, `close`, `read/write`, `lseek`)

Prologo (apertura/creazione di file):

CREATE fd = **creat**(name,mode);

```
int fd;          /* file descriptor  */
int mode;        /* attributi del file */
```

⇨ diritti di UNIX (di solito espressi in ottale)
⇨ file name aperto in scrittura

OPEN fd = **open**(name, flag);

```
char *name;
int flag;   /* 0 lettura, 1 scrittura, 2 entrambe */
int fd;     /* file descriptor  */
```

⇨ apre il file di nome **name** con modalità **flag**
⇨ in **/usr/include/fcntl.h** sono definite le costanti O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_EXCL

Esempi

```
fd=open("file", O_WRONLY| O_APPEND)
fd=open("file", O_WRONLY| O_CREAT| O_APPEND, 0644)
fd=open("file", O_WRONLY| O_CREAT| O_TRUNC, 0644)
fd=open("lock", O_WRONLY| O_CREAT| O_EXCL, 0644)
```

Epilogo (chiusura di file):

CLOSE retval = **close**(fd);

```
int fd, retval;
```

Operazioni di RICHIESTA e RILASCIO risorse
(max num. fd aperti per processo e per macchina)

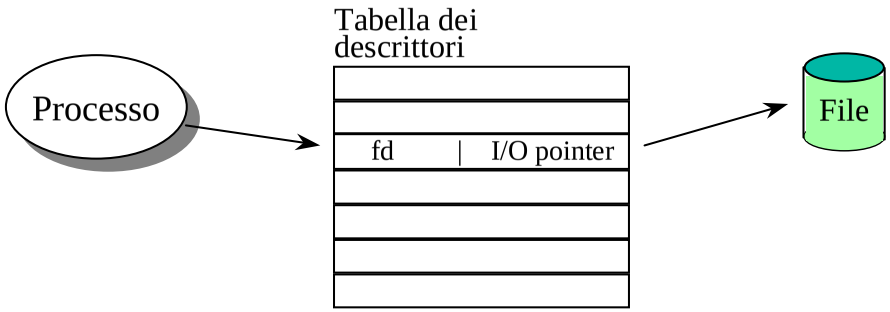
I FILE in UNIX

I file in Unix sono una sequenza di **BYTE**

ACCESSO **sequenziale**

I file sono rappresentati dai file descriptor

Presenza di **I/O pointer** associato al file (e al processo).
I/O pointer punta alla posizione corrente del file su cui il processo sta operando (scrivendo/leggendo)



File Descriptor

In generale, la lettura da fd 0 ⇒ legge da **standard input**
la scrittura su fd 1 ⇒ scrive su **standard output**
la scrittura su fd 2 ⇒ scrive su **standard error**

Questi tre **file descriptor** sono aperti *automaticamente*
dal **sistema** (shell) per ogni processo e collegati all'I/O

Per progettare **FILTRI**
cioè usare RIDIREZIONE e PIPING

i filtri leggono direttamente dal file descriptor 0
scrivono direttamente sul file descriptor 1

Completa omogeneità dei file con i dispositivi

```
fd = open ("/dev/printer", O_WRONLY);
```

Anche per i dispositivi usiamo le stesse primitive
open, read, write, close

Operazioni di Lettura e Scrittura

READ nread = **read**(fd, buf, n);

WRITE nwrite = **write**(fd, buf, n);

```
int nread, nwrite, n, fd;
char *buf;
```

- **lettura e scrittura** di un file avvengono a partire dalla **posizione corrente** del file ed avanzano il puntatore (**I/O pointer**) all'interno del file

- restituiscono:
  il **numero dei byte** su cui hanno lavorato
  -1 in caso di errore (come tutte system call)

Ogni utente ha la **propria visione** dei file aperti:

⇨ Nel caso di più utenti che aprono lo stesso file, ogni processo utente ha un proprio I/O pointer separato

⇨ **SE** un utente legge o scrive, modifica solo il proprio pointer, non modifica l'I/O pointer di altri

FILE SYSTEM
Un utente non ha visibilità delle azioni di un altro utente

Esempi di lettura/scrittura

COPIA da un FILE a un ALTRO

```
#include <fcntl.h>
#include <stdio.h>
#define perm 0644

main  ()
{ char f1 [20]= "file",
       f2 [40]= "/temp/file2";
   int infile, outfile; /* file descriptor  */
   int  nread;
   char buffer [BUFSIZ];

infile = open   (f1, O_RDONLY);
outfile = creat   (f2, perm);

while((nread = read(infile, buffer, BUFSIZ)) > 0)
           write (outfile, buffer, nread );

close (infile);
close (outfile);
}
```

Legge dal file *file* e scrive su *file2* in /temp

Copia da un File a un altro (uso argomenti)

```
#define perm 0777
main  (argc, argv)
int argc;
char **argv;

{     int infile, outfile, nread;
   char buffer [15];

infile  = open (argv[1], 0);
outfile = creat (argv[2], perm);
while (( nread = read (infile , buffer, 1)) > 0 )
          write (outfile, buffer, 1 );

close (infile);
close (outfile);
}
```

Con RIDIREZIONE

```
#define LUNG 1
main  ()
{ char  buffer [LUNG];
    while ( read   (0, buffer, LUNG) > 0 )
              write  (1 , buffer, LUNG);
}
```

Il sistema esegue i collegamenti tra file descriptor e file

Copia file con controllo degli errori

```
#include <fcntl.h>
#include <stdio.h>
#define perm 0744  /* tutti i diritti all'owner
                lettura al gruppo ed altri   */

main  (argc, argv)
int argc;
char **argv;
{ int status;
  int infile, outfile, nread;
  char  buffer[BUFSIZ];  /*buffer per i   caratteri
  */

  if (argc != 3)
        { printf (" errore \n"); exit (1); }

  if ((infile=open(argv[1], O_RDONLY)) <0)
        exit(1); /* Caso di errore */

  if ((outfile=creat(argv[2], perm )) <0)
           {close (infile); exit(1); }

  while((nread=read(infile, buffer, BUFSIZ)) >0 )
  { if(write(outfile, buffer, nread)< nread)
    {close(infile);close(outfile);exit(1);}
                    /* Caso di errore */
  }
  close(infile); close(outfile); exit(0);
}
```

Efficienza delle system call **read** e **write:** dipendenza dalle dimesioni del buffer

Esempio:
Inserimento di caratteri in un file

```
#include <fcntl.h>
#define perm 0744

main (argc, argv)
   int argc;   char **argv;
{  int fd;
   char *buff;
   int nr;

printf("il nome del file su cui inserire
      i caratteri è %s\n", argv[1]);

buff=(char *)malloc(80);
/* bisogna ALLOCARE memoria per il BUFFER */

if ((fd = open(argv[1], O_WRONLY)) < 0)
   fd = creat(argv[1], perm);
   /*oppure uso di open con quali flag?*/
printf("Aperto o creato con fd = %d\n", fd);

while ((nr=read(0, buff,80)) > 0)
   write(fd, buff, nr);

close(fd);
}
```

La Standard I/O Library

La Standard I/O library è costruita al di sopra delle System Call

E' una libreria contenente funzioni per accedere ai file a più **alto livello**.

Invece di file descriptor usa **stream** rappresentati da una struttura dati di tipo **FILE**

stdin è uno stream associato al file descriptor **standard input**
stdout è uno stream associato al file descriptor **standard output**
stderr è uno stream associato al file descriptor **standard error**

Fornisce:

formattazione ➔ printf("Ecco un intero %d \n", cont)
buffering ➔ Cosa fa la getc() ?
maggiore efficienza ➔ dimensione dei buffer

Si sconsiglia l'uso contemporaneo di System Call e funzioni della Standard I/O library nell'accesso a uno stesso file.

Operazioni **non** Sequenziali (random access)

LSEEK newpos = **lseek**(fd, offset, origin);
long int newpos, offset; int fd;
int origin; /* 0 dall'inizio, 1 dal corrente, 2 dalla fine*/

Si sposta la **posizione corrente** nel file per il processo invocante.

Le successive operazioni di lettura/scrittura a partire dalla nuova posizione

`lseek(fd, 10, 2)` cosa succede?

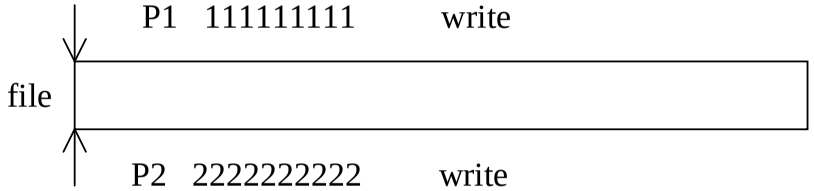
`lseek(fd, -10, 0)` ???

Operazioni sui dispositivi e file **solo sincrone**
cioè con attesa del completamento dell'operazione

ATOMICITÀ della SINGOLA OPERAZIONE
di lettura/ scrittura e di azione su un file.

Operazioni primitive

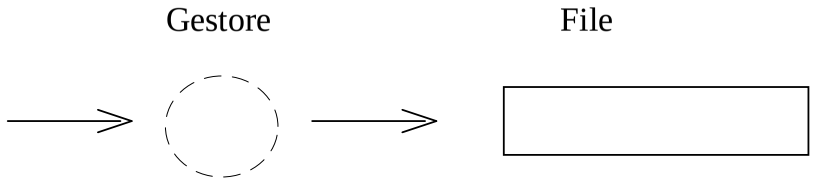
azioni elementari e non interrompibili della macchina virtuale UNIX



NON è garantita la
Atomicità delle sequenze di operazioni.

Per esempio
se più processi mandano file sulla stampante
Si possono mescolare le linee inviate alla stampante!!!!

==> Definizione di un **gestore** del file (system)
che incapsula la risorsa



I File e la multi-utenza

Ogni utente ha un identificatore detto **uid** (user id) e appartiene a un gruppo **gid** (group id), contenuti nel file /etc/passwd. Esempio:

```
franco:ITZ7b:230:30:F.Zambonelli:/home/franco:/bin/csh
```

Un processo acquisisce uid e gid dell'utente che lo lancia.

Il kernel memorizza per ogni file **user id** ed **group id** del processo creatore.

Un processo può accedere a un file se:
1. uid processo == 0
2. uid processo == uid proprietario file e diritti OK
3. uid processo != uid proprietario file ma
   gid processo == gid proprietario file e diritti OK
4. uid e gid proc != uid e gid file, ma diritti other OK

Attenzione: in realtà il kernel guarda **effective uid** e **gid** del processo che accede al file

Diritti di accesso a un file

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

ACCESS retval = **access** (pathname, amode);
char * pathname;
int amode;
int retval;

Il parametro ***amode*** può essere:
04 read access
02 write access
01 execute access
00 existence

access restituisce il valore 0 in caso di successo,
altrimenti -1 (e tipo di errore in errno)

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

Diritti di accesso a un file

Per cambiare i diritti di un **file**:

CHMOD retval = **chmod** (pathname, newmode);
char * pathname;
int newmode;
int retval;

Il parametro ***newmode*** contiene i nuovi diritti
chmod è eseguibile da owner o superuser

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

CHOWN retval = **chown** (pathname, owner_id, group_id);
char * pathname;
int owner_id;
int group_id;
int retval;

chown è eseguibile da owner o superuser

Problema: cosa succede se un file ha set-user-id settato?

Operazioni di LINK e UNLINK

UNLINK retval= **unlink**(name);
char *name;
int retval;

Questa primitiva consente di cancellare (DISTRUGGERE) un file

In realtà, come dice il suo nome, il suo compito è cancellare un link ➔ nel caso il numero di link arrivi a ZERO allora si opera anche la DISTRUZIONE del file cioè la liberazione dello spazio su disco

LINK retval= **link**(name1, name2);
char *name1, name2;
int retval;

Questa primitiva consente di creare un nuovo nome nome2 (un link) per un file esistente
➔ viene incrementato il numero di link

Problema dei diritti ➔ link guarda i diritti del direttorio

Tramite l'uso di queste due primitive viene realizzato, per esempio, il comando **mv** di UNIX

Esempio:
Implementazione del comando mv
(versione semplificata)

```
main (argc, argv)
   int argc;
   char **argv;
{
if (argc != 3)
   { printf ("Errore num arg\n"); exit(1); }
   /* controllo del numero di parametri */

if (link(argv[1], argv[2]) < 0)
   { perror ("Errore link"); exit(1);}
   /* controllo sulla operazione di link */

if (unlink(argv[1]) < 0)
   { perror("Errore unlink"); exit(1);}
   /* controllo sulla operazione di unlink */

printf ("Ok\n");
exit(0);
}
```

Direttori

a) Cambio di direttorio

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

Questa funzione **restituisce 0** se **successo** (cioè il cambio di direttorio è avvenuto),
altrimenti **restituisce -1** (in caso di **insuccesso**)

b) Apertura di direttorio

```
#include <dirent.h>
dir =   opendir (nomedir);
    char *nomedir;
    DIR *dir;
/* DIR è una struttura astratta e non usabile dall'utente */
```

Questa funzione **restituisce** un valore diverso da **NULL** se ha **successo** (cioè l'apertura del direttorio è avvenuta), altrimenti **restituisce NULL** (in caso di **insuccesso**)

c) Chiusura direttorio

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

Questa primitiva effettua la chiusura del direttorio

d) Lettura direttorio

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

La funzione **restituisce** un valore diverso da **NULL** se ha avuto **successo** (cioè a lettura del direttorio avvenuta), altrimenti **restituisce NULL** (in caso di **insuccesso**)
In caso di successo, descr punta ad una struttura 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 descr -> d_name rappresenta il nome di un file nel direttorio aperto

Questa stringa termina con un carattere nullo (convenzione C) ➔ possibilità di nomi con lunghezza variabile
La lunghezza del nome è data dal valore di d_namelen

Le primitive **chdir**, **opendir**, **readdir** e **closedir** sono **INDIPENDENTI** dalla specifica struttura interna del direttorio
➔ valgono sia per Unix BSD che per Unix System V

e) Creazione di un direttorio

MKDIR retval = **mkdir** (pathname, mode);
char * pathname;
int mode; /* diritti sul direttorio */
int retval;

La primitiva MKDIR crea un direttorio con il nome e i diritti specificati ===> vengono sempre creati i file

- **.** (link al direttorio corrente)
- **..** (link al direttorio padre)

mkdir restituisce il valore 0 in caso di successo, altrimenti un valore negativo

Altra primitiva è ***mknod*** il cui uso è però riservato al superuser (e non crea **.** e **..**)

Esempio:
Implementazione del comando ls

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

my_dir (name)
char *name;   /* nome del dir */
{  DIR *dir;  struct dirent * dd;
   int count = 0;

   dir = opendir (name);
   while ((dd = readdir(dir)) != NULL){
      printf("Trovato file %s\n", dd-> d_name);
      count++;
   }
   printf("Numero totale di file %d\n", count);
   closedir (dir);
   return (0);
}

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

Esempio:

Si vuole operare su una gerarchia di DIRETTORI alla ricerca di un file con nome specificato

Per ESPLORARE la gerarchia si utilizza la funzione per cambiare direttorio **chdir** e le funzioni **opendir**, **readdir** e **closedir**

```
/* file dirfun.c */
#define NULL 0
#include <sys/types.h>
#include <dirent.h>

/* La  soluzione seguente ASSUME che il nome del
direttorio sia dato in modo ASSOLUTO.
NOTA BENE: questa soluzione va bene se e solo se
il direttorio di partenza non è la radice (/).
PERCHÈ ? */

void esplora ();

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



```
/*  funzione  di esplorazione di   una  gerarchia:
opera in modo RICORSIVO */
void esplora (d, f)
char *d, *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))
               continue;
/* bisogna saltare i nomi del direttorio corrente
   e del direttorio padre */

      if (chdir(ff -> d_name) != 0) {
            /*è un file e non un direttorio*/
         if ( strcmp ( f, ff-> d_name) == 0)
            printf("file %s nel dir %s\n", f, d);
            /*eventuali altre operazioni sul file:
            ad esempio apertura,etc. */
      } else { /*abbiamo trovato un direttorio */
         strcpy(nd, d); strcat(nd, "/");
         strcat(nd, ff-> d_name);
         esplora ( nd, f);
         chdir("..");
         /* bisogna tornare su di un livello */
      }
      closedir(dir);
   }
}
```

Verifica dello stato di un file

STAT

```
#include <sys/types.h>
#include <sys/stat.h>
retval = stat (pathname, &buff);
        char * pathname;
        struct stat buff;
/* struttura che rappresenta il descrittore del file */
        int retval;
```

FSTAT

```
retval = fstat (fd, &buff);
        int fd;      /* file descriptor */
```

FSTAT può essere usato solo se il file è già aperto

Entrambe le primitive ritornano il valore 0 in caso di successo, altrimenti un valore negativo

Vediamo quali possono essere i campi della ***struct stat***:

```
struct stat {
    ushort   st_mode;   /* modo del file */
    ino_t    st_ino;    /* I_node  number */
    dev_t    st_dev;    /* ID del dispositivo */
    dev_t    st_rdev;   /* solo per file speciali */
    short    st_nlink;  /* numero di link */
    ushort   st_uid;    /* User ID del proprietario */
    ushort   st_gid;    /* Group ID del proprietario */
    off_t    st_size;   /* Lunghezza del file in byte */
    time_t   st_atime;  /* tempo dell'ultimo accesso */
    time_t   st_mtime;  /* tempo dell'ultima modifica*/
    time_t   st_ctime;   /* tempo ultimo cambiamento di stato */
            }
```


La struttura di un i-node

L' **i-node** è il descrittore del file

Struttura di un i-node per i ***dispositivi***

numero IDentificativo:
numero maggiore driver di dispositivo
(per una tabella di configurazione)
numero minore quale dispositivo

classe del dispositivo
a blocchi / a caratteri

Struttura di un i-node per i **file *normali/direttori***

- ***Proprietario*** user id , group id (UID e GID);

- ***tipo*** del file (ordinario, direttorio o special file);

- ***permessi*** read/write/execute per utente, gruppo e altri;

- i ***bit*** SUID, SGID, e 'sticky';

- numero dei ***link*** del file;

- ***dimensione*** del file

- ***indirizzi*** di tredici blocchi

RITROVARE i blocchi fisici del FILE

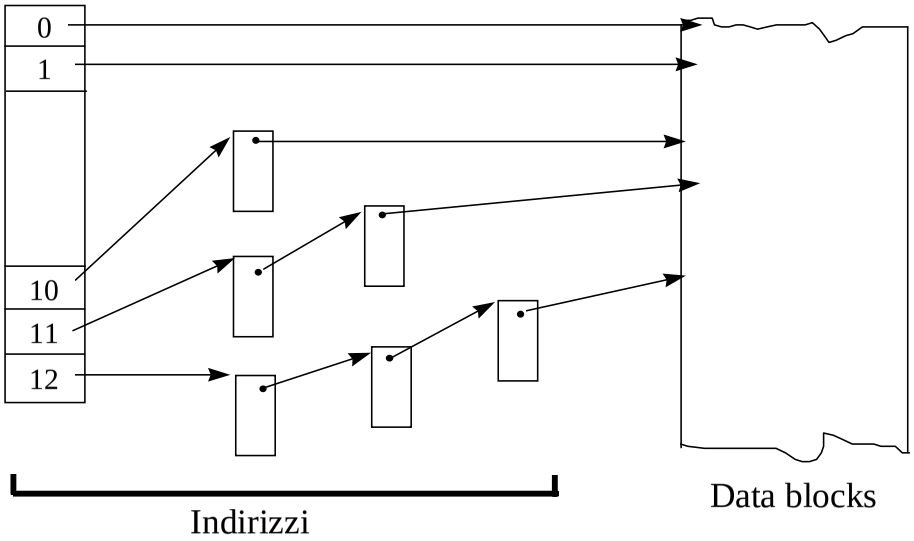
I primi dieci indirizzi **diretti**

L'undicesimo indirizzo è **indiretto**
indirizzo area che contiene gli indirizzi di blocchi

Es. con blocchi di 512 byte e ogni indirizzo di 4 byte
10+128 blocchi ➔ lunghezza file:
*10*512+128*512 = 70656*

Il dodicesimo **due livelli** di indirettezza:
*lunghezza massima di file 10+128+128*128 blocchi*

Il tredicesimo **tre livelli** di indirettezza fino a Gbyte:
*10+128+128*128+128*128*128 blocchi.*



Si favoriscono file di **media lunghezza**
File di dimensioni teoricamente **illimitata** (circa Gbyte)

Struttura fisica del file system

Il disco viene suddiviso in parti di dimensione fissa composte di blocchi

boot-block
"superblock" descrive il resto del file system (dim., struttura) una lista dei **blocchi liberi** una lista degli **i-node liberi**
"i-list" lista degli **i-node**
Blocchi dati

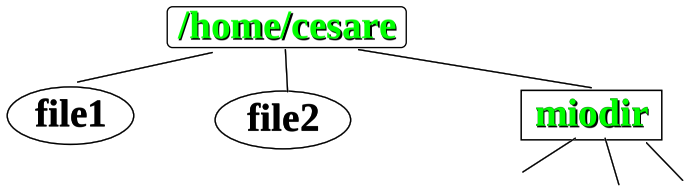
Il Direttorio

Un direttorio è un file con una struttura del tipo:

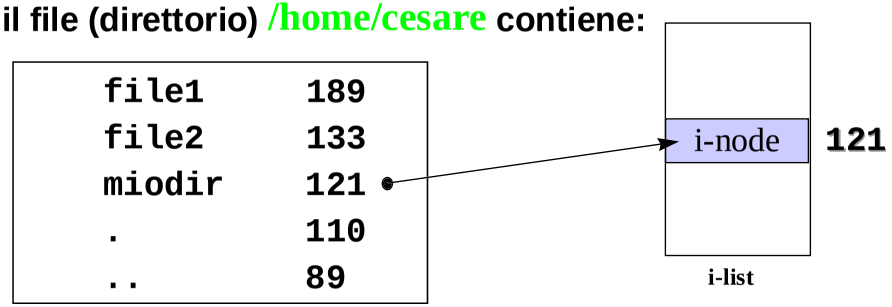
nomefile	i-number

nomefile è un nome relativo, non assoluto
i-number è collegato in modo univoco all' i-node del file

a ogni file (o direttorio) contenuto nel direttorio, viene associato l'i-number che lo identifica univocamente



il file (direttorio) /home/cesare contiene:



Cosa succede nel caso di `link("file1", "file3")` ??

Organizzazione del File System UNIX

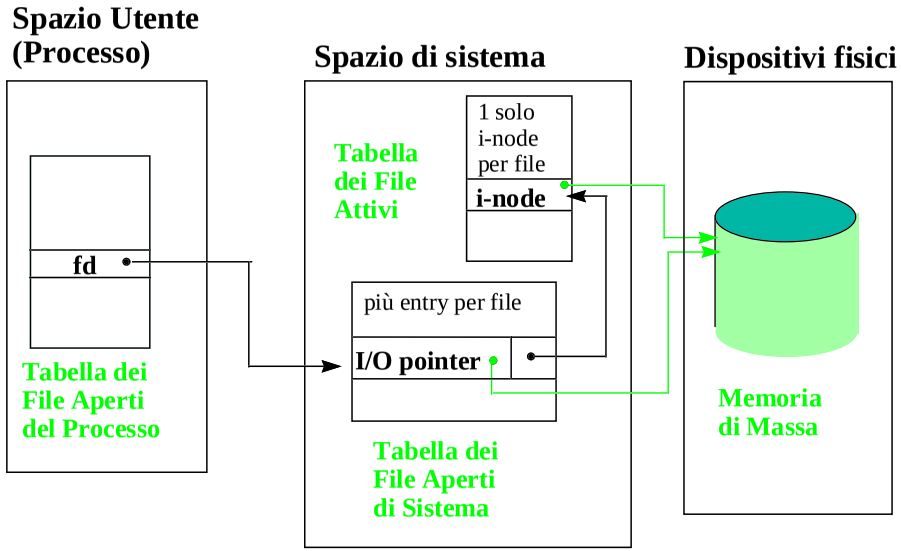


Tabelle globali:

Una sola tabella **file attivi** (**i-node attivi**) con un numero di entry pari al numero di file attivi (aperti anche da più processi)

Una sola tabella dei **file aperti**, con una entry per ogni apertura di file (possibili più entry referenti lo stesso file)

- un puntatore al corrispondente **i-node**;
- un puntatore (**I/0 pointer**)
  per le operazioni di lettura e scrittura sul file:
  punta al byte "corrente" del file a partire dal quale verranno effettuate le operazioni.

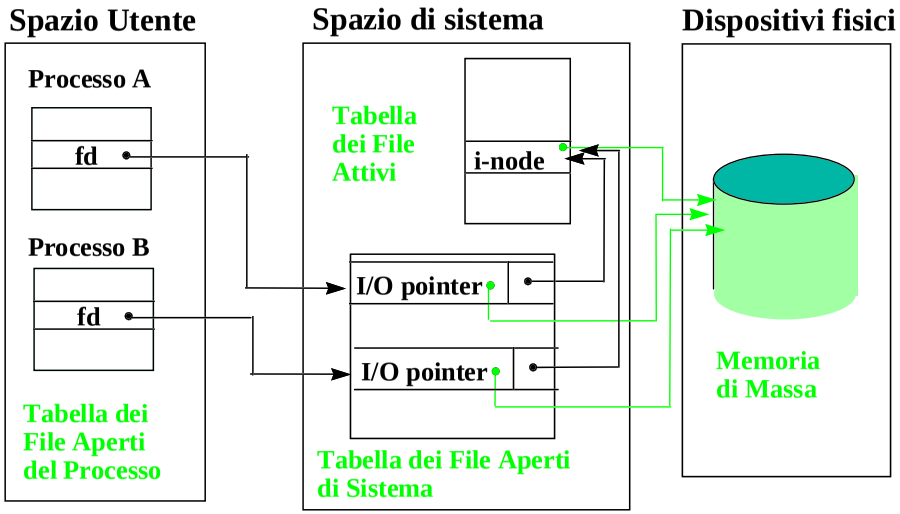
Apertura di un file

L'apertura di un file (system call `open()`) provoca:

- l'allocazione 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 del suo i-node nella tabella dei file attivi (se il file non è già stato aperto da un altro processo)

Condivisione di file

aperture separate di uno stesso file portano a condividere una sola entry della tabella degli i-node attivi, ma si avranno distinte entry nella tabella dei file aperti.



Condivisione di file

Caso di processo processo padre che apre un file prima di creare un processo figlio:

padre e figlio condividono la entry nella tabella dei file aperti, e anche la corrispondente entry nella tabella degli i-node attivi

➔ lo ***stesso I/O pointer***;