Input-Output di basso livello

File in UNIX

Il kernel di UNIX vede tutti i file come flussi non formattati di byte; il compito di interpretare ogni struttura logica interna ad un file è lasciato alle applicazioni.

I file UNIX sono strutture composte dai tre seguenti elementi:

Nome	**Stringa di caratteri alfanumerici utilizzata da utenti e programmi per fare riferimento al file.**
Inodo	**Struttura dati che contiene le informazioni sul file necessarie al SO per la sua gestione (attributi), oltre agli indirizzi per accedere ai dati contenuti nel file.**
Dati	**I dati effettivamente contenuti nel file.**

Il sistema assegna a ciascun file un identificatore numerico, detto ***i-***

Inodo

Le informazioni contenute all'interno di ogni i-nodo residente in un file system sono le seguenti:

Modo del file	**Flag di 16 bit dove sono memorizzati il tipo del file ed i suoi permessi di accesso (read, write ,execute)**
Contatore link	**Numero di riferimenti all'inodo nelle directory**
ID Proprietario	**Identificatore del proprietario del file**
ID Gruppo	**Identif. del gruppo cui appartiene il proprietario**
Dimensione	**Dimensione in byte del file**
Indirizzi	**Informazioni di indirizzamento ai dati del file**
Ultimo accesso	**Tempo dell'ultimo accesso ai dati del file**
Ultima modifica	**Tempo dell'ultima modifica ai dati del file**
Ult. mod. stato	**Tempo dell'ultima modifica all'inodo**

Accesso/creazione di File

Quando un processo si riferisce ad un file per nome (path assoluto o relativo), il kernel suddivide il path componente per componente, controllando i permessi di accesso alle directoryrelative.

In caso di esito positivo, il kernel:

- ritrova l'inodo del file, se il file esiste e il processo chiede di accedervi;
- assegna un inodo non usato, se il processo chiede di creare un nuovo file.

In entrambi i casi, il kernel restituisce al processo un intero non negativo, detto descrittore del file, che resta associato al file (attraverso il relativo inodo) fino a quando il file non viene rilasciato.

E' attraverso i descrittori che il kernel accede ai file e ne permette le

I/O di basso livello

- La maggior parte delle operazioni sui file ordinari in ambiente UNIX si possono eseguire utilizzando solo le cinque chiamate di sistema open, read, write, lseek, close.
- Il kernel associa un *file descriptor ad ogni file aperto*
  - *Il file descriptor è un intero*
  - *Quando un file viene aperto con open, la funzione open restituisce il file descriptor associato al file*
- *Le costanti simboliche STDIN_FILENO (0), STDOUT_FILENO (1) e STDERR_FILENO (2) sono definite in unistd.h*

Descrittori di file

Alla richiesta di aprire un file esistente o di creare un nuovo file, il kernel ritorna un **descrittore di file** al processo chiamante

Quando si vuole **leggere o scrivere** su un file si passa come argomento a **read** e **write** il descrittore ritornato da **open**

Per convenzione il descrittore 0 viene associato allo standard input, 1 allo standard output e 2 allo standard error

I numeri 0, 1 e 2 possono essere sostituiti dalle costanti

STDIN_FILENO STDOUT_FILENO STDERR_FILENO

definite nell'header <unistd.h>

La funzione **open**

```
#include <sys/types.h>  /*data types*/
#include <sys/stat.h>   /*  data returned by stat()*/
#include <fcntl.h>      /*  file control options */

int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

- Restituisce il descrittore del file, -1 in caso di errore;
- Permette sia di aprire un file già esistente che di creare il file nel caso in cui questo non esista;
- pathname è il pathname (assoluto o relativo) del file;
- oflag permette di specificare le opzioni, mediante costanti definite in <fcntl.h>, combinate con “|” (or bit-a-bit);
- mode è il modo del file ed è un parametro opzionale, utilizzato solo nel caso di creazione del file (“…” modo ISO C per dire variabile).

Chiamata di sistema open

oflag può assumere diversi valori (definiti nell'header <fcntl.h>):

- O_RDONLY apri solo in lettura
- O_WRONLY apri solo in scrittura
- O_RDWR apri in lettura e scrittura

Solo una delle precedenti costanti può essere specificata, con una combinazione OR di:

- O_APPEND esegue un appen dalla fine del file per ciascuna write
- O_CREAT crea il file se non esiste
- O_EXCL se utilizzato insieme a O_CREAT, ritorna un errore se il file esiste
- O_TRUNC se file esiste e aperto con successo write-only/read-write

I flag di **open**

O_RDONLY	Apre il file in sola lettura. Questa opzione non puo' essere combinata con O_WRONLY e O_RDWR.
O_WRONLY	Apre il file in sola scrittura. Questa opzione non puo' essere combinata con O_RDONLY e O_RDWR.
O_RDWR	Apre il file in lettura e scrittura. Questa opzione non puo' essere combinata con O_WRONLY e O_RDONLY.
O_APPEND	L'offset del file e' posto alla fine del file prima di ogni chiamata a write
O_CREAT	Crea il file se esso non esiste. Richiede che open sia utilizzata con l'argomento mode, per la specifica del modo del file.
O_EXCL	Genera un errore se e' stata specificata anche l'opzione O_CREAT e se il file gia' esiste.
O_TRUNC	Tronca il file a zero byte se esso esiste ed e' stato aperto in sola scrittura o in lettura-scrittura.
O_NOCTTY	Se pathname si riferisce ad un terminale, non lo alloca come terminale di controllo per il processo.
O_NONBLOCK	Se pathname si riferisce ad una FIFO o ad un file di dispositivo, definisce la modalita' nonblocking per l'apertura e l'I/O sul file.
O_SYNC	Specifica che ogni chiamata write deve attendere per il completamento dell'I/O sul dispositivo fisico.

I permessi per **open**

S_IRWXU	Permesso di lettura, scrittura ed esecuzione per il proprietario
S_IRUSR	Permesso di lettura per il proprietario
S_IWUSR	Permesso di scrittura per il proprietario
S_IXUSR	Permesso di esecuzione per il proprietario
S_IRWXG	Permesso di lettura, scrittura ed esecuzione per il gruppo
S_IRGRP	Permesso di lettura per il gruppo
S_IWGRP	Permesso di scrittura per il gruppo
S_IXGRP	Permesso di esecuzione per il gruppo
S_IRWXO	Permesso di lettura, scrittura ed esecuzione per gli altri
S_IROTH	Permesso di lettura per gli altri
S_IWOTH	Permesso di scrittura per gli altri
S_IXOTH	Permesso di esecuzione per gli altri

Esempi di open

- open(“prova.txt”, O_RDONLY)
- open(“prova.txt”, O_RDONLY | O_CREAT, S_IRWXU)
- open(“prova.txt”, O_RDWR | O_CREAT | O_EXCL, S_IRWXU)

creat

Un nuovo file può essere creato anche con:

```
#include <fcnl.h>
int creat(const char *pathname, mode_t mode)
```

Equivalente a:

```
open(pathname, O_WRONLY| O_CREAT | O_TRUNC, mode);
```

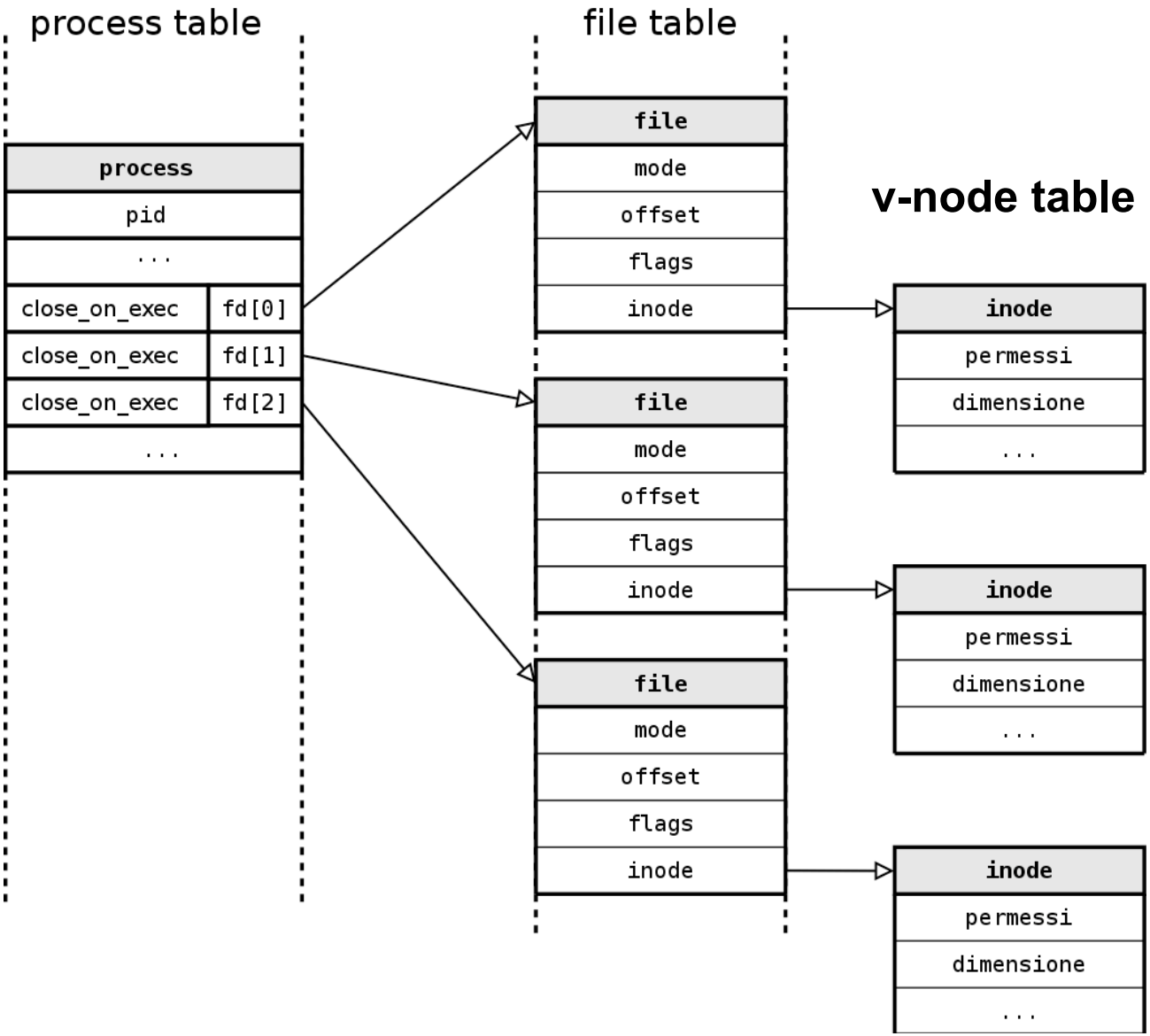
Il nuovo file è aperto solo per scrittura

Implementazione nel kernel

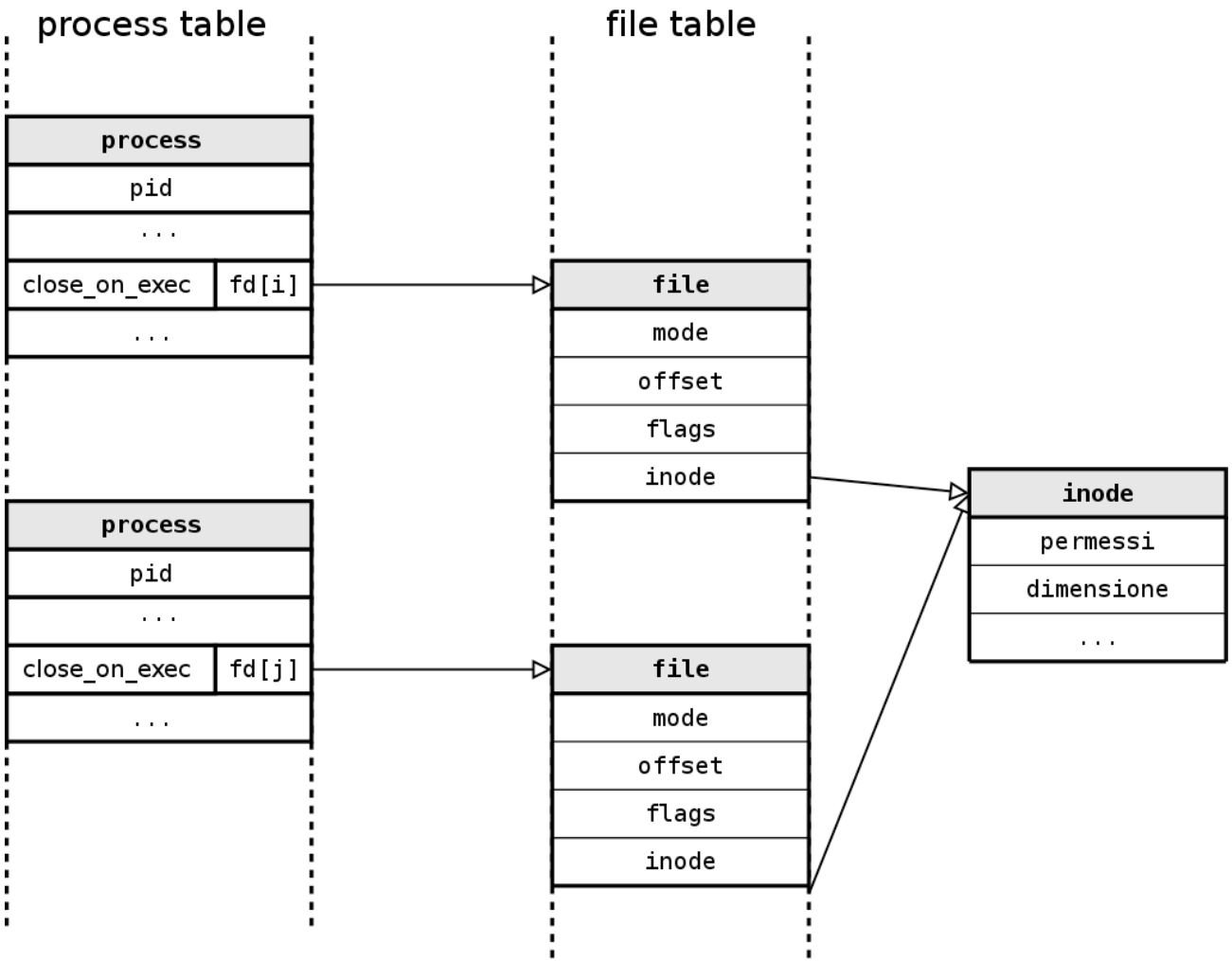
- Il kernel usa due strutture dati indipendenti per gestire i file aperti
  - Ogni processo mantiene la **lista dei propri file descriptor**

    (chiave astratta per accedere ai file, in POSIX, intero)
  - Ogni file descriptor punta ad un elemento della **file table**
  - La **file table** specifica per ogni file aperto:
    - la modalità di apertura del file (lettura, scrittura o entrambe)
    - le opzioni come O_APPEND, etc.
    - l'offset corrente
    - l'inode corrispondente

Un processo con 3 descrittori aperti



Due processi che accedono allo stesso file



Trattare gli errori

- Molte system call restituiscono -1 in caso di errore
- Per avere piu' informazioni, si usa la variabile globale ***errno*** (error number)
- La funzione ***perror***(const char *) stampa la stringa passata come parametro, e poi un messaggio in base al valore corrente di errno
- Esempi:

```
int fd = open("prova.txt", O_RDNLY);
if (fd<0) perror("errore di open");
```

```
int fd;
if ( (fd=open("prova.txt", O_RDNLY)) < 0)
        perror("errore di open");
```

close

```
#include <unistd.h>
int close(int filedes)
```

- Chiude il file identificato da filedes e precedentemente aperto con open
- Restituisce 0 in caso di successo o -1 in caso di errore

L'offset

- Ad ogni file aperto e' associato un intero, detto *offset*, che rappresenta la posizione (*espressa in numero di byte dall'inizio del file*) in cui verra' effettuata la prossima operazione di I/O
- L'offset e' inizializzato a zero da open
  - a meno che non sia specificato O_APPEND
- Le operazioni di read e write incrementano il valore dell'offset di un numero di byte pari al numero di byte letti/scritti

lseek

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int filedes, off_t offset, int whence);
```

- Modifica l'offset corrente del file
- Restituisce il nuovo valore dell'offset, o -1 in caso di errore

lseek

Il valore del parametro **offset** e' interpretato in base al parametro **whence**:

- SEEK_SET: L'offset corrente e' posto a **offset** byte dall'inizio del file.
- SEEK_CUR: L'offset corrente e' incrementato di **offset** byte.
  - Il valore del parametro **offset** puo' essere sia positivo che negativo
- SEK_END: L'offset e' posto a **offset** byte dalla fine del file.
  - Il valore del parametro **offset** puo' essere sia positivo che negativo

Per conoscere l'offset corrente, e' sufficente eseguire:

```
off_t currpos;
currpos = lseek(filedes, 0, SEEK_CUR);
```

Esempi

```
$./a.out
seek OK
$./a.out < /etc/passwd
seek OK
```

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

int main(void) {
 if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
   printf("cannot seek\n");
 else
   printf("seek OK\n");
exit(0);
}
```

Testa lo stdio per vedere se può fare seeking

read

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);
```

- Restituisce:
  - il numero di byte effettivamente letti
  - 0 se ci troviamo alla fine del file
  - -1 in caso di errore
- L'operazione di lettura avviene partendo dall'offset corrente del file
  - L'offset viene incrementato opportunamente

read

Il numero di byte letti può essere diverso dal paramentro nbytes quando:

- Il numero di byte ancora presenti nel file è inferiore ad nbytes.
- La lettura avviene da un terminale (si legge una riga alla volta).
- La lettura avviene da un buffer di rete (nbyte superiore)
- La lettura avviene da una pipe o una FIFO
- L'operazione e' interrotta da un segnale.

write

```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes);
```

- Restituisce:
  - il numero di byte effettivamente scritti
  - -1 in caso di errore
- L'operazione di scrittura avviene partendo dall'offset corrente del file
  - L'offset viene incrementato opportunamente

Esempio 9

```
#include <stdio.h>     /* perror */
#include <errno.h>     /* perror */
#include <unistd.h>    /* write, lseek, close, exit */
#include <sys/types.h> /* open, lseek */
#include <sys/stat.h>  /* open */
#include <fcntl.h>     /* open */

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int main(void)
{
        int fd;
```

Esempio 9

```
  if ((fd = open("file.hole", O_RDWR|O_CREAT, S_IRWXU)) < 0)
        perror("open error");

  if (write(fd, buf1, 10) != 10)
     perror("buf1 write error");
  /* L'offset ora e' 10 */

  if (lseek(fd, 20, SEEK_SET) == -1)
     perror("lseek error");
  /* L'offset ora e' 20 */

  if (write(fd, buf2, 10) != 10)
    perror("buf2 write error");
  /* L'offset ora e' 30 */

  close(fd);
  return 0;
}
```

Apre un file;
Scrive il buf1;
Sposta l’offest (buco);
Scrive il buf2.

Esempio 10

```
#include <stdio.h>     /* perror */
#include <errno.h>     /* perror */
#include <unistd.h>                    /* read, write */
#define BUFFSIZE 4096

int main(void) {
 int n;
 char buf[BUFFSIZE];

 while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
      if (write(STDOUT_FILENO, buf, n) != n)
            perror("write error");

 if (n < 0)
      perror("read error");
 return 0;
}
```

Esercizi

- Scrivere un programma che mostra il contenuto di un file a byte alterni (un carattere si e uno no)
- Scrivere un programma che mostra il contenuto di un file alla rovescia, cioe' a partire dall'ultimo carattere fino ad arrivare al primo

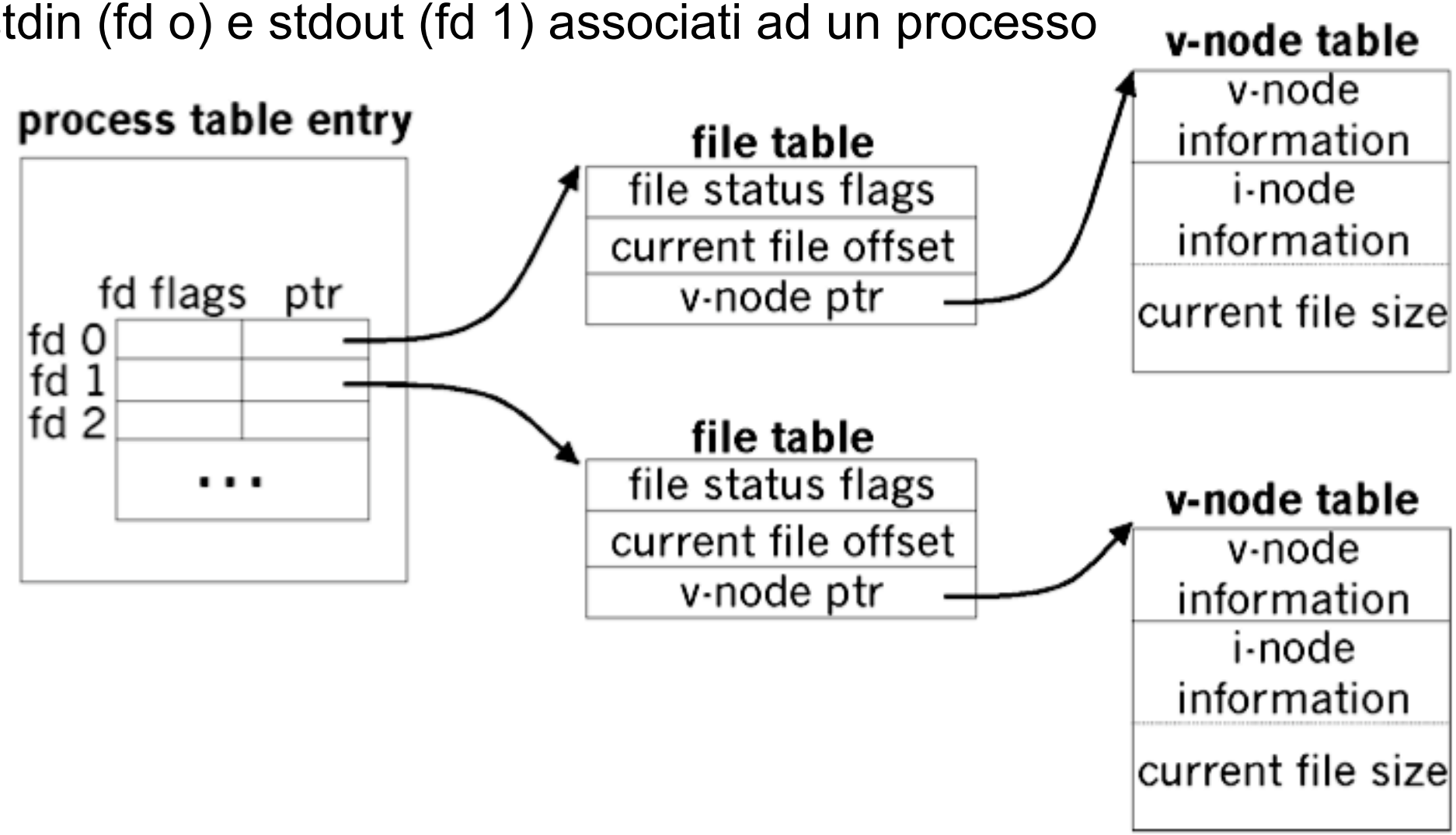
Condivisione di file

Il kernel utilizza tre strutture dati per la gestione dell' I/O:

1. Ciascun processo ha un elemento nella **tabella dei processi**. Tale elemento è un "vettore" di descrittori di file aperti, ciascuno con: un puntatore ad un elemento della **tabella dei file**
2. Il kernel possiede una tabella per ciascun file aperto con i flag di stato del file (lettura, scrittura, append,...), l'offset corrente ed un puntatore ad un elemento della **tabella dei v-node**
3. Ciascun file aperto (o device) ha una **struttura v-node**. Il v-node contiene informazioni sul tipo di file e sulle funzioni che operano su di esso (informazione in i-node).

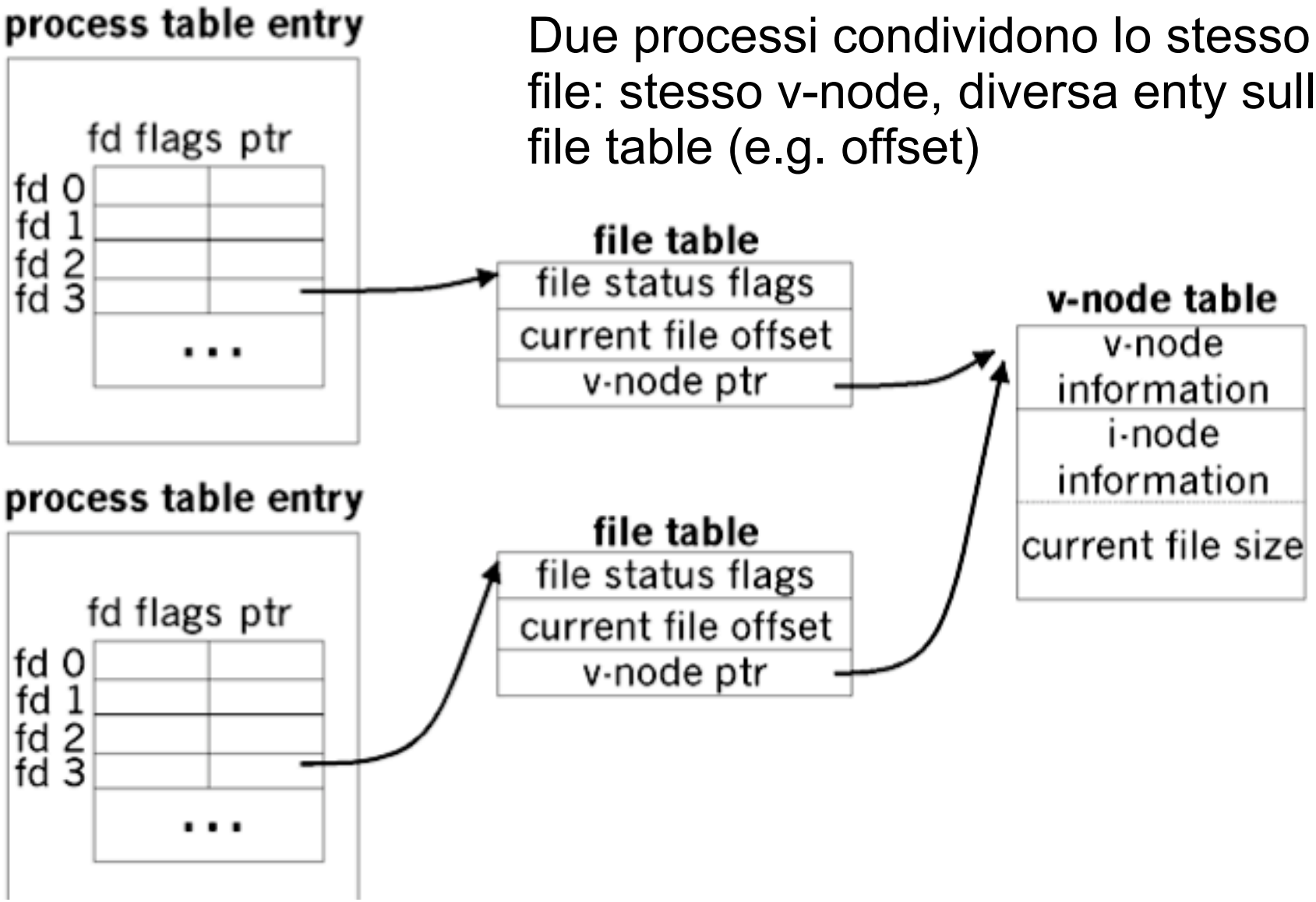
Condivisione di File

stdin (fd o) e stdout (fd 1) associati ad un processo



Condivisione di File

Due processi condividono lo stesso file: stesso v-node, diversa enty sulla file table (e.g. offset)



Accesso ad un file

- Cosa accade quando un processo cerca di accedere ad un file?
- Quando un processo accede ad un file mediante una write, l'elemento della **tabella dei file** relativo all'**offset** viene aggiornato e, se necessario (modificato size), viene aggiornato l‘i-node
- Se il file è aperto con O_APPEND, un flag corrispondente è messo nella **tabella dei file** (ogni write alla fine del file)
- Una chiamata ad lseek modifica solo l'**offset** corrente del file e *non viene eseguita nessuna operazione di I/O*.
- Se si chiede di posizionarsi alla fine del file, il valore corrente dell'offset nella tabella dei file viene preso dal campo della tavola di i-node che descrive la dimensione

Programma A

```
strcpy(string,"aaaaaaaaa\n");
fd=open("testfile",O_RDWR|O_CREAT|O_APPEND,
   S_IRUSR|S_IWUSR);
if (fd<0){ perror("Errore in apertura"); exit(1); }
do {
   if (write(STDOUT_FILENO,"Comando:",8)<8)
            perror("write error");
   input=getchar();
   __fpurge(stdin);
   string[0]=input;
   write(fd,string,10);
   lseek(fd,(off_t)3,SEEK_SET); // sposta l'offset ad “INIZIO” file
   if (write(STDOUT_FILENO,"Eseguito\n",9)<9)
             perror("write error");
} while (input!='f');
close(fd);
```

Programma B

```
strcpy(string,"bbbbbbbbb\n");
fd=open("testfile",O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
lseek(fd,0,SEEK_END); // sposta l'offset a FINE file
do {
   if (write(STDOUT_FILENO,"Comando:",8)<8)
            perror("write error su stdout");
   input=getchar();
   __fpurge(stdin);
   string[0]=input;
   if (write(fd,string,10)<10)
            perror("write error");
    if (write(STDOUT_FILENO,"Eseguito\n",9)<9)
            perror("write error su stdout");
} while (input!='f');
close(fd);
```

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - A scrive 5 stringhe
- Otterrete il seguente output:

qaaaaaaaa
waaaaaaaa
eaaaaaaaa
raaaaaaaa
taaaaaaaa

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - A scrive 5 stringhe
  - B scrive 7 stringhe

- Otterrete il seguente output:

```
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
```

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - A scrive 5 stringhe
  - B scrive 7 stringhe
  - A scrive 5 stringhe
  - B scrive 5 stringhe

- Otterrete il seguente output:

```
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
8bbbbbbbb
9bbbbbbbb
0bbbbbbbb
1bbbbbbbb
2bbbbbbbb
```

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - A scrive 5 stringhe
  - B scrive 7 stringhe
  - A scrive 5 stringhe

- Otterrete il seguente output:

1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
qaaaaaaaa
waaaaaaaa
eaaaaaaaa
raaaaaaaa
taaaaaaaa

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - A scrive 5 stringhe
  - B scrive 7 stringhe
  - A scrive 5 stringhe
  - B scrive 5 stringhe
  - Termina B
  - Termina A
- Otterrete il seguente output:

1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
8bbbbbbbb
9bbbbbbbb
0bbbbbbbb
1bbbbbbbb
2bbbbbbbb
fbbbbbbbb
faaaaaaaa

Ed invertendo l'ordine in cui A e B scrivono nel file ?

Esempio

- Esenguendo i due programmi nel seguente ordine:
  - Esegui A
  - Esegui B
  - B scrive 2 stringhe
  - A scrive 2 stringhe
  - B scrive 2 stringhe
  - A scrive 2 stringhe
  - Termina B
  - Termina A

1bbbbbbbb

2bbbbbbbb

3bbbbbbbb

4bbbbbbbb

fbbbbbbbb

yaaaaaaaa

faaaaaaaa

Duplicazione di File descriptor

- Un file descriptor puo' essere duplicato utilizzando:

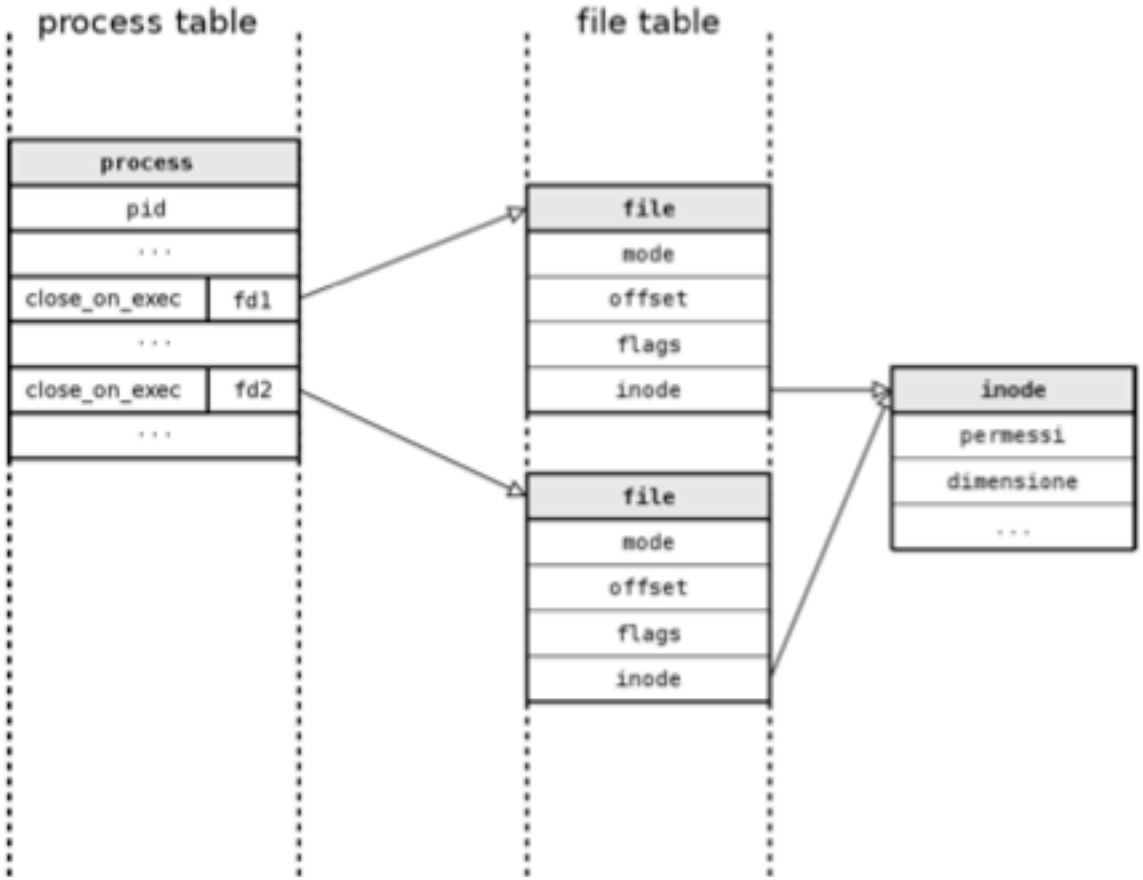
```
#include <unistd.h>
int dup (int filedes);
int dup2(int filedes, int filedes2);
```

- dup ritorna un file descriptor che punta allo stesso file indirizzato da *filedes*.
  - Il valore ritornato da dup e' il *minimo file descriptor* non utilizzato
- dup2 prende in input *filedes2*, il file descriptor da usare nella duplicazone.
  - Se *filedes2* e' aperto, dup2 chiude il file prima di duplicare il descrittore *filedes,* se *filedes* = *filedes2* ritorna fildes e non chiude
  - dup2 e' una operazione atomica

Open e Dup

fd1 = open(“file”, O_RDONLY);

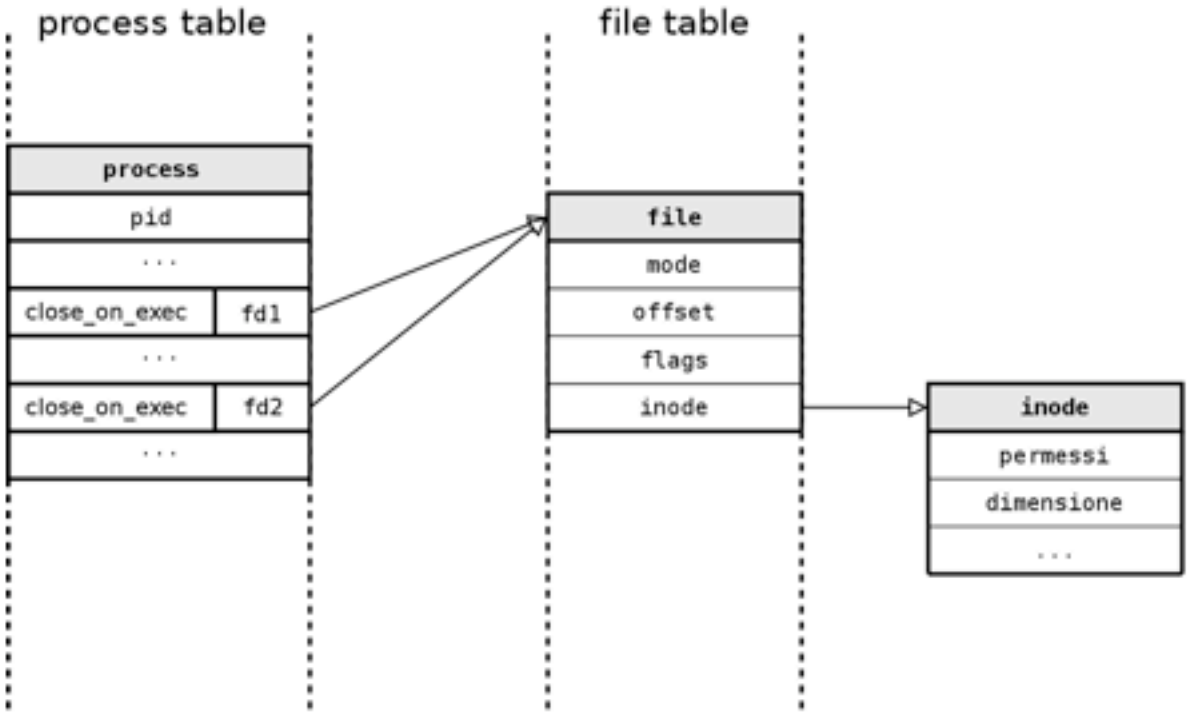
fd2 = open(“file”, O_WRONLY);



Open e Dup

fd1 = open(“file”, O_RDONLY);

fd2 = dup(fd1);



Programma A'

```
strcpy(string,"aaaaaaaaa\n");
fd=open("testfile",O_RDWR|O_CREAT|O_APPEND,
   S_IRUSR|S_IWUSR);
if (fd<0){ perror("Errore in apertura"); exit(1); }
dup2(fd, STDOUT_FILENO); // redirige stdout su fd
do {
   if (write(STDOUT_FILENO,"Comando:",8)<8)
            perror("write error");
   input=getchar();
   __fpurge(stdin);
   string[0]=input;
   write(fd,string,10);
   lseek(fd,(off_t)3,SEEK_SET); // sposta l'offset ad “INIZIO” file
   if (write(STDOUT_FILENO,"Eseguito\n",9)<9)
             perror("write error");
} while (input!='f');
close(fd);
```

Esecuzione di A'

Il file “testfile” contiene quanto segue:

Comando:aaaaaaaaa
Eseguito
Comando:aaaaaaaaa
Eseguito
Comando:aaaaaaaaa
Eseguito
Comando:aaaaaaaaa
Eseguito
Comando:aaaaaaaaa
Eseguito
Comando:faaaaaaaa
Eseguito

Ottenere info su File

```
int  stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
int fstat(int filedes,           struct stat *buf);
                                 Valori di ritorno: 0 se OK, -1 su errore.
```

- Queste system call prendono in input un puntatore ad una struttura stat che conterrà le informazioni sul file
- `stat` ed `lstat` prendono in input il nome del file
  - lstat dà informazioni sui link simbolici (info su link simbolico, non su file linkato)
- `fstat` prende in input il file descriptor del file
  - Il file deve essere aperto.

La struttura stat

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

La struttura stat

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

Tipo di File

- Regular file:
  - Contiene “dati” di qualche tipo
  - Attenzione: anche gli *eseguibili* sono “regular file”
- Directory file:
  - Contiene nomi e puntatori a inode
  - E' necessario utilizzare system call specifiche per manipolarlo
- Block Special file:
  - Rappresentano particolari device (per es., dischi)

Tipo di File

- Character Special file:
  - Rappresentano particolari device (per es., scheda audio)
- FIFO:
  - (o pipe) Utilizzati per comunicazione tra processi
- Socket:
  - Utilizzati per comunicazione tra processi
- Symbolic Link:
  - Link simbolico (o soft)

Tipo di File

- Il tipo di file associato ad un pathname od un file descriptor e' codificato nel campo st_mode della struttura stat.
- Per interpretare st_mode, si usano le seguenti macro:
  - S_ISREG(m): Is regular file ?
  - S_ISDIR(m): Is directory?
  - S_ISCHR(m): Is character device?
  - S_ISBLK(m): Is block device?
  - S_ISFIFO(m): Is Fifo?
  - S_ISLNK(m): Is symbolic link?
  - S_ISSOCK(m): Is socket?
- Le macro prendono come argomento il campo st_mode

Esempio

```
int main(int argc, char *argv[]){
    int         i;
    struct stat buf;
    char        *ptr;

   for (i = 1; i < argc; i++) {
     printf("%s: ", argv[i]);
     if (lstat(argv[i], &buf) < 0) {
        perror("lstat error");
        continue;
     }
     if (S_ISREG(buf.st_mode))       ptr = "regular";
     else if (S_ISDIR(buf.st_mode)) ptr = "directory";
     else if (S_ISCHR(buf.st_mode)) ptr = "character special";
     ...
     printf("%s\n", ptr);
    }
    return 0;
}
```

Struttura Stat

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

User e Group ID

- Ad ogni *file* sono associati uno User ID (uid) ed un Group ID (gid)
  - memorizzati in `st_uid` e `st_gid` della struttura `stat`.
- Si ricorda che ogni *processo* possiede i seguenti ID:
  - Real user e Real group:
    - Utente (e gruppo) che ha lanciato il processo
  - Effective user ed Effective group
    - Utente (e gruppo) che determina i diritti di accesso del processo

Accesso ai File

- Il campo st_mode codifica i permessi di accesso ai file
- Per accedere ad un file e' necessario:
  - avere diritto di **esecuzione** su TUTTE le directory nel path
    - e.g., /home/utente/LSO/esempio.txt
    - il permesso di **lettura** sulla directory consente di leggere i nomi dei file ma non di aprire un file
  - avere i permessi di accesso specifici per il file
- Per creare un file in una directory
  - permessi di scrittura sulla directory
- Per cancellare un file in una directory
  - permessi di scrittura sulla directory (non sul file!)

Accesso ai File

- L'accesso ai file e' regolato dalla seguente sequenza:
  - Se l'effective user del processo e' 0 (superuser): OK
  - Se l'effective user del processo e' uguale all'owner del file
    - Controlla i permessi per l'utente ed, in caso, nega l'accesso
  - Se l'effective group del processo e' uguale al group del file
    - Controlla i permessi del gruppo ed, in caso, nega l'accesso
  - Altrimenti
    - Controlla i permessi per gli “altri”.

Funzione access

```
#include <unistd.h>

int access(const char *pathname, int mode);

                                Restituisce 0 se OK, -1 su errore
```

- controlla i permessi di accesso ad un file in base ad UID e GID “reali”
  - mentre normalmente valgono UID e GID effettivi
- il parametro mode puo' assumere i valori
  - R_OK, W_OK, X_OK: Lettura, scrittura o esecuzione
  - F_OK: Esistenza

Esempio

```
int main(int argc, char *argv[]){
        if (argc != 2){
                printf("usage: a.out <pathname>\n");
                        return 1;

          }
        if (access(argv[1], R_OK) < 0)
                printf("access error for %s\n", argv[1]);
        else
                printf("read access OK\n");
        if (open(argv[1], O_RDONLY) < 0)
                printf("open error for %s\n", argv[1]);
        else
                printf("open for reading OK\n");
        return 0;
}
```

Esempio

```
lso:~>ls -l /etc/shadow
-r--------    1 root   root  1054 Mar  6 21:09 /etc/shadow
```

```
lso:~>ls -l a.out
-rwxrwxr-x    1 lso      lso   12049 Apr 12 14:38 a.out
```

```
lso:~>./a.out a.out
read access OK
open for reading OK
```

```
lso:~>./a.out /etc/shadow
access error for /etc/shadow
open error for /etc/shadow
```

```
lso:~> su; chmod u+s a.out; chown root.root a.out
lso:~> ls -l a.out
-rwsrwx--x    1 root     root  12049 Apr 12 14:38 a.out
```

```
lso:~>./a.out /etc/shadow
access error for /etc/shadow
```

chmod fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int  chmod(const char *path, mode_t mode);
int fchmod(int fildes,       mode_t mode);
```

- Consentono di modificare i permessi di accesso ai file
  - chmod prende in input un pathname
  - fchmod prende in input un file descriptor (file deve essere aperto).
- Il parametro “mode” puo' essere una combinazione delle seguenti costanti:
  - S_ISUID, S_ISGID, S_ISVTX — Set used id exe, group id exe, saved text
  - S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR — owner
  - S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP — group
  - S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH — others

Esempio

```
int main(void){
    struct stat statbuf;

      /* Pone a “uno” il bit set-group ID ed a “zero” il permesso di esecuzione per il
       gruppo */
    if (stat("foo", &statbuf) < 0)
        { printf("stat error for foo"); exit(1)}
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0){
        printf("chmod error for foo");

    /* Pone le protezione a "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        printf("chmod error for bar");
    return 0;
}
```

chown

```
#include <sys/types.h>
#include <unistd.h>
int  chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd,           uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

- Modificano il campo st_uid ed st_gid del file indicato dal pathname (chown e lchown) o dal file descriptor (fchown)
- Se il parametro “owner” o “group” e' uguali a -1, il campo corrispondente non viene modificato
- In molti sistemi, solo un processo del superuser puo' modificare il campo st_uid
- Un processo puo' modificare il gruppo se
  - (a) e' owner del file e
  - (b) il parametro group e' uguale all'effective GID del processo o ad uno dei gruppi “alternativi”